Probabilistic Loop Scheduling Considering Communication Overhead*

Sissades Tongsima, Chantana Chantrapornchai, and Edwin H.-M. Sha

University of Notre Dame, Notre Dame IN 46556, USA

Abstract. This paper presents a new methodology for statically scheduling a cyclic data-flow graph whose node computation times can be represented by random variables. A communication cost issue is also considered as another uncertain factor in which each node from the graph can produce different amount of data depending on the probability of its computation time. Since such communication costs rely on the amount of transfered data, this overhead becomes uncertain as well. We propose an algorithm to take advantage of the parallelism across a loop iteration while hiding the communication overhead. The resulting schedule will be evaluated in terms of confidence probability—the probability of having a schedule completed before a certain time. Experimental results show that the proposed framework performs better than a traditional algorithm running on an input which assumes fixed average timing information.

1 Introduction

In many practical applications such as interface systems, fuzzy systems, and 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. Traditional scheduling methods assume either the worst-case or the average-case computation time. However, the resulting schedule may be inefficient in the real operating situation. To improve the productivity, the probabilistic nature of such tasks needs to be properly considered during the scheduling phase. Furthermore, for a parallel system, which is a common platform for these computation-intensive applications, the communications between two processors should not be neglected. According to the current routing technology in parallel architectures, e.g. wormhole routing [6], an amount of data being sent among different processors constitutes a significant part of the underlying communication overhead. Therefore, such communications can directly affect the total execution time of an application. Since the amount of data created by a task may depend on varying task computation, these communication overheads can be varied. For instance, if a probabilistic

* This research was supported in part by the Royal Thai Government Scholarship and the NSF grant MIP 95-01006

task represents a group of statements from a conditional block, the computation times of this task depend on the probabilities inherited from the branching statement. This probability also implies the different amount of data produced from different branches in the conditional block. In order to realistically capture the random nature of task computation and communication times, a probabilistic model must be used. Therefore, to maximize scheduling results, scheduling techniques which properly consider such probabilistic data should be investigated.

Considerable research has been conducted in the area of scheduling of a directed-acyclic graphs (DAG) on multiple processing systems. (Note that DAGs are obtained from data-flow graphs, DFGs, by ignoring edges of a DFG containing one or more delays.) Many heuristics have been proposed to schedule DAGs, e.g., list scheduling, graph decomposition [7, 9] etc. These methods, however, do not explore the parallelism across iterations nor do they address the problems of probabilistic tasks. Loop transformations are also common techniques used to restructure loops from the repetitive code segment in order to reduce the total execution time of the schedule [1, 16, 23, 24]. These techniques, however, do not consider that the target systems have limited number of processors or that task computation times are uncertain. For global scheduling, software pipelining [13] is used to overlap instructions, whereby the parallelism is exposed across iterations. This technique, however, expands the graph by unfolding it. Furthermore, such an approach is limited to solving the problem without considering uncertain computation times for operations [5, 13].

Some research considers the uncertainty inherit in the computation time of nodes. Ku and De Micheli [11, 12] 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 [8]. In their approach, fuzzy set theory [25] was employed to model the imprecise computation time. Although their approach is equivalent to scheduling imprecise tasks to 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 proposed by Chao, LaPaugh and Sha [2, 3] and extended to handle multi-dimensional applications by Passos, Sha and Bass [19]. Rotation scheduling schedules loops by assigning nodes from the loop to the system with limited number of functional units. It implicitly explores retiming [15] 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. The idea of using loop pipelining to schedule DFGs while considering communication costs is presented in [22]. This method, called cyclo-compaction scheduling also consider the message collision scenario when wormhole routing

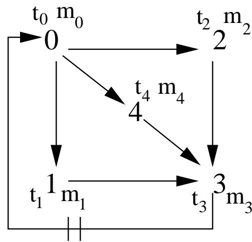
is applied. Nevertheless, this approach assumes computation time of a node to be a fixed value.

In this paper, we propose an algorithm which considers the probabilistic behavior in both computation times and communication overhead. A new graph model which is a generalized version of a regular DFG, called probabilistic extended data-flow graph (PEG) is introduced to represent an application with uncertain computation times as well as data volume. If this graph represents a repeated loop, a weighted-edge will be added between any two nodes that have inter-iteration dependencies ($A[i]$ depends on $B[i-1]$) where the weight on each edge represents the dependency distance. We assume that the data volume is initiated from a sender and will render the communication cost when both sender and receiver nodes are executed in different processors. The communication cost model is based on wormhole routing assumption where a data packet is broken down to small sub-packets, called flits, and they can be transmitted uninterruptedly from source to destination processors. When the size of data is large comparing with the distance between processors, the communication cost will be dominated by the packet length. XY-routing model is assumed in order to determine a route of the message being sent through the system. This assumption helps us determine if there exist the message collisions.

Since the timing information is uncertain, the scheduling result will have no control steps. Moreover, the resulting schedule represents only one iteration of the input graph where the rest of the iterations are assumed to be identical and the synchronization must exist at the end of each iteration. Our approach produces a static schedule for an application and this schedule is believed that if the system follows its order, the application will finish within a particular time. In other words, with some confidence probability our schedule guarantees to finish within a certain time constraint. Again, we used a DAG, called a probabilistic communication&task graph (PT&C-G) to model the scheduling order where each node can take varying computation time. Furthermore, a communication node will be added to the graph whenever data is transmitted from source to destination processors and message contentions will be handled by serializing these communication nodes.

As an example, if a target system architecture comprises of 4 processing elements connected as a 2D-mesh, a processor assignment and a possible schedule (order to execute nodes, namely a PT&C-G) can be presented as in Figures 2(a) and 2(b) respectively. Note that this schedule must satisfy the dependency constraint depicted in the PEG shown in Figure 1(a). Figure 1(b) lists possible computation time of a node associated with probability, as well as its corresponding data volume. Nodes x_i in the PT&C-G represent communication nodes which are added when their corresponding source and sink nodes are scheduled in different processors, e.g., node 3 depends on data from node 1 but both nodes are assigned to different processors. The communication node x_3 is appended to the PT&C-G since node 0 depends on this message (x_3) but gets scheduled to the different processor from its parent (node 3). By using our technique, the order in Figure 2(b) can be improved by overlapping the current iteration with node

0 from the next iterations resulting in Figures 2(c) and 2(d). According to Figure 2(c), node 0 can be executed simultaneously with nodes $1, 2$, and 4 which results in an ability to hide the communication latency x_0. In other words, this communication node can be computed while the other nodes are executing.



(a)

Node n_i	Comp. time (t_i)				Data volume (m_i)			
0	2	0.7	4	0.3	3	0.7	5	0.3
1	3	0.5	4	0.5	1	0.5	3	0.5
2	1	0.2	4	0.8	1	0.2	4	0.8
3	3	0.5	4	0.5	1	0.5	3	0.5
4	2	0.4	5	0.6	2	0.4	4	0.6

(b)

Fig. 1. Example of the PEG

If we use the traditional scheduling approach, which approximates all uncertain timing information with the node average computation times, to minimize the modified input graph, the order given by such scheduling, however, may not be a good static schedule in practice. For instance, rather than placing node 0 at PE_2 (as shown in Figure 2(c)), the traditional algorithm may assign node 0 to PE_1 since it can eliminate x_3 but when this schedule is employed in practice the performance of this system might be worse than we expected due to the timing variations. By using a simulator to simulate how a parallel computer computes tasks in a schedule, this paper shows the comparison between the results obtained from the proposed approach with the ones given by traditional scheduling [22]. The contributions of this paper can be listed as follows: First, the probabilistic model is used in this work to make a static schedule more realistic. Second, this paper also considers and minimizes the underlying communication costs which can degrade the performance of running an application on the parallel system. Third, the results obtained from our approach can tell designers with a confidence probability what to be expected in practice when applying our resulting static schedule. Finally, this paper compares the effectiveness of the proposed approach with traditional scheduling and shows the benefit of using our algo-

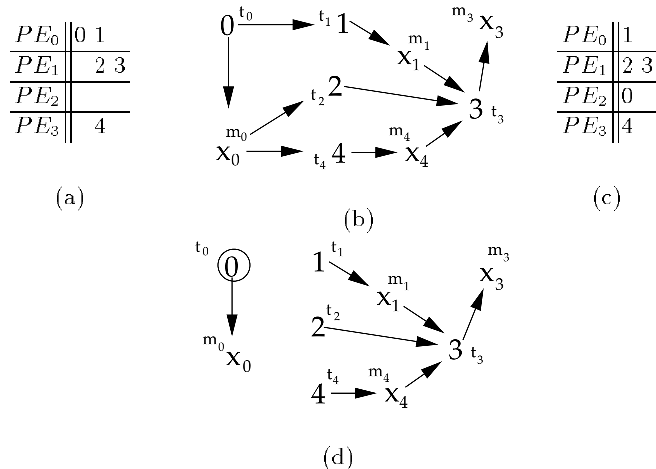


Fig. 2. Result from scheduling the graph in Figure 1

rithm by realistically simulating an execution of a resulting schedule in a parallel system.

The outline of this paper is as follows. The following section discusses some background and the graph model of this work. Then Section 3 presents how to construct a schedule as a graph and also shows an algorithm to probabilistically evaluate such a schedule. The proposed loop scheduling is also discussed in this section. Section 4 shows the experimental results from our algorithm and compare them with the results given by the traditional method. Finally, the conclusions are drawn in Section 5.

2 Background

In this paper, we generalize traditional DFGs in such a way that each node may have uncertain computation time. Furthermore, the amount of data produced by each node can be varied depending on the probability of the node computation time. The following presents the definition of this graph called a probabilistic extended data-flow graph (PEG).

Definition 1. *A probabilistic extended data-flow graph (PEG) is a node-weighted, edge-weighted, directed graph* $G = \langle V, E, \delta, m, t \rangle$ *where* V *is a set of nodes,* E *is a set of edges representing node precedence relations. The tuple* δ *is a function from* E *to positive integers representing a number of dependence distance between two nodes on the corresponding edge. The tuples* m *and* t *represent random variables of uncertain amount of data sent out from each node and random variables of uncertain computation time of each node respectively.*

As an example, Figure 1(a) depicts the PEG where all the random variables, associated with each node in the graph, are presented in Figure 1(b). Each entry

of the table presents two timing information of a node which are possible computation times and possible data volume. A probability corresponding to each timing value is presented next to its possible value in each entry. For instance, node 0 has possible computation times of 1 with probability of 0.5 and 2 with the same probability. In this example, both computation time of a node and its corresponding data volume are *dependent*. This is because a node could represent a group of operations in a conditional block. This representation causes a node to have varying computation times and varying amount of data produced at one instance. These timing and data volume, however, are related in their probabilities. For example, if node 0 takes 2 time units with 0.7 probability, then with the same probability there must also exist the amount of data produced when the node computation time is 2. In this paper, we assume that the probability distributions of both m and t are discrete, i.e., a single possible node computation time and data volume produced by this node are integer. The notation $P(rv = x)$ is read "the probability that random variable rv assumes value x".

A notation $n_u \xrightarrow{\delta} n_v$ conveys a precedence relation between nodes u (n_u) and v (n_v) where δ represents number of delays on this edge (short bar-lines in the graph). For example, the edge between nodes 3 and 0 can be shown as $n_3 \xrightarrow{2} n_0$. Note that in this graph, $n_3 \xrightarrow{2} n_0$ represents a loop carried dependence (also called delays in this paper) where two short bar lines are used to denote two dependency distance between n_3 and n_0 in the graph. In other words, for any iteration j, an edge e from n_u to n_v with delay $\delta(e)$ conveys that the computation of node v at iteration j depends on the execution of node u at iteration $j - \delta(e)$. An edge with no delays represents a data dependency within the same iteration.

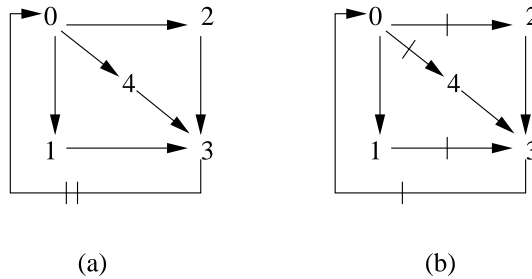
With the advent of worm-hole routing and parallel processing, we may assume that a distance between source and destination processors can be negligible when comparing with a packet length. Consider the following: given a packet length Pl (bits), a channel bandwidth Bw (bits/sec.), length of subdivided packet (flit) Fl, and number of processors D which are traversed when sending the packet from source to destination. The communication latency for transferring data from source processor to destination processor can be summarized as $T_{wh} = \frac{Pl}{Bw} + \frac{Fl}{Bw} \times D$. If Pl is much greater than D, the underlying communication costs are directly proportional to the amount of data (Pl) being transfered between two processors [6]. Note that the wormhole routing model is assumed in all the experiments conducted in this paper.

Since the computation time of a node and the corresponding data volume are random variables, manipulating the computation time of two vertices involves a function of two random variables, namely $Z = f(X, Y)$ [4, 18]. In this study, function f can be, for instance, addition and maximum operations. Assume that X and Y are random variables representing the computation time of two vertices. The outcome of this function f is also a random variable. As an example, suppose that we would like to add computation times of n_0 with that of n_2 (timing information is shown in Figure 1(b)). The addition result is depicted in Figure 3. Likewise, the maximum operation can be done in the same fashion.

$$
+\quad \begin{array}{c|c|c|}
 & 0.7 & 0.3 \\ \hline
t_0 & 2 & 4 \\ \hline
t_2 & 1 & 4 \\ \hline
 & 0.2 & 0.8
\end{array}
\qquad \Rightarrow \qquad
\begin{array}{c|c|c|c}
3 & 5 & 6 & 8 \\
0.14 & 0.06 & 0.56 & 0.24
\end{array}
$$

Fig. 3. Adding two random variables $T = t_0 + t_2$

The *retiming* operation rearrange delays in a data-flow graph so as to reduce the longest path of the graph [14,15,17]. Retiming of n_u ($r(n_u) = 1$) is equivalent to deleting one delay from *each* incoming edge of n_u and adding one delay on *each* outgoing edge of this node. A value of a retiming function reflects a number of delay moved through the retimed node. The example of this operation is shown in Figure 4 where $r(n_0) = r(n_1) = 1$. Originally $n_3 \stackrel{2}{\longrightarrow} n_0$ has 2 delays and with $r(n_0) = r(n_1) = 1$, one delay from $n_3 \stackrel{2}{\longrightarrow} n_0$ is moved from this edge to each of n_0 outgoing edges. The function $r(n_1) = 1$ then removes one delay from $n_0 \stackrel{1}{\longrightarrow} n_1$ and places it on $n_1 \stackrel{1}{\longrightarrow} n_3$ (see Figure 4(b)). After retiming, some operations (nodes in the graph) called prologue must be computed before entering the loop body—the original data-flow graph represents the whole entity of this loop. The number of prologue operations is directly related to values from the retiming functions [2]. On the other hand, some operations called epilogue must also be computed after the loop due to the reshaping of the loop body indices. To show how the retiming goal can be achieved on a graph in Figure 4(a), let each nodes in the graph have a unit computation time. After retiming the graph the longest path—a path comprises of edges with no delays and has the largest value of summation of the node computation times on this path, is shorter in the retimed graph. In a literature, such a path is also referred to the *clock period* of a graph.

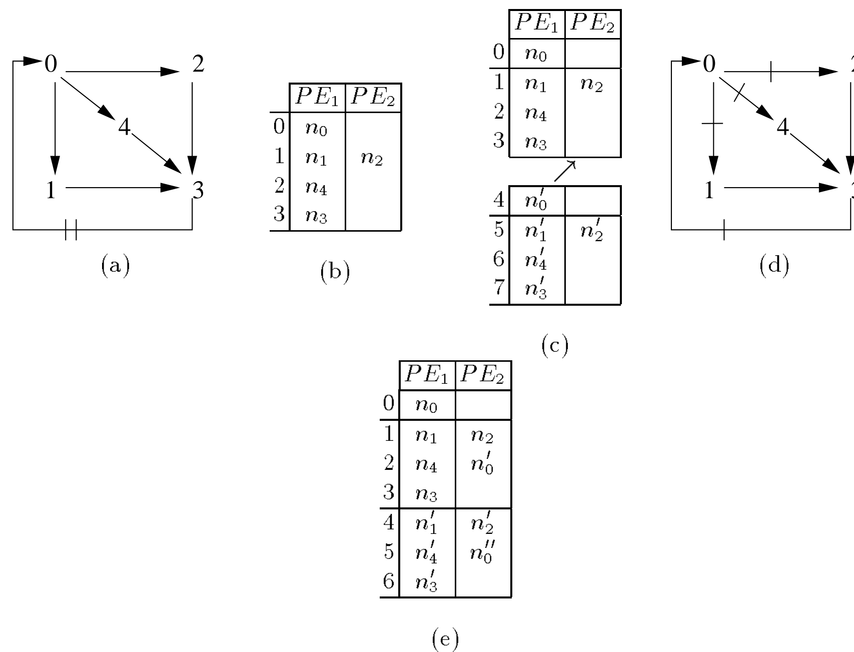


(a) (b)

Fig. 4. Retiming example where $r(n_0) = r(n_1) = 1$

Rotation scheduling employs the idea of reshaping a loop body [2]. It implicitly uses retiming to explore the parallelism across iterations. Given a legal

schedule—a schedule where all precedence relations are satisfied, and a corresponding data-flow graph, rotation scheduling incrementally retimes nodes from the first row of the table and reschedules them so as to reduce the schedule length. The idea behind this technique is that the dependency distance associated with all the parents and children of the rotated nodes will change due to the retiming property. Rotating a node introduces an opportunity to parallelize it with others in the graph. This process is equivalent to taking nodes from the first row of the next iteration to be rescheduled with the others. As an example, considering the scheduling problem when a limited number of target processors is accounted. Given a schedule table (see Figure 5(b)) which complies with all precedence relations the data-flow graph (see Figure 5(a)), rotation scheduling reshapes this repetitive loop body by including n'_0 from the next adjacent iteration and puts n_0 at the first iteration as a prologue operation (see Figure 5(c)). This action can be described by a retimed graph in Figure 5(d) where n_0 in Figure 5(a) is retimed by one. This node (n'_0) is then rescheduled to a new position such that the total computation time of the iteration is reduced. In this example, after performing rotation once, the schedule length or the time to finish executing one iteration, is reduced by one time step. Note that performing rotation can be done repeatedly until there will be no more reduction.



	PE_1	PE_2
0	n_0	
1	n_1	n_2
2	n_4	
3	n_3	

(b)

	PE_1	PE_2
0	n_0	
1	n_1	n_2
2	n_4	
3	n_3	
4	n'_0	
5	n'_1	n'_2
6	n'_4	
7	n'_3	

(c)

	PE_1	PE_2
0	n_0	
1	n_1	n_2
2	n_4	n'_0
3	n_3	
4	n'_1	n'_2
5	n'_4	n''_0
6	n'_3	

(e)

Fig. 5. Rotation scheduling example

3 Loop Scheduling & Communication

Our scheduling problem on parallel processing system where node computation times and communication costs can be varied has not yet been addressed in the previous work. Due to these uncertainties, a scheduler should also produce a schedule with uncertain schedule length. When considering communication, the message collision issue arises and complicates the scheduling problem. Therefore, to obtain a good static schedule which allows the corresponding application to have a shorter completion time in most cases becomes difficult. This section discusses these issues in more detail.

3.1 Probabilistic task&communication graph

Since this paper deals with scheduling uncertain tasks to a parallel system, the concept of control step (synchronization time used to initiate operations in an application) is no longer valid. Rather, the resulting static schedule is concerned with the *order* of nodes to be executed in a processor and how to bind each node to which processor. Furthermore, when it comes to the case that two dependent nodes, e.g., $n_u \longrightarrow n_v$, are bound to different processors, this order should be able to explicitly express this situation. In order to cope with this scenario, a graph called *probabilistic task&communication* is introduced.

Definition 2. *A probabilistic task&communication graph (PT&C-G) is a directed weighted acyclic graph $G = \langle V, E, w \rangle$ where v is a set of nodes, E is a set of precedence relations between any two nodes, and w is a function from V to random variables.*

An *execution order* represented by the PT&C-G is the precedence relations defined by the graph. This graph implies the execution of one *iteration* of a loop body in the corresponding application. Each node in one iteration will be computed exactly once according to the dependency order in the graph. The synchronization will be applied at the end of each iteration. In other words, all the nodes including the communication latencies, which may occur across an iteration due to loop carried dependences, must finish before starting the next iteration. All nodes from the PEG plus extra communication nodes—which are added to the graph when scheduling any two dependent nodes to different processors (assuming there is no communication costs when they are scheduled in the same processor)—will be included in this graph. Recall that the timing of a communication node can be derived from the data volume information in the PEG. Hence, a computation node n_i in a PTC-G is associated with x_i because its communication latency is computed from the amount of data volume of n_i. In the following graph examples, we put m_i next to each x_i to indicate this relationship. Edges between two computational nodes are defined according to the precedence relations in a PEG and/or the order of nodes executed in the same processor. Furthermore, extra edges between a computation node and its communication node, as well as between the communication node and the computation node

which the data is sent to are inserted. Edges between two communication nodes may be added to serialize a channel usage and avoid message collision if these nodes require the same channel to send data.

Unlike a probabilistic extended data-flow graph, a PT&C-G cannot have any cycle since it represents only one static schedule. If the target system applies wormhole routing, the communication cost can be calculated by the communication latency function presented in the previous section. It is worth noting that the PT&C-G model presented here is independent of any routing or architecture topology assumption. In other words, given different system architecture configuration, one can independently calculate the communication costs for this graph.

As an example, given a system with 4 processing elements connected as a mesh by 4 bidirectional links (channels), a PEG, and a processor assignment (see Figure 6). Let us assume that XY-routing technique [6] is employed to govern the pattern of sending data from one processor to the other. Suppose that Figure 6(c) is a processor assignment of this graph. A corresponding PT&C-G can be constructed as shown in Figure 6(d). Nodes from the original PEG $(n_0, n_1, n_2, n_3, n_4)$ are included in the PT&C-G where the weights w for these node are their computation times. Since n_1 is scheduled to the same processor as n_0, its parent, there is no communication node needed. Therefore an edge from n_0 to n_1 is presented in the graph. As for n_2 and n_4, a communication node x_0 is inserted in between their parent n_0 and each of them. Here we also assume that there is no message collision occurred when multicasting a message (sending a message from n_0 to both n_2 and n_4). Considering n_3, an edge from n_2 is added between them. Due to XY-routing behavior where a message is routed in X direction first and then in Y direction, sending a message from both n_1 and n_4 to n_3 uses different routes. Consequently, x_1 and x_4 are added to the graph just before n_3. Finally, since $n_3 \xrightarrow{2} n_0$ from the PEG and n_3 is scheduled PE_3 while n_0 is in different processor, this communication must be completed before the next iteration begins. The communication node x_3 is appended to the PT&C-G.

Let us consider a message collision scenario where communication nodes may be serialized due to the limited usage of a physical channel. For demonstrating the contention scenario, assume that no virtual channel [6] is employed. Therefore, if more than two messages are requesting for the same channel, e.g., the channel is still in used while there is another request for the channel, these two requests will have a message contention. As an example, consider Figures 6(a) and 6(b). If the processor assignment in Figure 7(a) is used in place of the one in Figure 6(c), the resulting PT&C-G will be constructed as presented in Figure 7(b). In this case, n_3 is scheduled to PE_2 where none of its parents are assigned to. Communication nodes are then required for the message sending to n_3. By observing Figure 7(b), there is a possible contention on channel between PE_0 and PE_2, because n_1 in PE_0 and n_2 in PE_1 use the same channel to send their data to n_3 in PE_3. Therefore, the communication nodes x_1 and x_2 may need to be serialized. Assume that x_1 gets to use this channel first, then x_2 has

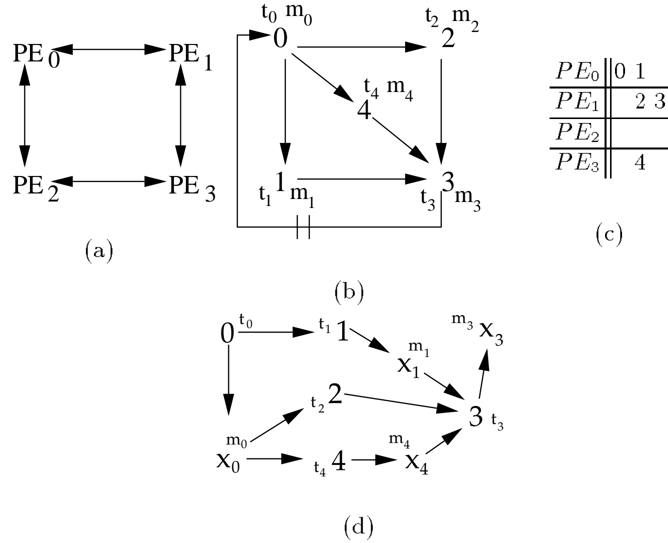


Fig. 6. Example of how to construct a PT&C-G from a PEG and its processor assignment

to wait for x_1 to finish and that explains the edge from x_1 to x_2. Since there is no possible contention with a message sending from n_4, no serialization is done for its communication node (x_4). The serialization order of such communication nodes, e.g., either x_1 before x_2 or vice versa, is important when evaluating how good this schedule order will be. If routing channels are allocated by hardware in a first come first serve fashion, this order may be changed due to varying node computation times. Hence, it results in different completion time of the PT&C-G. The following subsection discusses how this graph can be probabilistically evaluated.

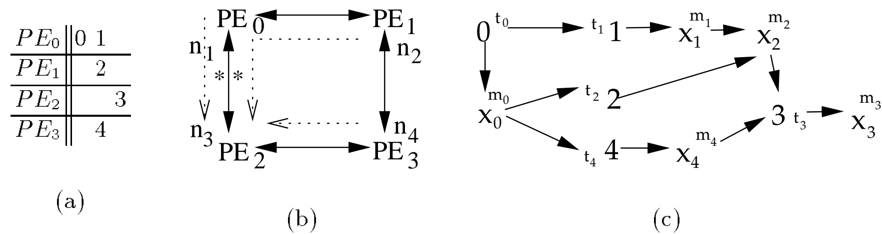


Fig. 7. Example of a PT&C-G where there exists a contention

3.2 Evaluating probabilistic schedule length of PT&C-G

Suppose a PT&C-G has only two nodes, e.g., $n_u \longrightarrow n_v$ where t_u and t_v are independent of each other. The schedule length of this graph can be computed by probabilistically adding these two random variables. On the other hand, if these two nodes do not connect to each other, the max operation can be used to compute the length as presented in Figure 8. Both "add" and "max" operations are two fundamental functions for evaluating the schedule length of the graph. Not all the node timing information (random variables) in PT&C-G are, however, independent of one another. Nevertheless, thoroughly checking the dependency of these random variables and efficiently calculating a probabilistic schedule length considering both dependent and independent cases are difficult. Since these communication nodes also cause the dependence of random variables when performing those basic probabilistic operations, here we dependently add the computation time of the node and its communication time, to be more realistic in estimating probabilistic schedule length.

Considering that n_v depends on data produced by n_u but these nodes are scheduled to different processors. According to our assumption, x_u, a communication node of n_u, must be inserted in between $n_u \longrightarrow n_v$. If $(t_u = 2, m_u = 1)$ with 0.3 probability and $(t_u = 5, m_u = 3)$ with 0.7 probability, one property that must be held is that if n_u takes 2 time units, this node must also produce 1 volume of data. Therefore, adding t_u and x_u could be done by adding outcomes which are associated with the same probability together, e.g., $t_u + m_u = 2 + 1$ with 0.3 probability and $t_u + m_u = 5 + 3$ with 0.7 probability. In our case we call this operation *adding two dependent random variables.*

max

	0.7	0.3
t_u	2	4
t_v	1	4
	0.2	0.8

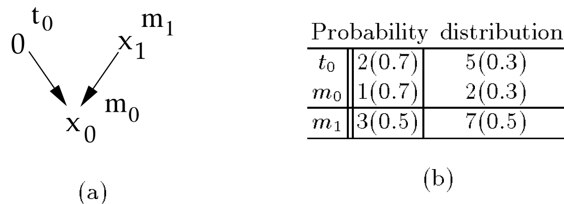
$\Rightarrow$

max(2, 1)	max(2, 4)	max(4, 1)	max(4, 4)
2	4	4	4
0.14	0.06 ↘	0.56 ↓	0.24 ↙
0.14	0.86		

Fig. 8. Maximizing two random variables $M = \max(t_u, t_v)$

It becomes more complex when the serialization of the communication nodes occurs in the graph. Consider a typical segment of a PT&C-G in Figure 9(a). The probabilistic value associated with these nodes are presented in Figure 9(b). If x_0 was any computation node, says n_y for example, we would have probabilistically maximized t_0 with m_1 and added this result to t_y. However, the probability distribution of m_0 is dependent upon that of t_0, and therefore, some combination will not occur when adding $\max(t_0, m_1)$ to m_0.

One possible way of getting around this problem is to separate the result from probabilistic max operation based on the probability distributions of t_0 or m_0, e.g., a group of possible timing whose probability is 0.7 and the reciprocal (0.3)



Probability	distribution	
t_0	2(0.7)	5(0.3)
m_0	1(0.7)	2(0.3)
m_1	3(0.5)	7(0.5)

(b)

Fig. 9. Example of serialized communication nodes

group. When it comes to the adding, pick a value from m_0 that is associated with each group to be added with the max result. As an example, Figure 10 depicts this calculation. Value 1 from m_0 is added with $\max(2,3)$ and $\max(2,7)$ where they all are associated with 0.7 probability. Likewise, value 2 is added with the rest of the combination. Each of the addition result inherits the probability from the maximum result (boxed numbers in the figure). Given a PT&C-G, Algorithm 1 summarizes how we evaluate a schedule length from this graph.

	0.7		0.3	
	0.35	0.35	0.15	0.15
$\max(t_0, m_1)$	$(2,3)$	$(2,7)$	$(5,3)$	$(5,7)$
	+		+	
m_0	1		2	
$m_0 + \max(t_0, m_1)$	4	8	7	9

Fig. 10. Maximizing and adding dependent random variables

In this algorithm, $weight(u), start_time(u)$, and $finish_time(u)$ are data structure that hold their corresponding random variables. The probabilistic timing information (computation or communication time) of n_u can be retrieved by calling $weight(u)$. Either dependently or independently adding $start_time(u)$ and $weight(u)$ to establish $finish_time(u)$, a finishing time of n_u which will then be propagated to all n_u's successors. The starting time of a node comes from probabilistically maximizing finishing time of all the node's predecessors. Note that if n_u is a communication node and when adding its starting time to its weight, the operation discussed in Figure 10 may be applied. As an example, consider a PT&C-G in Figure 11.

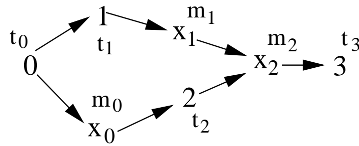
The algorithm starts off with n_0 where its computation time t_0 gets propagated to n_1 and x_0. Since n_1 and x_0 have no other parents, t_0 is then added to t_1 and m_0, providing the starting time for x_1 and n_2 respectively. At this point the $finish_time(x_1)$ is holding a random variable $[4(0.15), 5(0.15), 7(0.35), 8(0.35)]$ and $finish_time(n_2)$ has $[5(0.25), 7(0.5), 9(0.25)]$. Using the strategy described in Figure 10, $finish_time(x_2)$ becomes $[6(0.075), 8(0.25), 9(0.175), 11(.1625),$

Algorithm 1 Calculate probabilistic schedule length of a PT&C-G

Input: PT&C-G $G = \langle V, E, w \rangle$

Output: Probabilistic schedule length

```
 1: start_time(u) = 0, and finish_time(u) = weight(u), ∀u ∈ V
 2: Q ← get_root(G);
 3: while Q ≠ ∅ do
 4:    u = dequeue(Q) and mark node u visited
 5:    if u is a communication node then
 6:       finish_time(u) ← dependently add weight(u) and start_time(u)
 7:    else
 8:       finish_time(u) ← add two independent random variable, weight(u) and
          start_time(u)
 9:    end if
10:    for all children v of u do
11:       start_time(v) ← maximize finish_time(u) and start_time(v)
12:       indegree(v) = indegree(v) − 1
13:       if indegree(v) = 0 then
14:          Q = enqueue(v)
15:       end if
16:    end for
17: end while
```



(a)

Probability distribution

t_0	1(0.5)	2(0.5)	m_0	2(0.5)	3(0.5)
t_1	2(0.3)	3(0.7)	m_1	1(0.3)	3(0.7)
t_2	2(0.5)	4(0.5)	m_2	1(0.5)	4(0.5)
t_3	2(1.0)				

(b)

Fig. 11. PT&C-G for computing probabilistic schedule length

$12(0.0875), 13(0.25)]$ which will then be added to t_3 to complete the calculation for the probabilistic schedule length. Since the computation time of n_3 is fixed, this value is then added to each of the previous possible outcome resulting in $[8(0.075), 10(0.25), 11(0.175), 13(.1625), 14(0.0875), 15(0.25)]$. According to this result, one can conclude that approximately 75% of the time, this graph will have its schedule length (sl) less than or equal to 14, or $P(sl \leq 14) = 0.75$.

3.3 Loop scheduling algorithm

In order to get a good static schedule where the total completion time of the schedule is minimized, the rotation scheduling concept presented in Section 2 is employed. Traditional rotation scheduling, however, does not consider communications between nodes. Furthermore, it has no concept of how to deal with uncertain timing information. Therefore, this algorithm needs to be extended to consider these problems while computing a schedule. Algorithm 2 presents the framework which handle the above considerations. Based on this framework, some detailed modifications are required in each routine. First, an algorithm for producing the initial schedule such as list scheduling [7, 9] needs to be modified so as to consider probabilistic computation and communication timing information. The variation of list scheduling algorithms differ from one another by how each algorithm prioritized the list. Note that this list keeps nodes (called ready nodes) whose all their parents have been scheduled. The scheduler will select a node with the highest priority to be scheduled to a processor so that a resulting intermediate schedule is in the most compact form of the current schedule length. New ready nodes may be inserted into the list during this time. The above sequence is repeated until the ready list is empty and no more nodes to be added to the list. Having discussed the list scheduling process, it is clear that if a graph contains uncertain timing information, the fundamental question here is how to compare two random variables. In addition, if the communication issue is taken into account, the scheduler must be able to handle the message collision case.

To compare two random variables, considering the example in Figure 12. If t_u and t_v are to be compared in this case, one can set up a probability value to help decide if one is bigger or smaller than the other. As in the example, we obtain that t_v is smaller than t_u with a 0.56 probability. If a probability constraint is set to 50%, then t_v is really smaller than t_u according to this constraint. This approach can be used to prioritize the list in list scheduling. For deciding where to schedule a node, one can tentatively schedule the node to different possible positions and select the one which contributes the smallest length among all of its intermediate probabilistic schedule lengths. Not only does this comparison technique apply to list scheduling, but also is used in Algorithm 2 where rotation scheduling stores its best resulting schedule and processor assignment.

Regarding to the communication issue, communication channel usage information must be kept so that the scheduler knows if there exists a contention [22]. The communication node concept presented in the previous subsection can be applied to acknowledge when a channel is released. When the contention occurs,

Algorithm 2 Rotation scheduling framework

Input: PEG $G = \langle V, E, \delta, m, t \rangle$, number of processors and their connections
Output: PT&C-G G_s, processor assignment $proc$, and retiming function r

1: $(G_{s_cur}, proc_cur) \leftarrow$ initial schedule /* produce an initial order and processor assignment */
2: $G_s \leftarrow G_{s_cur}$, $proc \leftarrow proc_cur$
3: **for** $i = 1$ to $2|V|$ **do**
4: $\quad u \leftarrow$ pick rotated node from G_{cur}
5: $\quad r(u) = r(u) + 1$ /* retime node u */
6: $\quad proc_cur \leftarrow$ find available processor for u
7: $\quad G_{s_cur} \leftarrow$ reschedule u to G_{s_cur}
8: $\quad$**if** $better(G_s, proc_s, G_{s_cur}, proc_cur)$ **then**
9: $\quad\quad G_s \leftarrow G_{s_cur}, proc \leftarrow proc_cur$ /* store the best schedule and processor assignment */
10: $\quad$**end if**
11: **end for**

	0.2	0.8
t_u	1	4
t_v	3	7
	0.7	0.3

$\Rightarrow$

$t_u < t_v$	$(1 < 3)$ 0.14	$(1 < 7)$ 0.06	$(4 < 7)$ 0.24	0.44
$t_u > t_v$	$(4 > 3)$ 0.56			0.56

Fig. 12. Comparing which random variable is greater

serializing communication nodes helps us approximate the real situation. One can use the above comparison method to suggest where possible contentions can happen. Once the contention is detected, the communication nodes are serialized. Note that possible serialization of the communication nodes may be generated, and the above technique may be used as a tool to select a sequence of communication nodes which yields the shortest possible intermediate schedule length. As an example consider the segment of the PT&C-G from Figure 7(c) again in Figure 13. Suppose that the contention occurs when n_1 and n_2 send data to their children. If t_1 is much larger than t_2, the segment in Figure 13(b) should be smaller than the previous one because the time consuming portion has been changed from (t_1, m_1, m_2) to (t_1, m_1). Communication node x_1 does not need to wait for the available channel, previously occupied by x_2, since by the time n_1 finishes, n_2 and x_2 will be done. On the other hand, in Figure 13(a), x_2 needs to be idle, waiting for x_1 to finish using channel before it can send the message.

By incorporating those aforementioned techniques, we can construct an initial schedule which is represented by a PT&C-G and a processor assignment. Recall that if an input graph has cycles which imply inter-iteration dependences (delays), list scheduling will ignore these cycles and consider only edges with no delays. After a PT&C-G is derived, corresponding nodes from the PEG to which non-zero delay edges are connected, will be checked with their processor

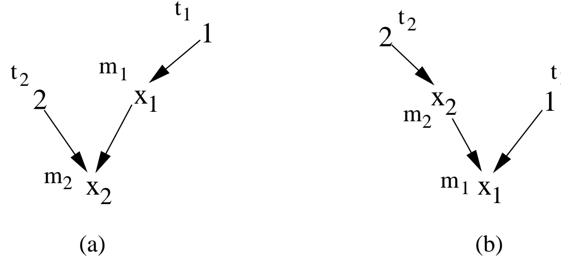


Fig. 13. Comparison between two serializations

assignment if the communication nodes should be added to the PT&C-G (source and sink nodes are scheduled to different processors). At this point the modified rotation scheduling algorithm is able to perform loop pipelining. Nodes at the top of a schedule—nodes which are ready to be executed first in this schedule, will be selected to be rotated. (Recall that these rotated nodes will be retimed and dependency distance on the edges connected to these nodes will be changed accordingly.) For the rescheduling part, the algorithm will again search for a position to reschedule these nodes one by one so that the resulting PT&C-G is smallest among all other positions. The comparison technique discussed earlier can be used here to help select a good PT&C-G. Since, the order of the nodes in this graph may be different depending on the processor assignment, the message collision and communication channel usage should be carefully investigated at each step. The new intermediate PT&C-G and processor assignment will be created for every rescheduling process. This allows the algorithm to evaluate how good each schedule is and select the best PT&C-G and its processor assignment as its final result.

4 Experiment

In this section, we perform experiments using our algorithm on several benchmarks. These benchmarks were obtained from examples used in [10] and from a fuzzy rule-based system which assumes 5 rules, 3 linguistic variables whose fuzzy sets contain 64 elements, and 50% overlap, centroid as a defuzzification method, and max-min inference [21]. These inputs were modified to serve the purpose of the experiment in this paper. Nodes from two mutual exclusive branches are grouped together to create a node in a PEG. A probabilistic computation time of a node in the PEG is then obtained from putting a node computation time and its probability (inherited from the probability of taking a branch) together with the one from the reciprocal branch. All arithmetic operations are assumed to take 2 time units except the multiplication and the division which take 3 time units. In this experiment, the probability of taking each branch was arbitrarily given. Note that in practice these probability values can be obtained by an analysis of the program structures or a profiling methodology [20]. The size of data volume

associated with each new node is assumed to be a number of outgoing edges of each grouped node. All target architectures in these experiments are 2-D mesh. The XY-routing approach is assumed in each target system for determine how a message is delivered to a destination. In order to compute a communication overhead for each message, we assume that wormhole routing with no virtual channel technique is applied in all systems. The algorithm used the data volume information in the PEG to create the communication cost (wormhole routing latency) used in constructing a PT&C-G when a message is transfered from one processor to the other. The result of these experiments are compared with the technique presented in [22] where each uncertain timing information in those input graphs assume their average values.

Table 1. Results from list scheduling v.s. probabilistic version, tested for 4 processors

Ben.	#nds	Average case		Probabilistic case	
		range	$P(sl<?)\approx 0.9$	range	$P(sl<?)\approx 0.9$
liu1-uf-2	22	(17,42)	34	(14,31)	26
liu2	16	(29,51)	48	(29,51)	48
liu2-uf-2	32	(45,85)	76	(41,72)	65
liu2-uf-4	48	(57,96)	87	(52,91)	83

Table 2. Results from rotation scheduling v.s. probabilistic version, tested for 4 processors

Ben.	#nds	Average case		Probabilistic case	
		range	$P(sl<?)\approx 0.9$	range	$P(sl<?)\approx 0.9$
liu1-uf-2	22	(17,39)	33	(13,29)	21
liu2	16	(23,44)	41	(25,40)	38
liu2-uf-2	32	(45,85)	76	(40,68)	62
liu2-uf-4	48	(52,91)	82	(50,89)	80

Table 1 shows results obtained from running a modified version of list scheduling (probabilistic list scheduling). The target architecture of this table is four processors connected as a mesh. Column *ben.* presents the list of selected benchmarks while its adjacent column presents the number of nodes in each benchmark. Most of them are obtained from [10]. The benchmarks with suffix "uf-x" are unfolded graph where "x" represents the unfolding factor. Column *average case* presents results given by running traditional list scheduling with the benchmark graphs assuming average node computation and communication times. The results in this column are compared with the results in Column *probabilistic case* where results from running the modified list scheduling algorithm (considering probabilistic timing information) are presented. Inside each of these two

columns, there are two sub-columns called range and $P(sl <?) \approx 0.9$. The former one shows spanning range (minimum and maximum values) of the resulting schedule lengths. The latter shows how long the schedule length (sl) can be expected if a confidence probability is given (in this experiment this confidence is approximately 90 percent, i.e., $P(sl <?) \approx 0.9$. Note that for Column *average case*, in order to compare both average and probabilistic approaches, the probabilistic timing information were substituted back to the resulting PT&C-G and then Algorithm 1 was used to evaluate the schedule lengths.

Table 3. Simulation results for tested benchmarks: 4 processors

Ben.	#nds	Average case		Probabilistic case	
		list	rot.	list	rot.
liu1-uf-2	22	25.78	23.71	22.76	17.40
liu2	16	42.77	39.32	42.77	31.55
liu2-uf-2	32	62.45	62.45	57.91	54.91
liu2-uf-4	48	72.31	69.91	69.52	69.11

Table 2 presents the comparison between results obtained by running the algorithm presented in [22] with the proposed algorithm's. The definition for each column is the same as that of Table 1. Tables 4 and 5 discuss the same issue but with eight processors as the target system. Since those results shown in previous tables were all evaluated from the PT&C-G model where we used our heuristic to decide how to serialize communication nodes when the message collision occurs, such a serialization pattern that we assumed might not happen every time in practice. Therefore, in order to show the effectiveness of the proposed approach, the simulation on each experiment was performed. The simulation takes the following as its inputs: the processor assignment, the order of nodes within each processor, and the benchmark graph (for list scheduling) or retimed graph (for rotation scheduling). A fixed timing information for each node will be randomly selected from the range of uncertain time according to its probability. The simulator then performs the simulation regardless of where

Table 4. Results from list scheduling v.s. probabilistic version, tested for 8 processors

Ben.	#nds	Average case		Probabilistic case	
		range	$P(sl<?)\approx 0.9$	range	$P(sl<?)\approx 0.9$
liu1-uf-2	22	(18,41)	34	(14,31)	26
liu2	16	(29,51)	48	(29,51)	48
liu2-uf-2	32	(45,85)	76	(41,72)	65
liu2-uf-4	48	(57,96)	87	(52,91)	83
liu3	79	(50, 198)	124	(69,165)	95
fuzzy	88	(75,904)	792	(75,645)	541

Table 5. Result from rotation scheduling v.s. probabilistic version, tested for 8 processors

Ben.	#nds	Average case		Probabilistic case	
		range	$P(sl<?)\approx 0.9$	range	$P(sl<?)\approx 0.9$
liu1-uf-2	22	(17,31)	28	(13,29)	21
liu2	16	(23,43)	38	(31,38)	35
liu2-uf-2	32	(44,74)	67	(42,58)	58
liu2-uf-4	48	(50,89)	70	(47,73)	67
liu3	79	(69,165)	94	(38,135)	92
fuzzy	88	(63,557)	466	(63,362)	312

communication nodes are put in the PT&C-G. The results from the simulator operating on each schedule were collected for a thousand times and only average values of these results are presented in the Tables 3 and 6. First Table 3 presents the simulation of the results from Tables 1 and 2 while Table 6 presents the simulation of those results shown in Tables 4 and 5. In these simulations, Column *average case* presents results obtained from running the simulator on those processor assignments generated by traditional list and rotation scheduling while column *probabilistic case* presents the simulation results given by the proposed scheduling algorithms.

Table 6. Simulation results for tested benchmarks: 8 processors

Ben.	#nds	Average case		Probabilistic case	
		list	rot.	list	rot.
liu1-uf-2	22	25.87	19.23	21.42	17.34
liu2	16	42.77	33.77	39.32	30.77
liu2-uf-2	32	62.45	59.39	57.91	50.61
liu2-uf-4	48	64.67	65.82	62.23	58.48
liu3	79	73.36	67.94	71.30	51.63
fuzzy	88	690.41	321.96	448.77	292.12

5 Conclusion

We have presented a loop pipelining algorithm for scheduling nodes with varying computation times for parallel systems while considering the communication costs. These communication costs can be uncertain since they are associated with the amount of data generated by those uncertain computation nodes. Moreover, such costs only occur when data is transferred between two processors. The proposed algorithm reflects the message collision issue by introducing a graph model called the probabilistic task&communication graph. Communication nodes are separated from the computation nodes and can be serialized if the contention

occurs. We also proposed a method for evaluating the underlying probabilistic schedule length. The result obtained from this evaluation can help designer decide how long in practice a probabilistic schedule will take. Experimental results show the effectiveness of our algorithm. By comparing the results obtained from the schedule produced by our method and the traditional approach which assumes the average computation time, the resulting schedule length obtained from our schedule is shorter than using that obtained by the traditional approach on average.

References

1. U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pages 192–219. IEEE, August 1990.
2. L. Chao, A. LaPaugh, and E. Sha. Rotation scheduling: A loop pipelining algorithm. In *Proceedings of the 30th Design Automation Conference*, pages 566–572, Dallas, TX, June 1993.
3. L.-F. Chao and E. H.-M. Sha. Static scheduling for synthesis of DSP algorithms on various models. *Journal of VLSI Signal Processing*, 10:207–223, 1995.
4. W. Feller. *An introduction to probability theory and its applications.* John Wiley & Sons, New York, 1968.
5. E. M. Girczyc. Loop winding—a data flow approach to functional pipeline. In *Proceedings of the International Symposium on Circuits and Systems*, pages 382–385, May 1987.
6. K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill Series in Computer Science, New York, NY, 1993.
7. R. A. Kamin, G. B. Adams, and P. K. Dubey. Dynamic list-scheduling with finite resources. In *Proceedings of the 1994 International Conference on Computer Design*, pages 140–144, Cambridge, MA, October 1994.
8. I. Karkowski and R. H. J. M. Otten. Retiming synchronous circuitry with imprecise delays. In *Proceedings of the 32nd Design Automation Conference*, pages 322–326, San Francisco, CA, 1995.
9. A. A. Khan, C. L. McCreary, and M. S. Jones. A comparison of multiprocessor scheduling heuristics. In *Proceedings of the 1994 International Conference on Parallel Processing*, volume II, pages 243–250, 1994.
10. T. Kim, N. Yonezawa, J. W. S. Liu, and C. L. Liu. A scheduling algorithm for conditional resource sharing—a hierachical reduction approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):425–438, April 1994.
11. D. Ku and G. De Micheli. *High-Level synthesis of ASICS under Timing and Synchronization constraints.* Kluwer Academic, 1992.
12. D. Ku and G. De Micheli. Relative scheduling under timing constraints: Algorithm for high-level synthesis. *IEEE transactions on CAD/ICAS*, pages 697–718, June 1992.
13. M. Lam. Software pipelining. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, GA, June 1988.
14. C. E. Leiserson and J. B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, 1983.

15. C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.
16. W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. Technical Report TR 92-1294, Cornell University, Ithaca, NY, July 1992.
17. B. Lockyear and C. Ebeling. Optimal retiming of multi-phase, level-clocked circuits. In T. Knight and J. Savage, editors, *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*, pages 265–280, Cambridge, MA, 1992.
18. P. L. Meyer. *Introductory Probability and Statistical Applications.* Addison-Wesley, Reading, MA, 2nd edition, 1979.
19. N. L. Passos, E. Sha, and S. C. Bass. Loop pipelining for scheduling multi-dimensional systems via rotation. In *Proceedings of the 31st Design Automation Conference*, pages 485–490, June 1994.
20. D. A. Patterson and J. L. Hennessy. *Computer architecture: A Quantitative approach.* Morgan-Kaufman, 1996.
21. T. J. Ross. *Fuzzy Logic with Engineering Applications.* MC-Graw Hill, 1995.
22. S. Tongsima, E.-H. Sha, and N. L. Passos. Communcation-sensitive loop scheduling for DSP applications. *IEEE Transactions on Signal Processing*, 45(5):1309–1322, May 1997.
23. M. E. Wolfe. *High Performance Compilers for Parallel Computing*, chapter 9. Addison-Wesley, Redwood City, CA, 1996.
24. M. E. Wolfe and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.
25. L. A. Zadeh. Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets and Systems*, 1:3–28, 1978.