Probabilistic Loop Scheduling for Applications with Uncertain Execution Time

Sissades Tongsima, Edwin H.-M. Sha, Member, IEEE
Chantana Chantrapornchai, David R. Surma Member, IEEE and
Nelson Luiz Passos, Member, IEEE

ABSTRACT - One of the difficulties in high-level synthesis and compiler optimization is obtaining a good schedule without knowing the exact computation time of the tasks involved. The uncertain computation times of these tasks normally occur when conditional instructions are employed and/or inputs of the tasks influence the computation time. The relationship between these tasks can be represented as a data-flow graph where each node models the task associated with a probabilistic computation time. A set of edges represents the dependencies between tasks. In this research, we study scheduling and optimization algorithms taking into account the probabilistic execution times. Two novel algorithms, called *probabilistic retiming* and *probabilistic rotation scheduling*, are developed for solving the underlying non-resource and resource constrained scheduling problems respectively. Experimental results show that probabilistic retiming consistently produces a graph with a smaller longest path computation time for a given confidence level, as compared with the traditional retiming algorithm that assumes a fixed worst-case and average-case computation times. Furthermore when considering the resource constraints and probabilistic environments, probabilistic rotation scheduling gives a schedule whose length is guaranteed to satisfy a given probability requirement. This schedule is better than schedules produced by other algorithms that consider worst-case and average-case scenarios.

KEYWORDS - Scheduling, loop pipelining, probabilistic approach, retiming, rotation scheduling.

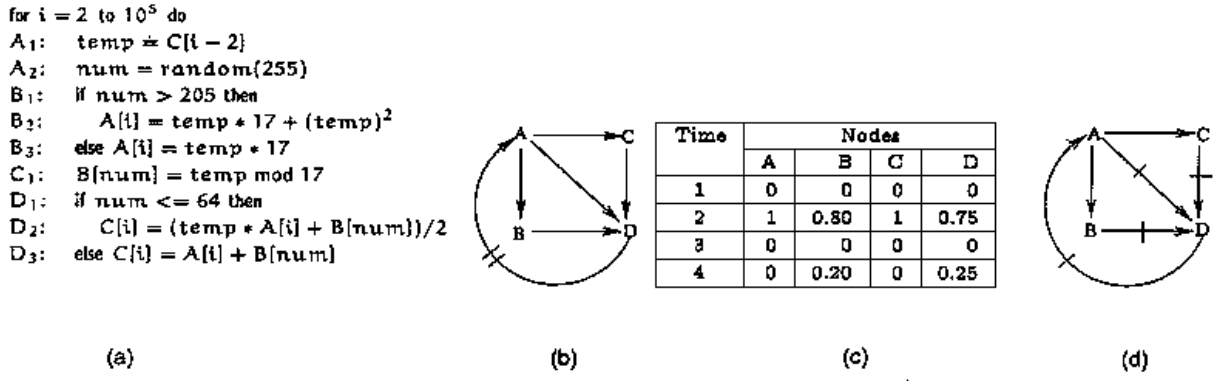
1. Introduction

In many practical applications such as interface systems, fuzzy systems, artificial intelligence systems, and others, the required tasks normally have uncertain computation times (called *uncertain* or *probabilistic* tasks for brevity). Such tasks normally contain conditional instructions and/or operations that could take different computation times for different inputs. A dynamic scheduling scheme may be considered to address the problem; however, the decision of the run-time scheduler which depends on the local on-line knowledge may not give a good overall schedule. Although many static scheduling techniques can thoroughly check for the best assignment for dependent tasks, existing methods are not able to deal with such uncertainty. Therefore, either worst-case or average-case computation times for these tasks are usually assumed. Such assumptions, however, may not be applicable for the real operating situation and may result in an inefficient schedule.

For iterative applications, statistics for the uncertain tasks are not difficult to collect. In this paper, two novel loop cheduling algorithms, *probabilistic retiming* (PR) and *probabilistic rotation scheduling* (PRS), are proposed to statically schedule these tasks for non-resource (assume *unlimited* number of target processors) and resource constrained (assume *limited* number of target processors) systems respectively. These algorithms expose the parallelism of the probabilistic tasks across iterations as well as take advantage of the inherent statistical data. For a system without resource constraints, PR can be applied to optimize the input graph (i.e., reduce the length of the longest path of the graph such that the probability of the longest path computation time being less than or equal to some given computation time, c, is greater than or equal to a given confidence probability θ). The resulting graph implies a schedule for the non-resource constrained system where the longest path computation time determines its *schedule length*. On the other hand, the PRS algorithm is used to schedule uncertain tasks to a fixed number of multiple processing elements. It produces a schedule length from the given graph and incrementally reduces the length so that the probability of it being less than the previous length is greater than or equal to the given confidence probability.

- *S. Tongsima is with the National Electronics and Computer Technology Center, National Science and Technology Development Agency, Bangkok 10400, Thailand. E-mail: stongsim@hpcc.nectec.or.th.*
- *E.H.-M. Sha is with the Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556. E-mail: esha@cse.nd.edu.*
- *C. Chantrapornchai is with the Faculty of Sience, Silpakorn University, Noakorn Pathom 73000, Thailand.*
- *D.R. Surma is with the Department of Mathematics and Computer Science, Valparaiso University, Valparaiso, IN 46383.*
- *N.L. Passos is with the Department of Computer Science, Midwestern State University, Wichita Falls, TX 76308.*

This paper is reprinted from a paper published in IEEE Transaction on Computer. Vol. 49, No. 1, January 2000.



Time	Nodes			
	A	B	C	D
1	0	0	0	0
2	1	0.80	1	0.75
3	0	0	0	0
4	0	0.20	0	0.25

Figure 1. Illustrates a sample code segment, the corresponding PG and its computation time, and the retimed graph. (a) Code segment. (b) PG. (c) Timing information. (d) Retimed PG.

In order to be compatible with the current high performance parallel processing technology, we assume that synchronization is required at the end of each iteration. Such a parallel computing style is also known as *synchronous parallelism* [19], [10]. Both PR and PRS take an input application which can be modeled as a probabilistic data-flow graph (PG), which is a generalized version of a data-flow graph (DFG) where a node corresponds to a task (a collection of statements), and a set of edges representing dependencies between these tasks and determine a schedule. The loop-carried dependences (dependency distances) between tasks in different iterations are represented by short bar lines on the corresponding edges. Since the computation times of the nodes can be either fixed or varied, a probability model is employed to represent the timing of the task.

Fig.1b shows an example of a PG consisting of four nodes. Note that such a graph models the code segment presented in Fig. 1a, where, for example, A in the PG corresponds to A_1 and A_2 of the code segment. Two bar lines on the edge between nodes D and A represent the dependency distances between these two nodes. The computation time of nodes A and C are known to be fixed (2 time units). In this code, the uncertainty occurs in the computation of nodes B and D. Assume that each arithmetic operation and the assignment operation (=) take 1 time unit. Furthermore, the computation time of the comparison and random number generating operations are assumed negligible. Hence, it may take either 4 or 2 time units to execute node B. Put another way, about 20 percent of the time (51 out of 256), statement B_2 will be executed and node B will take 4 time units; otherwise node B takes only 2 time units (B_3 has only one operation). Likewise, approximately 25 percent (64 out of 256), node D takes 4 time units, and about 75 percent, it will take 2 time units. Each entry in Fig. 1c shows a probability associated with each node's possible computation time (the probability distribution). By taking into account these varying timing characteristics, the proposed technique can be applied to a wide variety of applications in high-level synthesis and compiler optimization.

Considerable research has been conducted in the area of finding a schedule of a directed-acyclic graph (DAG) for multiple processing systems. (Note that DAGs are obtained from DFGs by ignoring edges of a DFG containing one or more dependency distances.) Many heuristics have been proposed to schedule DAGs, e.g., list scheduling, graph decomposition [13], [11], etc. These methods, however, consider neither exploring the parallelism across iterations nor addressing the problems of probabilistic tasks.

For *instruction level parallelism* (ILP) scheduling, trace scheduling [9] is used to globally schedule DAGs by rearraging some operations in the graphs. Percolation scheduling is used in a development environment [1] for microcode compaction, i.e., parallelism extraction of horizontal microcode. Nevertheless, the graph model used in these techniques does not reflect the uncertainty in node computation times. In the class of global cyclic scheduling, software pipelining [16] is used to overlap instructions, whereby the parallelism is exposed across iterations. This technique, however, expands the graph by unfolding or unrolling [22] it resulting in a larger code size. Loop transformations are also common techniques used to construct parallel compilers. They restructure loops from the repetitive code segment in order to reduce the total execution time of the schedule [2], [3], [20], [27], [28]. These techniques, however, do not consider that the target systems have limited number of processors or that task computation times are uncertain.

Modulo scheduling [24], [25], [26] is a popular technique in compiler design for exploiting ILP in loops which results in optimized codes. This framework specifies a lower bound, called initiation interval (II), to start with and strives to schedule nodes based on such knowledge. Much research was introduced to improve and/or expand the capability of modulo scheduling. For example, research was presented which improved modulo scheduling by producing schedules while considering limited number of registers [7], [8], [21]. In [17], a combination of modulo scheduling and loop unrolling was introduced and applied in the IMPACT compiler [4]. These ILP approaches, however, are limited to solving problems without considering uncertain computation times (probabilistic graph model).

Some research considers the uncertainty inherit in the computation time of nodes. Ku and De Micheli [14], [15] proposed a relative scheduling method which handles tasks

with unbounded delays. Nevertheless, their approach considers a DAG as an input and does not explore the parallelism across iterations. Furthermore, even if the statistics of the computation time of uncertain nodes is collected, their method will not exploit this information. A framework that is able to handle imprecise propagation delays is proposed by Karkowski and Otten [12]. In their approach, fuzzy set theory [29] was employed to model the imprecise computation times. Although their approach is equivalent to finding a schedule of imprecise tasks to a non-resource constrained system, their model is restricted to a simple triangular fuzzy distribution and does not consider probability values.

For scheduling under resource constraints, the *rotation scheduling* technique was presented by Chao et al. [5], [6] and was extended to handle multi-dimensional applications by Passos et al. [23]. Rotation scheduling attempts to pipeline a loop by assigning nodes from the loop to the system with a limited number of processing elements. It implicitly uses traditional retiming [18] in order to reduce the total computation time of the nodes along the longest paths (also called the critical paths), in the DFG. In other words, the graph is transformed in such a way that the parallelism is exposed but the behavior of the graph is preserved. In this paper, the rotation scheduling technique is extended so that it can deal with uncertain tasks.

Since the computation time of a node in a PG is a random variable, the total computation time of this graph is also a random variable. The concept of a *control step* (the synchronization of the tasks "within" each iteration) is no longer applicable. A schedule conveys only the execution *order* or *pattern* of the tasks being executed in a functional unit and/or between different units. In order to compute the total computation time of this ordering, a *probabilistic task-assignment graph* (PTG) is constructed. A PTG is obtained from a PG in which non-zero dependency distance edges are ignored and each node is assigned to a specific functional unit in the system. The PTG also contains additional edges, called *flow-control* edges where a connection from *u* to *v* means that *u* is executed immediately before *v* using the same functional unit. Note that in the non-resource constrained scenario, the PTG will be the DAG portion of the PG (a subgraph that contains only no dependency distance edges).

Let us use the example in Fig. 1b. Assume that the term longest path computation time entails finding the maximum of the summation of computation times of nodes along paths which contain no dependency distances. After examining all possible longest paths of this graph, it is likely (60 percent) that its longest path computation time is less than or equal to 8. The details of how this value is determined is given in Section 3. Note that if all nodes in this graph are assigned their worst-case values, the longest path computation time (or schedule length for non-resource constrained systems) of this graph will be 10. One might wish to reduce the longest path of this graph in nearly all cases, for example reducing the chance of the clock period being greater than 6. By applying probabilistic retiming, the longest path computation time of the graph may be improved with respect to the given constraint. The modified graph after retiming is shown in Fig. 1d. The longest path computation time of this graph is less than than or equal to 6 with 20 percent chance.

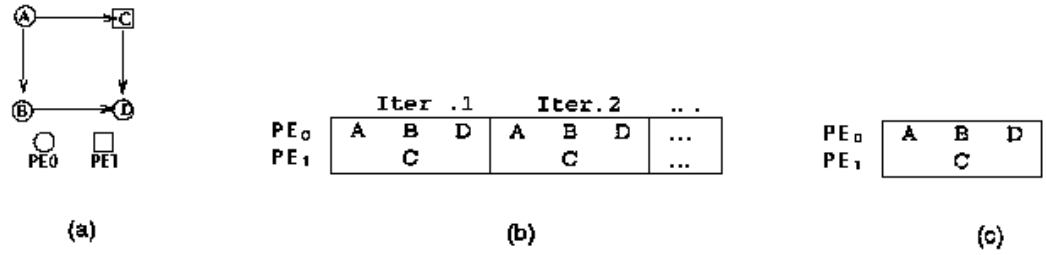


Figure 2. Illustrates an example of PTG, its corresponding repeated pattern, and the static execution order. (a) The PTG. (b) Initial execution pattern. (c) Schedule.

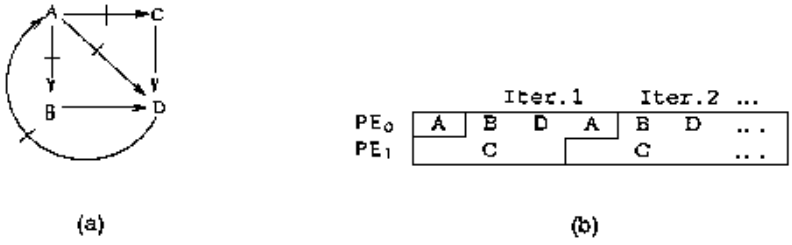


Figure 3. Illustrates the corresponding retimed PG and the repeated pattern after changing iteration window. (a) Rotate A. (b) Reshaping iteration window.

If we need to schedule nodes from the PG to two homogeneous functional units, a possible PTG can be constructed as shown in Fig. 2a. Since the input graph is cyclic, an *execution pattern* of this PTG is repeated and the synchronization is applied at the end of each iteration, as shown in Fig. 2a. The solid edges in this PTG represent those zero dependency distance edges, called dependency edges, from the input graph (see Fig. 1b). In this figure, nodes *A, B* and *D* are assigned to PE_0 and node *C* is bound to PE_1. Note that *D* is implicitly executed after *A*; therefore, the direct edge from *A* to *D* from the original input graph can be omitted. A corresponding static schedule which shows only one iteration from the execution pattern is shown in Fig. 2c.

The resulting longest path computation time of the PTG is less than 9 units with 90 percent certainty. This longest path timing and its probability are also known as a schedule length for resource constrained systems. We can improve the resulting schedule length by applying our probabilistic rotation scheduling algorithm to the PG and its PTG. In this case the algorithm first selects the root node *A* to be rescheduled. Then one dependency distance from the incoming edges of node *A* is moved to all its outgoing edges. Fig.3a. shows the resulting transformation graph of the PG. This new graph will be used as a reference to later update the PTG. The new execution pattern is equivalent to reshaping the iteration window as presented in Fig. 3b.

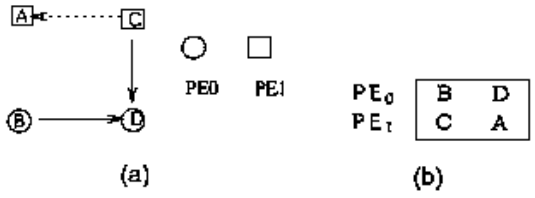


Figure 4. Illustrates the resulting PTG and its execution order after rescheduling A. (a) PTG. (b) Static execution order.

By applying the PRS algorithm, node *A* from the next iteration (see Fig. 3b.) is introduced to the static execution pattern. Note that node *A* has no inter-iteration dependencies associated with it. Therefore, *A* can be rescheduled to any available functional unit. One possible schedule is to assign node *A* immediately after node *C* in PE_1. The resulting PTG and the new execution order are shown in Fig. 4a and 4b, respectively. The dotted arrow from *C* to *A* in this new PTG represents the flow-control edge. For this PTG, the resulting schedule length will be less than seven with higher than 90 percent confidence.

The remainder of this paper is organized as follows. Section 2 presents the graph model used in this work. Required terminology and fundamental concepts are also presented. Section 3 discusses probabilistic retiming and the algorithm for computing a total computation time of a probabilistic graph. The probabilistic rotation scheduling algorithm and the supported routines will be discussed in Section 4. Experimental results are discussed in Section 5. Finally, Section 6 draws conclusions of this research.

2. Preliminaries

In this section, the graph model which is used to represent tasks with uncertain computation times is introduced. Terminology and notations relevant to this work are also discussed. We begin by examining a DFG that contains tasks with uncertain computation time which can be modeled as a *probabilistic graph* (PG). The following gives the formal definition for such a graph.

Definition 2.1. *A probabilistic graph (PG) is a vertex-weighted, edge-weighted, directed graph* $G = \langle V,E,d,T \rangle$, *where V is the set of vertices representing tasks, E is the set of edges representing the data dependencies between vertices, d is a function from E to the set of non-negative integers, representing the number of dependency distance on an edge, and* T_v *is a random variable representing the computation time of a node* $v \in V$.

Note that traditional DFGs are a special case of PGs where all probabilities equal one. Each vertex $v \in V$ is weighted with a *probability distribution* of the computation time, given by T_v, where T_v is a discrete random variable corresponding to the computation time of v such that $\Sigma_{\forall x} \Pr(T_v = x) = 1$. The notation $\Pr(T_v = x)$ is read "the probability that random variable T assumes value x". The *probability distribution* of T is assumed to be discrete in this paper. The granularity of the resulting probability distribution, if necessary, depends on the needed degree of accuracy.

An edge $e \in E$ from u to v, $u, v \in V$, is denoted by $u \xrightarrow{e} v$ and a path p starting from u and ending at v is indicated by the notation $u \overset{p}{\sim} > v$. The number of dependency distances of path $p(d(p))$, $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{k-1}} v_k$ is $d(p) = \sum_{i=0}^{k-1} d(e_i)$. As an example, Fig. 1b has the set of edges $E = \{A \xrightarrow{e_1} B,\ A \xrightarrow{e_2} C,\ A \xrightarrow{e_3} D,\ B \xrightarrow{e_4} D,\ C \xrightarrow{e_5} D,\ D \xrightarrow{e_7} A\}$. The number of dependency distances on each edge $e \in E$ is given by $d(e)$, where, for $i = 1,\ldots, 6$, $d(e_i) = 0$ and $d(e_7) = 2$.

The *execution order* or *execution pattern* of a PG are determined by the precedence relations in the graph. During one iteration of the graph each vertex in the execution order is computed exactly one time. Multiple iterations are identified by index *i*, starting from 0. Inter-iteration dependencies are represented by weighted edges or dependency distances. For any iteration *j*, an edge *e* from *u* to *v* with dependency distance $d(e)$ conveys that the computation of node *v* at iteration *j* depends on the execution of node *u* at iteration $j - d(e)$. An edge with no dependency distances represents a data dependency within the same iteration. A legal data flow graph must have strictly positive dependency distance cycles, i.e., the summation of the $d(e)$ along any cycle cannot be less than or equal to zero.

2.1 Retiming Overview

Retiming operations rearrange registers in a circuit or dependency distances in a data-flow graph in such a way that the behavior of the circuit is preserved while achieving a faster circuit. Traditionally, retiming [18] optimizes a synchronous circuit (graph) $G = \langle V, E, d, t\rangle$ which has non-probabilistic functional elements, i.e., each of the vertices $v \in V$ is associated with a fixed numerical timing value. The optimization goal is normally to reduce the *clock period* or *cycle period* $\Phi(G)$ (also known as longest path computation time). The cycle period represents the execution time of the longest path (referred to as the critical path) that has all zero dependency distance edges. It is defined by the equations

$\Phi(G) = \max\{t(p) : d(p) = 0\}$, where

$p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{k-1}} v_k$,

$t(p) = \sum_{i=0}^{k} t(v_i)$, and $d(p) = \sum_{i=0}^{k-1} d(e_i)$.

Retiming of a graph $G = \langle V, E, d, t\rangle$ is a transformation function from vertices to the set of integers, $r : V \mapsto Z$. The retiming function describes the movement of dependency distances with respect to the vertices so as to transform G into a new graph $G_r = \langle V, E, d_r, t\rangle$, where d_r represents the number of dependency distances on the edges of G_r. The positive (or negative) value of the retiming function determines the movement of the dependency distances. During retiming the same number of dependency distances is pushed from all incoming (outgoing) edges of a node to all outgoing (incoming) edges. If a single dependency distance is pushed from all incoming edges of node $u \in V$ to all outgoing edges of node u, then $r(u) = 1$. Conversely, if one dependency distance is pushed from all outgoing to all incoming edges of u, then $r(u) = -1$. The absolute value of the retiming function conveys the number of dependency distances that are pushed. An algorithm to find a set of retiming functions to minimize the clock period of the graph presented in [18] is a polynomial time algorithm which has the time complexity of $O(|V|\,|E| \log |V|)$.

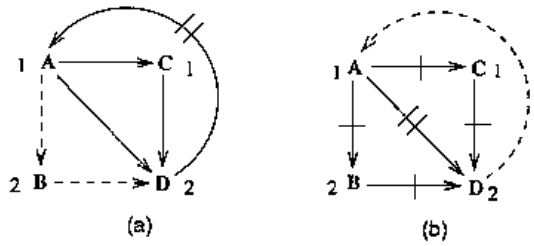


Figure 5. Illustrates retiming transformations ((a) before and (b) after retiming) where dotted edges represent the critical path.

Consider Fig. 5a, which illustrates a simple graph with four vertices, A, B, C and D. The numbers next to the vertices in the figure represent the required computation times. Fig. 5b represents a retimed version of Fig. 5a where $r(B) = r(C) = 1$, $r(A) = 2$, and $r(D) = 0$. In this case, the movement of dependency distances is as follows: $r(A) = 2$ is equivalent to removing two dependency distances from the incoming edge of vertex A, $D \xrightarrow{e} A$ and adding them onto edges $A \xrightarrow{e} B$, $A \xrightarrow{e} C$, and $A \xrightarrow{e} D$. The retiming functions for nodes C and B are $r(B) = r(C) = 1$. This means that one dependency distance from $A \xrightarrow{e} B$ is pushed through vertex B to edge $B \xrightarrow{e} D$. Similarly, one dependency distance from edge $A \xrightarrow{e} C$ is pushed through vertex C to $C \xrightarrow{e} D$. An equivalent set of retimings in Fig. 5b is $r(B) = r(C) = -1$, $r(D) = -2$, and $r(A) = 0$. This equivalent set of retimings produces the same graph by pushing the dependency distances backward through nodes D, B and C, instead of forward through nodes A, B and C. The dotted lines in Fig. 5a represent the critical path of the graph, for which $\Phi(G) = 5$. After retiming, the critical path becomes $\Phi(G) = 3$, as illustrated by the dotted line in Fig. 5b.

The following summarizes some essential properties of the retiming transformation:

1. r is a legal retiming if $d_r(e) \geq 0$, $\forall e \in E$.
2. For an edge $u \xrightarrow{e} v$, where $u, v \in V$, $d_r(e) = d_r(e) + r(u) - r(v)$.
3. For a path $u \overset{p}{\sim} > v$, where $u, v \in V$, $d_r(p) = d_r(p) + r(u) - r(v)$.
4. In any directed cycle (l) of G and G_r, $d_r(l) = d(l) > 0$.

Property 1 guarantees that the retimed graph will not have any edge containing a negative number of dependency distances. Properties 2 and 3 explain the movement of such distances. If $r(v)$, $v \in V$, has a positive value, the distances will be deleted from the incoming edge(s) of v and inserted onto the outgoing edge(s), and vice versa if $r(v)$ has the negative value. Finally, Property 4 ensures that the number of dependency distances in any loop of the graph remains constant. That requires that all cycles have at least one dependency distance. Since retiming is an optimization technique which is subject to unlimited number of target resources, the resulting longest path computation time after the transformation is the underlying schedule length. Consider only a DAG part of the retimed graph where edges with nonzero dependency distances in the retimed graph are ignored. The iteration boundaries of this schedule will be at the root nodes (beginning of the iteration) and at the leaf nodes (end of the iteration).

2.2 Rotation Scheduling

In [5], Chao et al. proposed an algorithm, called *rotation scheduling*, which uses the retiming algorithm to deal with scheduling a cyclic DFG under resource constraints. The input to the rotation scheduling algorithm is a DFG and its corresponding static schedule, i.e., a synchronized order of the nodes in the DFG. Rotation scheduling reduces the schedule length (the number of control steps needed to execute one iteration of the schedule) by exploiting the parallelism between iterations. This is accomplished by shifting the scope of a static schedule in one iteration, called the iteration window, down by one control step. Looking at a

static iteration, rotation scheduling analogously rotates tasks from the top of the schedule of each iteration down to the end. This process is equivalent to retiming those tasks (nodes in the DFG) in which one dependency distance will be deleted from all their incoming edges and added to all their outgoing edges resulting in an intermediate retimed graph. Once the parallelism is extracted, the algorithm reassigns the rotated nodes to new positions so that the schedule length is shorter.

As an example, the cyclic DFG in Fig. 6a is to be scheduled using two processing elements. Fig. 6b presents one possible static schedule for such a graph. By using rotation scheduling, this schedule can be optimized. First, the algorithm uses node A from the next iteration. The original graph is retimed by $r(A) = 1$, i.e., one dependency distance from $E \xrightarrow{e} A$ is moved to all outgoing edges of A (see Fig. 6c). By doing so, node A now can be executed at any control step in this new iteration window. Assume that rotation scheduling uses a remapping strategy that places node A immediately after node C in PE_1. The resulting static schedule length is then reduced by one control step as shown in Fig. 6d. In Section 4, the concept of the schedule length and the remapping strategy will be extended to handle probabilistic inputs.

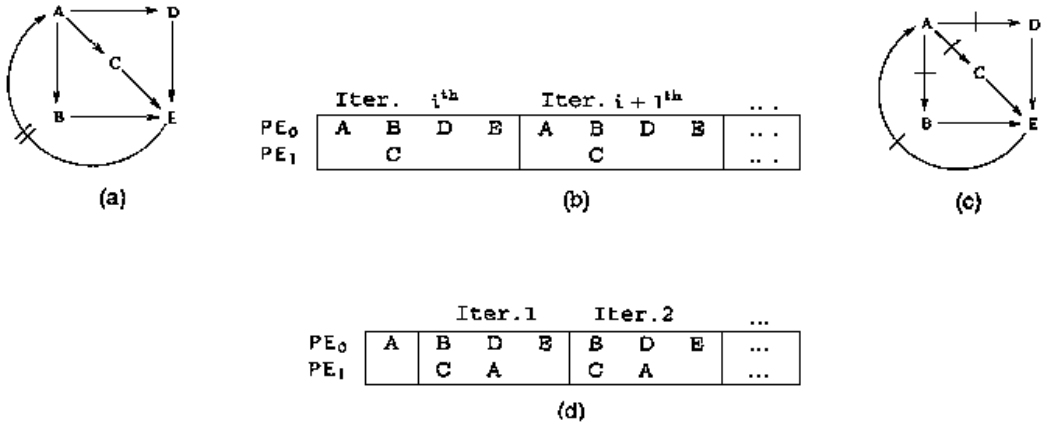


Figure 6. Illustrates an example to represent how rotation scheduling optimizes the underlying schedule length. (a) Cyclic DFG. (b) Static schedule. (c) Retimed. (d) Resulting schedule.

3. Nonresource Constrained Scheduling

Assuming there are infinite available resources, one can optimize a PG with respect to a desired longest path computation time and confidence level, i.e., attempt to reduce the longest path computation time of the graph. The distribution of dependency distances in the PG is done according to a probabilistic timing constraint where the *probability* of obtaining the timing result (longest path computation time) being less than or equal to a given value c is greater than some confidence probability value θ. This resulting timing information is essentially the schedule length of the nonresource constrained problem. This section presents an efficient algorithm for optimizing a probabilistic graph with respect to a desired computation time (c) and its corresponding confidence probability (θ). In order to evaluate the modified graph, we need to know the probability distribution associated with its computation time. The remaining subsections discuss these issues.

3.1 Computing Maximum Reaching Time

Let G_{dag} be the DAG portion (the subgraph that has only edges with no dependency distances) of a probabilistic graph G. Assume that two dummy nodes, v_s and v_d, are added to G_{dag}, where v_s connects to all source nodes (roots) and v_d is connected by all sink nodes (leaves). Traditionally, the longest path computation time of a graph is computed by maximizing the summation of computation times of nodes along the critical (longest) paths between these dummy nodes. Likewise, for a probabilistic graph, we can compute the summation of the computation time for each path from v_s to v_d in the graph. In this case the largest summation value is called the maximum reaching time or mrt of the graph. The mrt of a PG exhibits a possible longest path computation time of the graph and its associated probability. Therefore, unlike the traditional approach, the summation and maximum functions of computation time along the paths in a PG become functions of multiple random variables.

To compute an mrt of a PG, we need to modify the graph so that v_s and v_d are connected to the DAG portion of the original graph. Formally, a set of zero dependency distance edges is used to connect vertex v_s to all roots, and to connect all leaves to vertex v_d. Since it is nontrivial to efficiently compute a function of dependent random variables, Algorithm 1 computes the mrt(G) assuming that the random variables are independent. This algorithm traverses the input graph in the breadth-first fashion starting from v_s and ending at v_d. In general, the algorithm accumulates the probabilistic computation times along each traversed path. When it reaches a node that has more than one parent, all the values associated with its parents are maximized.

Algorithm 1 Calculate maximum reaching time of graph G
Require probabilistic graph PG
Ensure mrt(G) = temp_{mrt} (v_s, v_d)

1. $G_0 = \langle V_0, E_0, d, T\rangle$ such that $V_0 = V + \{v_s, v_d\}$
2. $E_0 = E - \{e \in E : d(e) \neq 0\} + \{v_s \xrightarrow{e} v \in V_r, \quad u \in V_l \xrightarrow{e} v_d\}$
3. $\forall\, u \in V_0$, $\text{temp}_{mrt}(v_s, u) = 0$, $T_{v_s} = T_{v_d} = 0$, $Queue = v_s$
4. **while** *Queue* $\neq 0$ **do**
5. get node u from top of the *Queue*
6. $\text{temp}_{mrt}(v_s, u) = \text{temp}_{mrt}(v_s, u) + T_u$
7. **for all** $u \xrightarrow{e} v$ **do**
8. decrement the incoming degree of node v by one
9. $\text{temp}_{mrt}(v_s, v) = \max(\text{temp}_{mrt}(v_s, u), \text{temp}_{mrt}(v_s, v))$
10. **if** incoming degree of node v becomes 0 **then**
11. insert node v into the *Queue*
12. **end if**
13. **end for**
14. **end while**

Lines 1 and 2 produce DAG G_0 from G containing only edges $e \in E$, with $d(e) = 0$, and the additional zero dependency distance edges connecting v_s to every root node $v \in V_r$ of G and connecting every leaf node $u \in V_l$ of G to v_d. Line 3 initializes the $\text{temp}_{mrt}(v_s, u)$ value for each vertex u in the new graph and sets the computation time of T_{v_s} and T_{v_d} to zero. Lines 4-12 traverse the graph in topological order and compute the mrt of each v with respect to v_s ($\text{temp}_{mrt}(v_s, v_d)$). Note that the temp_{mrt} for node v with respect to v_s is originally set to zero. It stores the current maximum computation time of all node v's visited parents. When the first parent of v is dequeued, v has its indegree reduced by one (Line 8) and temp_{mrt} mrt is updated (Line 9). Vertex v's other parents are in turn dequeued, and the process is repeated. Eventually, the last parent of node v will be dequeued and maximized. At this point, node v will be inserted into the queue since all parents have been considered, i.e., indegree of v equals zero (Line 10). Node v will be eventually dequeued by Line 5. Line 6 will then add T_v to the temp_{mrt} of node v producing the final mrt with respect to all paths reaching node v.

Note that the initial computation times are integers and the probabilities associated with these times being greater than the given value c are accumulated as one value in the algorithm. Only O(c+1) values need to be stored for each vertex. Therefore, the time complexity for calculating the summation (Line 6), or the maximum (Line 9) of two vertices is O(c^2). Since the algorithm computes the result in a breadth first fashion, the running time of Algorithm 1 is O($c^2|V|\,|E|$), while the space complexity is bounded by O(c|V|).

3.2 Probabilistic Retiming

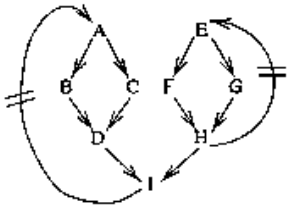
Using the concept of mrt, Algorithm 2 presents the probabilistic retiming algorithm which reduces the longest path computation time of the given PG to meet a timing constraint. Such a constraint is that $\Pr(\text{mrt}(v_s, v_d) < c) > \theta$ where c is the desired longest path computation time of the graph and θ is the confidence probability. This requirement can be rewritten as $\Pr(\text{mrt}(v_s, v_d) < c) \leq \delta$, where δ is 1 - θ. The algorithm retimes vertices whose probability of computation time being greater than c is larger than the acceptable probability value. Initially, the retiming value for each node is set to zero and nonzero dependency distance edges are eliminated. Then, v_s is connected to the root-vertices of the resulting DAG and v_d is connected by the leaf-vertices of the DAG. Lines7-17 traverse the DAG in a breath-first search manner and update the temp_{mrt} for each node as in Algorithm 1. After updating a vertex, the resulting temp_{mrt} is tested to see if the requirement, $\Pr(\text{temp}_{mrt}(G) > c) \leq \delta$, is met. Line 19, then decreases the retiming value of any vertex v that violates the requirement unless the vertex has previously been retimed in a current iteration. The algorithm then repeats the above process using the retimed graph obtained from the previous iteration. If the algorithm finds the solution for a given clock period, the final retimed graph implies the number of required resources to achieve such a schedule length.

Algorithm 2 Probabilistic retiming
Require probabilistic graph, a requirement $\Pr(\text{temp}_{mrt}(G) > c) \leq \delta$
Ensure retiming function r for each node to meet the requirement

1. $\forall$ node $v \in V$ initialize retiming function $r(v)$ to 0
2. **for** i = 1 to $|V|$ **do**
3. retime graph G_r with the retiming function $r(v)$
4. G_0 = directed acyclic portion (DAG) of G_r
5. prepend dummy node v_s to G_0 {connects to all root nodes}
6. append dummy node v_d to G_0 {connected by all leaf nodes}
7. **for all** nodes in G_0 **do**
8. $\text{temp}_{mrt}(v_s, u) = 0$
9. insert v_s into *Queue*
10. $T_{v_s} = T_{v_d} = 0$ {set timing of two dummies to zero}
11. **end for**
12. **while** *Queue* $\neq 0$
13. get node u from the *Queue*
14. $\text{temp}_{mrt}(v_s, u) = \text{temp}_{mrt}(v_s, u) + T_u$ {adding two random variables}
15. **for all** $u \xrightarrow{e} v \in G_0$ **do**
16. decrement number of incoming degrees of node v by one
17. $\text{temp}_{mrt}(v_s, v) = \max(\text{temp}_{mrt}(v_s, u), \text{temp}_{mrt}(v_s, v))$ {maximizing two random variables}
18. **if** $\Pr(\text{temp}_{mrt}(v_s, v) > c) > \delta$ **and** u has not been retimed **then**
19. $r(u) = r(u) - 1$ {move one dependency distances from all outgoing edges to all incoming edges}
20. **end if**
21. **if** number of incoming edges of node v is 0 **then**
22. insert node v into a ready Queue
23. **end if**
24. **end for**
25. **end while**
26. **end for**

Line 19 pushes a dependency distance onto all incoming edges of a node that violates the timing constraint. Since all descendents of this node will also be retimed, Line 19 in essence moves a dependency distance from below v_d to above this node. In other words, $r(u) = r(u) - 1$ for all nodes from u to v_d. Hence only the incoming edges of vertex u will have an additional dependency distance. Once all nodes are not retimed in the current iteration, the requirement $\Pr(\text{mrt}(v_s, v_d) > c) \leq \delta$ is met. Then the algorithm stops and reports the resulting retiming functions associated with nodes in the graph. Otherwise, the algorithm repeats at most $|V|$ times. Since the computation of the maximum reaching time is performed in every iteration, the time complexity of this algorithm is $O(c^2|V|^2|E|)$ while the space complexity remains the same as in the maximum reaching time algorithm. The resulting retiming function returned by Algorithm 2 guarantees (necessary condition) the following:

Theorem 3.1. *Given $G = \langle V, E, d, T\rangle$, a desired cycle period, c, and a confidence probability, $\theta = 1 - \delta$, if Algorithm 2 (probabilistic retiming algorithm) finds a solution, then the resulting retimed graph G_r satisfies the requirement* $\Pr(\text{mrt}(G) \leq c) \geq \theta$.

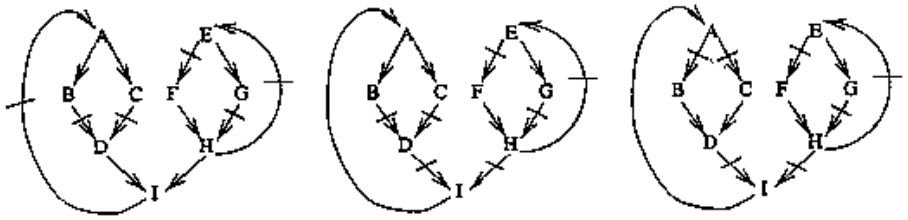


(a)

Time	Nodes								
	A	B	C	D	E	F	G	H	I
1	0.3	0	0	0.9	0	0.5	0.9	0.5	0
2	0.7	0.8	0	0.1	0.5	0	0	0	0.5
3	0	0	0.5	0	0	0.5	0	0.5	0.5
4	0	0	0.5	0	0.5	0	0.1	0	0
5	0	0.2	0	0	0	0	0	0	0

(b)

Figure 7. Illustrates an example of a 9-node graph and its corresponding probabilistic timing information.



(a) (b) (c)

Figure 8. Illustrates retimed graph corresponding to Table 1, 2, and 3.

Table 1. Shows first iteration showing probability distribution of mrt(v_s, v), $v \in V$

$v \in V$	Pr(mrt(v_s, v)) for different time							$r(v)$
	1	2	3	4	5	6	> c	
A	0.3	0.7	0	0	0	0	0	0
B	0	0	0.24	0.56	0	0.06	0.14	0
C	0	0	0	0.15	0.5	0.35	0	0
D	0	0	0	0	0.108	0.372	0.520	-1
E	0	0.5	0	0.5	0	0	0	0
F	0	0	0.250	0	0.5	0	0.25	-1
G	0	0	0.45	0	0.45	0.05	0.050	0
H	0	0	0	0.056	0	0.338	0.606	-1
I	0	0	0	0	0	0	1	-1
v_d	0	0	0	0	0	0	1	0

Table 2. Shows second iteration showing probability distribution of mrt(v_s, v), $v \in V$

$v \in V$	Pr(mrt(v_s, v)) for different time							$r(v)$
	1	2	3	4	5	6	> c	
A	0.3	0.7	0	0	0	0	0	0
B	0	0	0.24	0.56	0	0.06	0.14	0
C	0	0	0	0.15	0.5	0.35	0	0
D	0.9	0.1	0	0	0	0	0	-1
E	0	0.5	0	0.5	0	0	0	0
F	0.5	0	0.5	0	0	0	0	-1
G	0.9	0	0	0.1	0	0	0	0
H	0	0.225	0	0.45	0.05	0.225	0.5	-1
I	0	0	0	0.112	0.112	0.225	0.550	-2
v_d	0	0	0	0.013	0.103	0.27	0.613	0

Table 3. Shows third iteration showing probability distribution of mrt(v_s, v), $v \in V$

$v \in V$	Pr(mrt(v_s, v)) for different time							$r(v)$
	1	2	3	4	5	6	> c	
A	0	0	0.15	0.5	0.35	0	0	0
B	0	0	0	0	0.12	0.4	0.48	-1
C	0	0	0	0	0	0.075	0.925	-1
D	0.9	0.1	0	0	0	0	0	-1
E	0	0.5	0	0.5	0	0	0	0
F	0.5	0	0.5	0	0	0	0	-1
G	0.9	0	0	0.1	0	0	0	0
H	0	0.225	0	0.45	0.05	0.225	0.05	-1
I	0	0.5	0.5	0	0	0	0	-2
v_d	0	0	0	0	0	0.037	0.963	0

3.3 Example

Consider the PG and the probability distribution associated with nodes in the graph in Fig.7. For this experiment, let c = 6 be the desired longest path computation time and δ = 0.2 be the acceptable probability. Algorithm 2 works by first checking and computing mrt from v_s to A and E. Then, it topologically calculates the mrt of the adjacent nodes of A and E. After it computes the mrt of node I, mrt(v_s, v_d) is obtained.

Three iterations of Algorithm 2 computing the results of the maximum reaching time from v_s to v including v_d are tabulated in Tables 1, 2 and 3. After the first iteration, the retiming value associated with nodes D, F, H, and I are shown in Column $r(v)$ of Table 1. The values in Columns 2-8 show the probability that the mrt(v_s,v), $\forall v \in V$, ranges from 1 to 6 and greater than 6 (>6), respectively. The retimed graph associated with the retiming value in Table 1 after the first iteration is presented in Fig. 8a. Table 2 presents the maximum reaching time from the dummy node v_s to each node $v \in V$ as well as the retiming function for each vertex after the second iteration. Fig. 8b presents the retimed graph corresponding to the retiming function presented in Table 2. By computing the mrt(v_s,v) of the retimed graph in Fig. 8b, it becomes apparent that nodes B and C need to be retimed. Fig. 8c illustrates the final retimed graph in accordance with the retiming function presented in Table 3. Note that Table 3 also presents the final maximum reaching time and retiming value for each vertex which satisfies the required configuration. From this final retimed graph, one could, therefore, allocate a minimum of five processing elements in order to compute the graph in six time units with 80 percent confidence.

4. Resource-Constrained Scheduling

In this section, we present a probabilistic scheduling algorithm which considers the limited number of resources. The traditional rotation scheduling framework is extended to handle the probabilistic environment. We call this algorithm *probabilistic rotation scheduling* (PRS). Given a PG, the algorithm iteratively optimizes the PG with respect to the confidence probability and the number of resources.

Before presenting this algorithm, we first discuss two important concepts that make scheduling under the probabilistic environment different from traditional scheduling problems. First, in the probabilistic model, a synchronization control step is not available. A node can begin its execution if all of its parents have already been

executed. This is similar to the asynchronous model where data request and handshaking signals are used to communicate between nodes. The schedule can be viewed as a directed graph where edges show either the data requirement to execute a node or the order that a node can be executed in a particular functional unit. Note that a synchronization will be applied at the end of each iteration. Second, the task remapping strategy for PRS should take the probabilistic nature of the problem into account. The following subsection discuss these concepts in more details.

4.1 Schedule Length Subject to the Confidence

The concept of mrt can be used to compute the underlying schedule length. Hence, the conventional way of calculating schedule length has to be redefined to include the mrt notion. In order to do so, we update the probabilistic data flow graph by adding the resource information and extra edges between two nodes executed consecutively in the same functional unit and have no data dependencies between them. This graph, called the *probabilistic task-assignment graph* (PTG), represents a schedule under the probabilistic model.

Definition 4.1. *A probabilistic task-assignment graph* (*PTG*) $G = \langle V, E, w, T, b\rangle$, *is a vertex-weighted, edge-weighted, directed acyclic graph, where* V *is the set of vertices representing tasks,* E *is the set of edges representing the data dependencies between vertices,* w *is a edge-type function from* $e \in E$ *to* $\{0,1\}$, *where* 0 *represents the type of dependency edge and* 1 *represents the type of flow-control edge,* T_v *is a random variable representing the computation time of a node* $v \in V$, *and* b *is a processor binding function from* $v \in V$ *to* $\{PE_i, 1 \le i \le n\}$, *where* PE_i *is processing element* i *and* n *is the total number of processing elements*.

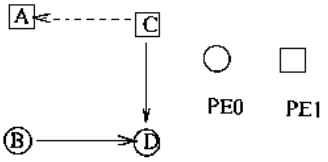


Figure 9. Illustrates an example of a probabilistic task-assignment graph (PTG) where the nodes are assigned to PE0 and PE_1.

As an example, Fig. 9 shows an example of the PTG with two functional units PE_0 and PE_1. Nodes B and D are assigned to PE_0. That is, $b(B_0 = b(D) = PE_0$. Meanwhile, $b(C) = b(A) = PE_1$. Edges consists of $C \xrightarrow{e_1} A$, $C \xrightarrow{e_2} D$, $B \xrightarrow{e_3} D$, where $w(e_1) = 1$ and $w(e_2) = w(e_3) = 0$. Note here that if there exist edges $A \rightarrow B$ and $B \rightarrow D$ and all of the nodes are scheduled to the same processor, edge $A \rightarrow D$, which was a true dependence edge, can be ignored. Note also that removing redundancy edges is simple and should be utilized to speed up the calculation of mrt. In Fig. 9, edge $C \xrightarrow{e_1} A$ is control-typed since A now has no dependency to C but has to execute after C due to resource constraints. Other edges represent data dependencies. Applying the mrt algorithm to the PTG, we can define the *probabilistic schedule length*. This length is expressed in terms of confidence probabilistic as follows.

Definition 4.2. *A probabilistic schedule length of PTG* $G = \langle V, E, w, T, b\rangle$ *with respect to a confidence level* θ, $\mathrm{psl}(G, \theta)$, *is the smallest computation time* c *such that* $\Pr(\mathrm{mrt}(G) > c) < 1 - \theta$.

For example, consider the probability distribution of the $\mathrm{mrt}(G)$ shown in table 4. Given a confidence probability $\theta = 0.8$, the probabilistic schedule length $\mathrm{psl}(G, 0.8)$ is 14. This because the smallest possible computation time is 14, where $\Pr(\mathrm{mrt}(G) > 14) < 0.2$, i.e., $0.04365 + 0.02293 + 0.00875 = 0.07818 < 0.2$. Therefore, with 80 percent confidence, the computation time of G is less than 14.

4.2 Task Remapping Heuristic: Template Scheduling

In this section, we propose a heuristic, called *template scheduling* (TS), to search for a place to reschedule a task. This remapping phase plays an important role in reducing the probabilistic schedule length in PRS. Since the computation time is a random variable, there is no fixed control step within an iteration. As long as a node is placed after its parents, any scheduling location is legal.

In template scheduling, a *schedule template* is computed using the expectation of the computation time of each node. This template implies not only the execution order, but also the expected control step that a node can start execution. In order to determine an expected control step, each node in a PTG is visited in the topological order and the following is computed:

Definition 4.3. *The expected control step of node* v *of PTG* $G = \langle V, E, w, T, b\rangle$, $\mathrm{Ecs}(v)$, *is computed by*

$$\mathrm{Ecs}(v) = \max(\mathrm{Ecs}(u_i) + ET_{u_i}),$$

where $u_i \xrightarrow{e} v \in E$, ET_u *represents the expected computation time of node* u *and* $\mathrm{Ecs}(v_i) = 0$ *for all root nodes* $v_i \in V$.

This definition assumes node v can start its execution right after all parents finish their execution. By observing this template, one can ascertain how long (the number of control steps) each processing element would be idle. The template scheduling decides where to reschedule a node using "their degree of flexibility."

Table 4. Shows possible computation time of the mrt(G)

	8	9	10	11	12	13	14	15	16
Prob	0.00197	0.04373	0.20902	0.25140	0.23661	0.18194	0.04365	0.02293	0.00875

Definition 4. *Given a PTG $G = \langle V, E, w, T, b\rangle$, a degree of flexibility of node u with respect to the processing element PE_i,* dflex(u, i), *is computed by*:

$$\text{dflex}(u, i) = \text{Ecs}(v) - \text{Ecs}(u) - ET_u,$$

where $u \xrightarrow{e} v \in E$ *and u and v are assigned to* PE_i.

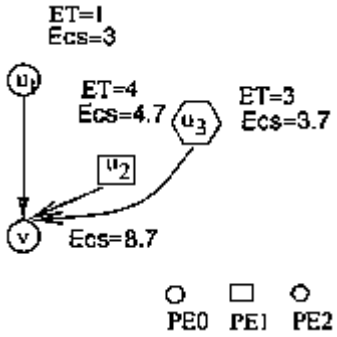


Figure 10. Illustrates an example of how to obtain the expected control step.

The degree of flexibility conveys the expected size of available time slot within PE_i. Fig. 10 shows a typical case where node v has more than one parent. u_1, u_2 and u_3 are parents of node v and each of these parents has the expected computation time 1, 4, and 3, respectively. In the same order, the expected control steps of these nodes are 3, 4.7, and 3.7, respectively. Therefore, the expected control step Ecs(v) = 8.7. According to Definition 4.3, the degree of flexibility of u with respect to PE_0, is 8.7 – 3 – 1 = 4.7. This value conveys how long PE_0 has to wait before v can be executed. Note that the degree of flexibility of a node, which is executed at last in any PE, is undefined. The following steps compute the new G after rescheduling node v.

Algorithm 3 Rescheduling rotated nodes using the template scheduling heuristic
Require: PTG, rescheduled node v and confidence probability θ
Ensure: Rescheduling new PTG with shortest psl

1. Assume that all nodes in the PTG have their expected computation times precomputed
2. $\forall$ node $u \in V$ compute Ecs(u) and dflex(u)
3. **for** each of target processors (PE_i) **do**
4. Using the maximum dflex to select node x which is scheduled to PE_i
5. schedule v after x
6. reconstruct a new PTG (assigned with PE_i) with this assignment
7. compare it with others PTG and get the one that has the best psl
8. **end for**

This rescheduling policy hopes that placing a node in the processor with the expected biggest idle time slot results in the least potential of increasing the total execution time. If a computation time of the node is much smaller than the expected time slot, this approach may allow the next rescheduled node to be placed here also. This is similar to worst-fit policy, where the scheduler strives to schedule a node to biggest slot. In section 5, we demonstrate the effectiveness of this heuristic over the method that exhaustively finds the best place for a node. Note that this exhaustive search is not performed globally, rather the search is done locally in each remapping iteration. We call this heuristic a *local search* (LS).

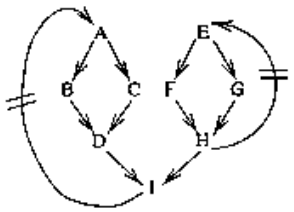
4.3 Rotation Phase

Having discussed the rescheduling heuristic, the following presents the probabilistic rotation scheduling (PRS). Note that the previous heuristic or any rescheduling heuristic can be used as rescheduling part of this PRS algorithm. The experiments in Section 5 show the efficacy of the PRS framework with different rescheduling heuristics.

Algorithm 4 Probabilistic Rotation Scheduling
Require: PG and designer's confidence probability θ
Ensure: PTG with a shortest psl

1. $\forall u \in V$ compute ET_u
2. $G_s \Leftarrow$ find_initial_schedule {finding an initial schedule for PG and keep it in G_s}
3. **for** i = 1 to 2$|V|$ **do**
4. $R \Leftarrow$ all roots of a DAG portion of G_s {these are nodes to be rotated}
5. retime each of the nodes in R
6. reschedule these nodes one by one using the heuristic previously presented
7. compute psl of the new graph with respect to θ
8. **if** psl(G_s, θ) ≤ psl (G_{best}, θ) **then**
9. $G_{\text{best}} \Leftarrow G_s$ {considering that G_{best} is initialized to G_s first}
10. **end if**
11. **end for**

In order to use template scheduling, an expected computation time of each task will be precomputed. After that, an initial schedule is constructed by find_initial_schedule. Note that the algorithm for creating the initial schedule can be any DAG scheduling, e.g., probabilistic list scheduling discussed previously. Rotation scheduling loops for 2$|V|$ times to reschedule all nodes in the graph at least once. Like traditional rotation scheduling, only nodes that have all their incoming edges with nonzero dependency distances will be drawn from each of these edges and placed on their outgoing edges. Then, these rotated nodes will be rescheduled one by one using the template scheduling technique. After all rotated nodes are scheduled, if the resulting PTG is better than the current one, Algorithm 4 will save the better PTG.

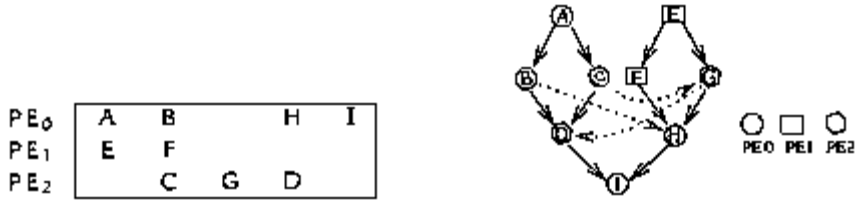


(a)

Time	Nodes								
	A	B	C	D	E	F	G	H	I
1	0.3	0	0	0.9	0	0.5	0.9	0.5	0
2	0.7	0.8	0	0.1	0.5	0	0	0	0.5
3	0	0	0.5	0	0	0.5	0	0.5	0.5
4	0	0	0.5	0	0.5	0	0.1	0	0
5	0	0.2	0	0	0	0	0	0	0

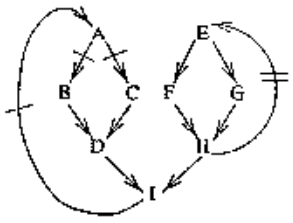
(b)

Figure 11. Illustrates an example of the computation time of graph in Figure 1b.



(a) (b)

Figure 12. Illustrates initial assignment and the corresponding execution order. (a) Static execution order. (b) PTG.



(a)

	A	B	C	D	E	F	G	H	I
ET_v	1.7	2.6	3.5	1.1	3.0	2.0	1.3	2.0	2.5
Ecs	-	1.7	1.7	6.5	0.0	3.0	5.2	6.5	8.5
dflex	0.0	2.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0

(b)

Figure 13. Illustrates the probabilistic graph after A is rotated and the template values. (a) New PTG. (b) Ecs and dflex.

Table 5. Shows possible computation time of the mrt of a PTG

	8	9	10	11	12	13	14	15	16
Prob	0.00197	0.04373	0.20902	0.25140	0.23661	0.18194	0.04365	0.02293	0.00875

4.4 Example

Let us revisited the PG example in Section 3.3 as shown in Fig. 11a and the corresponding computation time in Fig. 11b. The confidence probability is given as $\theta = 0.8$. After list scheduling is applied, the initial execution order is constructed as shown in Fig. 12a. The corresponding PTG is presented in Fig. 12b. Nodes *A*, *B*, *H* and *I* are assigned to PE_0, nodes *E* and *F* are scheduled PE_1 and nodes *C*, *G* and *D* are assigned to PE_2. Edges $B \xrightarrow{e} H$ and $C \xrightarrow{e} G$ and $G \xrightarrow{e} D$ are flow-control edges.

For this assignment, the mrt of such a PTG is computed as shown in Table 5. Therefore, with higher than 80 percent confidence probability, psl(*G*, 0.8) = 14.

According to the structure of the PTG, either *A* or *E* can be rescheduled. In the first rotation, PRS selects *A* to be rescheduled. One dependency distance is moved from all incoming edges of *A* and pushed to all outgoing edges of *A*. The resulting retimed graph PG is shown in Fig. 13a. In this graph, node *A* requires no direct data dependency from any node. Therefore, *A* can be placed at any position in the schedule. Fig. 13b shows the expected computation time, the expected control step, and the degree of flexibility of each node in this PTG.

Based on the values in the table from Fig. 13b, it is obvious that an expected waiting time between *B* and *H* in PE_0 where psl can be reduced. The resulting PTG and its execution order are shown in Fig. 14, where psl(*G*, 0.8) = 12. After running PRS for 18 iterations, the shortest possible schedule length was found in the 15th iteration. In Fig. 15, we present the resulting schedule length of the this trial which is less than 9 with probability greater than 80 percent (psl(*G*, 0.8) = 9).

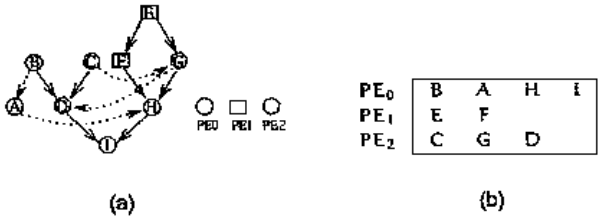


Figure 14. Illustrates the PTG, execution order and its mrt after the first rotation where psl (G, 0.8) = 12. (a) PTG. (b) Execution order.

5. Experimental Results

In this section we perform experiments both using nonresource and resource constrained scheduling on two general classes of problems. The first class are real applications which may have a combination of nodes with probabilistic computation times and with fixed computation times. The second are well-known DSP filter benchmarks. Since these benchmarks contain two uniform types of nodes, namely multiplication and addition, the basic timing information consisting of three probability distribution are assigned to each benchmark graphs. In order to show the usability of the proposed algorithm, three applications are profiled to get their probabilistic timing information. The profiler reports the processing time requirement in these applications and the corresponding frequency of this time value. The frequency of timing occurences is used to obtain a node probability distributions. A node in these graphs may represent a large number of operations which cause the uncertain computation time as well as operations which have fixed timing information. Each timing information is discretized to a smaller unit such as nanoseconds.

The DSP filter benchmarks used in these experiments include a Biquadratic IIR filter, a 3-stage IIR filter, a fourth-order Jaunmann wave digital filter, a fifth-order elliptic filter, an unfolded fifth-order elliptic filter with an unfolding factor equal to 4 (uf = 4), an all-pole lattice filter, an unfolded all-pole lattice filter (uf = 2), an unfolded all-pole lattice filter (uf = 6), a differential equation solver and a Volterra filter. The rest of the benchmarks are the application for image processing (Floyd-Steinberg), the application to search for a solution which maximize some unknown function by using genetic algorithm, and the famous example of the application in the fuzzy logic area, the inverted pendulum problem. All of the experiments were performed using SUN UltraSparc.

Table 6. Shows probabilistic retiming versus worst case traditional retimimg

Benchmark	#nodes	c worst	θ = 0.9		θ = 0.8	
			c	%	c	%
Biquad IIR	8	78	60	23	57	26
Diff. Equation	11	118	81	31	77	35
3-stage direct IIR	12	54	44	19	41	24
All-poll Lattice	15	157	120	24	117	25
4^{th} order WDF	17	156	116	26	112	28
Volterra	27	276	216	22	212	23
5^{th} Elliptic	34	330	240	28	236	29
All-poll Lattice (uf=2)	45	468	350	25	346	25
All-poll Lattice (uf=6)	105	1092	811	26	806	26
5^{th} Elliptic (uf=4)	170	1633	1185	27	1174	28
Floyd-Steinberg	13	30	23	23	21	30
Genetic application	18	202	180	11	127	37
Fuzzy application	24	19	17	11	17	11

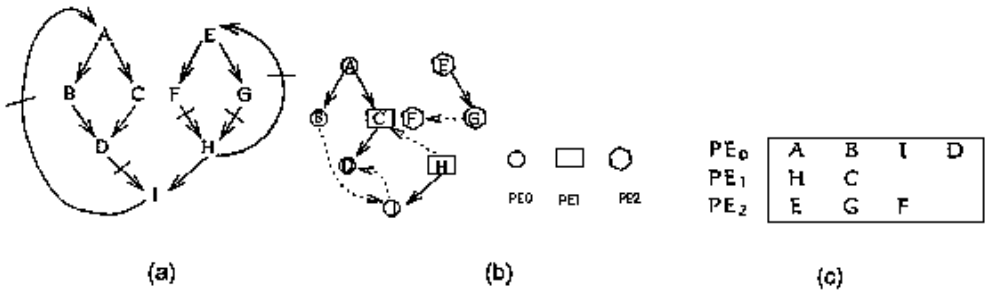


Figure 15. Illustrates the final PG, PTG, execution order where psl(G, 0.8) = 9. (a) PG. (b) PTG. (c) Final execution order.

Table 7. Shows probabilistic retiming versus average-case analysis

Benchmark	Traditional	Algorithm 2	
	mrt(G^r_{avg})	θ = 0.9	θ = 0.8
Biquad IIR	70.40	52.64	52.30
Diff. Equation	76.05	73.07	72.50
3-stage direct IIR	41.90	37.70	38.36
All-poll Lattice	114.45	111.77	111.40
4th order WDF	106.73	106.44	105.98
Volterra	204.00	202.44	202.00
5th Elliptic	233.30	228.41	227.59
All-poll Lattice (uf=2)	342.17	338.11	337.62
All-poll Lattice (uf=6)	800.51	794.02	793.39
5th Elliptic (uf=4)	800.51	794.02	793.39
Floyd-Steinberg	18.47	18.45	18.17
Genetic application	150.89	144.01	112.46
Fuzzy application	18.03	16.08	16.08

5.1 Nonresource Constrained Experiments

In each experiment, for a given confidence level $\theta = 1 - \delta$, Algorithm 2 is used to search for the best longest path computation time. In order to do this, the current desired longest path computation time (c) is varied based on whether or not a feasible solution is found. For instance, if c is too small, the algorithm will report that no feasible solution exists. In this case, c is increased and Algorithm 2 is reapplied. This process will repeated, until the smallest feasible c is found.

Table 6 shows the results for traditional retiming using worst-case computation time assumptions (column c worst) and the probabilistic model with two high confidence probabilities (θ = 0.9 and 0.8). The average running time for these experiments was determined to be below 60 seconds including the input/output interfaces. The algorithms are implemented in a straightforward way where array is used to store probability distributions. Column 3 in the table presents the optimized longest path computation times obtained from applying traditional retiming using the worst-case computation time for each node in the graph benchmarks. For columns where θ = 0.9 and θ = 0.8, the probabilistic retiming algorithm is applied to the benchmarks (G) while each of these confidence probabilities is used as its input. The numbers show in both columns are the given c where $\Pr(\text{mrt}(G) \le c) \ge \theta$. The value c from this requirement is the smallest input value which Algorithm 2 can find a solution to satisfy such a requirement. Notice that for all benchmarks the longest path computation time with $\theta = 0.9$ are still smaller than the computation time in Column 3. In order to quantify the improvement of the probabilistic retiming algorithm, the "%" columns list the percent of computation time reductions with respect to the value from Column 3.

Table 7 compares the probabilistic retiming algorithm to the traditional retiming algorithm with average computation times used for each node in the graphs. First, the probabilistic nodes of each input graph are converted to fixed time nodes resulting in G_{avg}, i.e., each node assumes its average computation time rather than probabilistic computation time. Traditional retiming is then applied to the resulting graph, resulting in graph G^r_{avg}. The purpose of this table is to compare G^r_{avg} (obtained from running traditional retiming on G^r_{avg}) with retimed PGs. In order to compare with the results produced by the proposed algorithm, the placement of dependency distance in each G^r_{avg} is preserved while the original probabilistic computation times are replaced with the average computation times. Put another way, we transformed each G^r_{avg} back to a probabilistic graph. Algorithm 1 is then used to evaluate these graphs while only the *expected values* of each result are shown in the table. Columns 4 and 5 present the expected values of the results obtained from running probabilistic retiming on each PG where the confidence probability of 0.9 and 0.8 are considered. Note that these results are consistently better (smaller value) than the results obtained from running traditional retiming on each of G^r_{avg}. Hence, the approach of using the expected values for each node is neither a good heuristic in the initial design phase nor does it give any quantitative confidence to the resulting graphs.

5.2 Resource-Constrained Experiments

We tested the probabilistic rotation scheduling (PRS) algorithm on the selected filter and application benchmarks: the fifth elliptic filter, 3 stage-IIR filter, Volterra filter, Lattice filter, and Floyd-Steinberg, Genetic algorithm, Fuzzy logic applications. Table 8 demonstrates the effectiveness of our approach on both 2-adder, 1-multiplier and 2-adder, 2-multiplier systems for those filter benchmarks. The specification of 3 and 4 general purpose processors (PEs) are adopted for the other three application benchmarks. The performance of PRS is evaluated by comparing the resulting schedule length with the schedule length obtained from the modified list scheduling technique (capable of handling the probabilistic graphs). We also show the effectiveness of template scheduling (TS) by comparing its results with other heuristics, namely, local search (LS), and as-late-as possible scheduling (AL). The average execution times of AL and TS are very comparable (about 12 seconds running on UltraSparc) while LS takes much longer time and does not give the outstanding results comparing with those from TS.

In each rescheduling phase of PRS, the LS approach strives to reschedule a node to all possible legal location (local search) and returns the assignment which yields the minimum psl(G, θ). This method is simple and gives a good schedule; however, it is time consuming and not practical to try all possible scheduling places in every iteration of PRS. Furthermore, a PTG needs to be temporarily updated in every trial in order to compute the possible schedule length. On the contrary, the AL method reduces the number of trials by attempting to schedule a task only once at the farthest legal position in each functional unit or processor while the TS heuristic re-maps the scheduled node after the node with the highest degree of flexibility in each functional unit.

Table 8. Shows comparison of the results obtained from applying benchmarks modified list and probabilistic rotation scheduling (using different remapping heuristics)

Spec.	Benchmarks	#nodes	θ = 0.9				θ = 0.8			
			PL	PRS			PL	PRS		
				AL	LS	TS		AL	LS	TS
2 Adds. 1 Mul.	Diff. Equation	11	169	152	133	**133**	165	147	131	**131**
	3-stage direct IIR	12	188	184	151	**151**	184	179	147	**147**
	All-poll Lattice	15	229	225	142	**141**	225	220	138	**138**
	Volterra	27	526	468	361	**361**	519	461	354	**354**
	5^{th} Elliptic	34	318	298	293	**293**	314	294	289	**289**
3 PEs	Floyd-Steinberg	13	42	32	**27**	30	38	28	28	**27**
	Genetic application	18	434	295	**216**	275	438	180	150	**150**
	Fuzzy application	24	52	46	45	**45**	52	45	43	**43**
2 Adds. 2 Mul.	Diff. Equation	11	120	103	**83**	90	117	100	**83**	91
	3-stage direct IIR	12	124	120	87	**87**	120	110	83	**82**
	All-poll Lattice	15	229	225	140	**139**	225	220	136	**136**
	Volterra	27	359	270	**237**	259	353	265	**221**	256
	5^{th} Elliptic	34	288	288	274	**271**	284	274	270	**267**
4 PEs	Floyd-Steinberg	13	42	30	**26**	28	38	24	24	**24**
	Genetic application	18	434	291	**205**	275	337	180	180	**180**
	Fuzzy application	24	49	41	**38**	40	47	38	**35**	36

Table 9. Shows comparing probabilistic rotation with traditional rotation running on graphs with average computation times

Spec.	Benchmarks	#nodes	worst case		θ = 0.9			θ = 0.8		
			L	R	PL	PRS	AVG	PL	PRS	AVG
2 Adds. 1 Mul.	Diff. Equation	11	228	180	169	**133**	136	165	**131**	131
	3-stage direct IIR	12	252	204	188	**151**	163	184	**147**	179
	All-poll Lattice	15	312	204	229	**141**	153	225	**138**	149
	Volterra	27	750	510	526	**361**	526	519	**354**	519
	5^{th} Elliptic	34	438	396	318	**293**	299	314	**289**	294
3 PEs	Floyd-Steinberg	13	59	38	42	**30**	40	38	**27**	31
	Genetic application	18	500	400	434	**275**	416	438	**180**	410
	Fuzzy application	24	69	55	52	**45**	66	52	**43**	63

Columns θ = 0.8 and θ = 0.9 show the results when considering the probabilistic situations with confidence probabilities 0.8 and 0.9. Column "PL" presents the probabilistic schedule length (psl) after modified list scheduling is applied to the benchmarks. Columns "LS", "AL", and "TS" show the resulting psl, after running PRS against those benchmarks using the remapping heuristics LS, AL and TS respectively. Among these three heuristics, the TS scheme produces better results than AL which uses the simplest criteria. Further, it yields as good as or sometimes even better results than given by the LS approach, while TS taking less time to select a re-scheduled position for a node. This is because in each iteration the LS method finds the local optimal place. However, scheduling nodes to these positions does not always result in the global optimal schedule length.

In Table 9, based on the system that has 2 adders and 1 multiplier (for filter benchmarks) and 3PEs (for application benchmarks), we present the comparison results obtained from applying the following techniques to the benchmarks: modified list scheduling, traditional rotation scheduling, probabilistic rotation scheduling using template scheduling heuristic, and traditional rotation scheduling considering average computation times. Columns "L" and "R" show the schedule length obtained from applying modified list scheduling and traditional rotation scheduling respectively to the benchmarks where all probabilistic computation times are converted into their worst-case computation times.

Obviously, considering the probabilistic case gives the significant improvement of the schedule length over the worst case scenario.

Column "PL" presents the initial schedule lengths obtained from using the modified list scheduling approach. The results in column "PRS" are obtained from Table 8 (PRS using template scheduling heuristic). In column "AVG", the psls are computed by using the graphs (PTGs) retrieved from running traditional rotation to the benchmarks where the average computation time is assigned to each node. These results demonstrate that considering the probabilistic situation while performing rotation scheduling can consistently give better schedules than considering only worst-case or average-case computation times.

6. Conclusion

We have presented scheduling and optimization algorithms which operate in probabilistic environment. A probabilistic data-flow graph is used to model an application which takes this probabilistic nature into account. The probabilistic retiming algorithm is used to optimize the given application when nonresource constrained environments are assumed. Given an acceptable probability and a desired longest path computation time, the algorithm reduces the computation time of the given probabilistic graph to the desired value. The concept of maximum reaching time is used to calculate timing values of the probabilistic graph. When a limited number of processing elements is considered, the probabilistic rotation scheduling algorithm (where the probabilistic concept and loop pipelining are integrated to optimize a task schedule) is proposed. Based on the maximum reaching time notion, the probabilistic schedule length is used to measure the total computation time of these tasks being scheduled in one iteration. Given a probabilistic graph, the schedule is constructed by using the task-assignment probabilistic graph and the probabilistic schedule length is computed with respect to a given confidence probability θ. Probabilistic rotation scheduling is applied to the initial schedule in order to optimize the schedule. It produces the best optimized schedule with respect to the confidence probability. The remapping heuristic, template scheduling, is incorporated in the algorithm in order to find the scheduling position for each node.

Acknowledgment

The research was partially supported by U.S. National Science Foundation Career Grant MIP 95-01006.

References

[1] A. Aiken and A. Nicolau, "Development Environment for Horizontal Micronode," *IEEE Trans. Software Eng*., vol. 14, Feb. 1987.

[2] A. Aiken and A. Nicolau, "Optimal Loop Parallelization," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 308-317, June 1988.

[3] U. Banerjee, "Unimodular Transformations of Double Loops," *Proc. Workshop Advances in Languages and Compilers for Parallel Processing*, pp. 192-219, Aug 1990.

[4] P.P. Chang et al., "Impact: An Architectural Framework for Multiple Instruction Issue Processor," *Proc. 18th Int'l Symp. Computer Architecture*, pp. 266-275, 1991.

[5] L. Chao, A. LaPaugh, and E. Sha, "Rotation Scheduling: A Loop Pipelining Algorithm," *Proc. 30th Design Automation Conf*., pp. 566-572, June 1993.

[6] L. Chao and E. Sha, "Static Scheduling for Synthesis of DSP Algorithms on Various Models," *J. VLSI Signal Processing*, pp. 207-223, Oct. 1995.

[7] A.E. Eichenberger and E.S. Davidson, "Stage Scheduling: A Technique to Reduce the Register Requirements of a Modulo Schedule," *Proc. 28th Int'l Symp. Microarchitecture*, pp. 338-349, Nov. 1995.

[8] A.E. Eichenberger, E.S. Davidson, and S.G. Abraham, "Minimum Register Requirements for a Modulo Schedule," *Proc. 27th Int'l Symp. Microarchitecture*, pp. 75-84, Nov. 1994.

[9] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers*, vol. 30, no. 7, pp. 478-490, July 1981.

[10] I. Foster, *Designing and Building Parallel Program: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1994.

[11] R.A. Kamin, G.B. Adams, and P.K. Dubey, "Dynamic List-Scheduling with Finite Resources," Proc. *1994 Int'l Conf. Computer Design*, pp. 140-144, Oct. 1994.

[12] I. Karkowski and R.H.J.M. Otten, "Retiming Synchronous Circuitry with Imprecise Delays," *Proc. 32nd Design Automation Conf*., pp. 322-326, 1995.

[13] A.A. Khan, C.L. McCreary, and M.S. Jones, "A Comparison of Multiprocessor Scheduling Heuristic," *Proc. 1994 Int'l Conf. Parallel Processing*, vol. II, pp. 243-250, 1994.

[14] D. Ku and G. De Micheli, *High-Level Synthesis of ASICS under Timing and Synchronization Constraints*. Kluwer Academic, 1992.

[15] D. Ku and G. De Micheli, "Relative Scheduling under Timing Constraints: Algorithm for High-Level Synthesis," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, pp. 697-718, June 1992.

[16] M. Lam, "Software Pipelining," *Proc. ACM SIGPLAN '88 Conf. Programming Language Design and Implementation*, pp. 318-328, June 1988.

[17] D.M. Lavery and W.W. Hwu, "Unrolling-Based Optimization for Modulo Scheduling," *Proc. 28th Int'l Symp. Microarchitecture*, pp. 327-337, Nov. 1995.

[18] C.E. Leiserson and J.B. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, vol. 6, pp. 5-35, 1991.

[19] B.P. Lester, *The Art of Parallel Programming*. Englewood Cliffs, N.J.: prentice Hall, 1993.

[20] W. Li and K. Pingali, "A Singular Loop Transformation Framework Based on Non-Singular Matrices," Technical report TR 92-1294, Cornell Univ., Ithaca, N.Y., July 1992.

[21] J. Llosa, M. Valero, and E. Ayguadé, "Heuristics for Register-Constrained Software Pipelining," *Proc. 29th Int'l Symp. Microarchitecture*, pp. 250-261, Dec. 1996.

[22] K.K. Parhi and D.G. Messerschmit, "Static Rate-Optimal Scheduling of Iterative Data-Flow Program via Optimum Unfolding," *IEEE Trans. Computers*, vol. 40, no. 2, pp. 178-195, Feb. 1991.

[23] N.L. Passos, E. Sha, and S.C. Bass, "Loop Pipelining for Scheduling Multi-Dimensional Systems via Rotation," *Proc. 31st Design Automation Conf.*, pp. 485-490, June 1994.

[24] B.R. Rau and J.A. Fisher, "Instruction-Level Parallel Processing," *J. Supercomputing*, vol. 7, pp. 9-50, July 1993.

[25] B.R. Rau and C.D. Glaeser, "Some Scheduling Techniques and a Easily Schedulable Horizontal Architecture for High Performance Scientific Computing," *Proc. 14th Ann. Workshop Microprogramming*, pp. 183-198, Oct. 1981.

[26] B. Ramakrishna Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," *Proc. 27th Ann. Int'l Symp. Microarchitecture*, pp. 63-74, Nov. 1994.

[27] M.E. Wolfe, *High Performance Compilers for Parallel Computing*, chapter 9, Redwood City, Calif.: Addison-Wesley, 1996.

[28] M.E. Wolfe and M.S. Lam, "A Loop Transformation Theory and Algorithm to Maximize Parallelism," *IEEE Trans. Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452-471, Oct. 1991.

[29] L.A. Zadeh, "Fuzzy Sets as a Basis for a Theory of Possibility," *Fuzzy Seta and Systems*, vol. 1, pp. 3-28, 1978.

Sissades Tongsima received the BEng degree in industrial instrumental engineering in 1991 from Kink Mongkut Institute of Technology, Ladkrabang, Thailand. In 1992, he was awarded the Royal Thai Government Scholarship to pursue his studies in computer science, majoring in the parallel and distributed computing area. He obtained his MS and PhD degrees from the Department of Computer Science and Engineering, University of Notre Dame, in 1995 and 1999, respectively. Dr. Tongsima's research interests while studying at Notre Dame include data scheduling and partitioning on parallel and distributed systems, high-level synthesis, loop transformations, rapid prototyping, and fuzzy systems. Upon completing of his studies, he returned to Thailand, where, in June 1999, he joined the National Electronics and Computer Technology Center (NECTEC). He is currently a researcher in the High Performance Computing Division at NECTEC.

Edwin H.-M. Sha (S'88-M'92) received the MA and PhD degrees from the Department of Computer Science, Princeton University, Princeton, New Jersey, in 1991 and 1992, respectively. Since August 1992, he has been with the University of Notre Dame, Notre Dame, Indiana. He is now an associate professor and the associate chairman of the Department of Computer Science and Engineering. His research areas include multimedia, memory systems, parallel and pipelined architectures, loop transformations and parallelizations, software tools for parallel and distributed systems, high-level synthesis in VLSI, fault-tolerant computing, VLSI processor arrays, and hardware and software co-design.

He had published more than 100 research papers in referred conferences and journals during the past seven years. In 1994, he was the program committee chair for the Fourth IEEE Great lakes Symposium on VLSI and he served as the program committee cochair for the 2000 ISCA 13th International Conference on Parallel and Distributed Computing Systems. He has served on program committees for many international conferences. He has served as an associate editor for several journals. He received the Oak Ridge Association Foundation CAREER Award in 1995. He was also invited to be a guest editor for special issue on Low Power Design of the *IEEE Transactions on VLSI Systems* and is now serving as an editor for the *IEEE Transactions on Signal Processing*. He received the CSE Undergraduate Teaching award in 1998. He is a member of the IEEE.

Chantana Chantrapornchai received her PhD degree in computer science and engineering from the University of Notre Dame in 1999. She received her bachelor's degree in computer science from Thammasart University, Bangkok, Thailand, in 1992 and her MS degree in computer science from Northeastern University, Boston, in 1994. Currently Dr. Chantrapornchai is a faculty member in the Department of Mathematics, Faculty of Science, Silpakorn University, Thailand. Her research interests include high-level synthesis, rapid prototyping, and fuzzy systems.

David R. Surma received the BS degree in electrical engineering from Valparaiso University, Valparaiso, Indiana, where he was a Presidential Scholar, in 1985. He received MS degree in electrical engineering with a specialty in computer engineering from the University of Arizona, Tucson, in 1989. In 1998, he was awarded the PhD degree in computer science and engineering from the University of Notre Dame, Notre Dame, Indiana. From 1990-1996, he taught electrical and computer engineering at Valparaiso University. Since then he has taught computer science at Indiana University and has held a visiting assistant professorship at Notre Dame. Currently, He is an assistant professor of computer science at Valparaiso University. His research interests include parallel and distributed systems, communication and data scheduling, computer networks and real-time systems. He is a member of the IEEE.

Nelson Luiz Passos received the BS degree in electrical engineering, with specialization in telecommunications, from the University of Sao Paulo, Brazil, in 1974. He received the Ms degree in computer science from the University of North Dagota, Grand Forks, in 1992, and the PhD degree in computer science and engineering from the University of Notre Dame, Indiana, in 1996.

From 1974 to 1990, he worked at Control Data Corporation. Since1996, he has been at Midwestern State University, Wichita Falls, Texas, where he is currently an associate professor with the Department of Computer Science. His research interests include parallel processing, VLSI design, loop transformations.

Dr. Passos was awarded the Professional Excellence and Bill Norris Shark Club Awards at Control Data Corporation. While with the Fellowship. He received a U.S. National Science Foundation grant award in 1997 to support to new research om multidimensional retiming. He is a member of the IEEE.