Scheduling with Confidence for Probabilistic Data-flow Graphs

*Sissades Tongsima** *Chantana Chantrapornchai** *Edwin H.-M. Sha*†
Dept. of Computer Science and Engineering,
University of Notre Dame,
Notre Dame, IN 46556

Nelson L. Passos
Department of Computer Science,
Midwestern State University,
Wichita Falls, TX 76308

Abstract

One of the biggest problems in high-level synthesis is to obtain a good schedule without the knowledge of exact computation time of tasks. While the target applications in high-level synthesis are becoming larger, a task in the applications such as artificial intelligent systems or interface may have uncertain computation time. In this paper, an algorithm to schedule these repetitive tasks and optimize the schedule is presented. A probabilistic data-flow graph is employed to model the problem where each node represents a task associated with the probabilistic computation time and a set of edges represents the dependences between the tasks. A novel polynomial-time probabilistic retiming algorithm for optimizing the graph and an algorithm for computing the optimized schedule, subject to the acceptable probability and resource constraint, are presented. The optimization algorithm also guarantees to give such a short schedule length with a given qualitatively provable, confidence level. The experiments show that the resulting schedule length for a given confidence probability can be significantly reduced.

1 Introduction

During the initial design phase, the execution time of a task may be assigned either a fixed "worst case" or "average case" value. Nevertheless, in reality, the execution time of tasks may vary due to a number of factors such as fabrication variation, module selections, input-driven sensitivity, etc. Furthermore, in many applications such as interface systems, fuzzy systems, and artificial intelligence systems, etc., some of the tasks normally have varying execution times. Hence, after the system is implemented, it usually does not give the actual maximum performance. In order to correct the issue of varying timing characteristics, costly and time consuming redesign cycles are often required. A repetitive refinement of the design is necessary to adjust the system to obtain the practical results after the low-level detailed design are finished [12]. With the current design methodologies, more than 40 redesign cycles may have to be performed. Therefore, it is important to develop techniques that can produce a good initial schedule so that the number of redesign cycles can be significantly reduced. Furthermore, such a schedule should also be guaranteed to achieve an expected performance within a given qualitatively provable, confidence level.

In many iterative applications, the statistics of the computation times of uncertain tasks are not difficult to be collected. By taking advantage of these statistical data, the schedule which gives the qualitatively provable performance can be constructed. To achieve such a goal, the proposed algorithm applies a transformation called *probabilistic retiming* which optimizes an input application without considering resource constraints for given a confidence level. The input application is modeled as a hierarchical data-flow graph (DFG) where a node corresponds to a task, e.g., a collection of statements, and a set of edges represents dependencies between these tasks. The dependency distances or *delays* between tasks in different iterations is represented by short bar lines on those edges. The computation time of these nodes can be either fixed or varied. To handle these cases, in this research, a probability model is employed to model the timing of these tasks. Then the probabilistic task scheduling is applied to effectively schedules both certain and uncertain tasks to multiple functional units. After that, the total execution time of the application with a confidence probability is calculated.

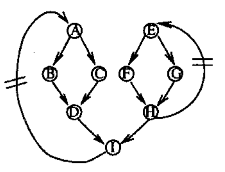
Considerable research has been conducted in the area of scheduling nodes from directed-acyclic graphs (DAGs), obtained by ignoring edges with delays. Many heuristics have been proposed, e.g., list scheduling, and graph decomposition [6, 8], to schedule such a graph. However, these techniques did not investigate the parallelism across iterations, i.e., overlapping the computation of tasks by considering the edges delays. Loop transformations are also used to restructure loops in order to reduce the total execution time of a problem [1, 11, 14, 15]. However, these techniques assume the systems have neither limited resources nor probabilistic tasks. For the class of global scheduling, *software pipelining* [9] is used to overlap instructions, exposing parallelism across iterations. This technique, however, expands the graph by unfolding it. Furthermore, such approach is limited to solve

*was supported in part by the Royal Thai Government Scholarship
†was supported in part by the NSF CAREER grant MIP 95-01006

problems without considering the uncertainty of the computation time [4,9].

The traditional *retiming* [10] technique was adopted to reduce the total computation time along the critical paths in a DFG for nonresource-constrained problems. By applying retiming, the dependencies across iterations are explored. The graph is transformed in such a way that the parallelism is exposed but the behavior of the graph is preserved. In [2,3], a methodology, called rotation scheduling, was developed to address the problem of assigning tasks under limited number of processing elements. Nonetheless, those techniques assumed tasks with exact computation time. Recently, Karkowski and Otten introduced a model to handle the imprecise propagation delay of events [7]. In their approach, the fuzzy set theory [16] was employed to model imprecise computation time. This model is, however, restrict to a simple triangular fuzzy distribution and does not consider probability values.

In this paper, we employ the idea of pipelining tasks by using a probabilistic retiming algorithm. The computation time of the tasks is represented by random variables which can be modeled as a probability distribution. For example, one might say that for 20% of the time, task X will take 2 time units to execute. The remaining occurrences of X require 3 time units. Figure 1(a) illustrates a DFG G, representing a set of tasks to be scheduled. The set of vertices consists of the nodes A, B, C, D, E, F, G, H, and I. Considering T_v to be a random variable representing the computation time[1] for some node v, a possible probability distribution of the execution time for those nodes, denoted by $p(x) = \mathcal{P}(T_v = x)$ where x represents the possible computation time, is presented in Figure 1(b).



(a)

	A	B	C	D	E	F	G	H	I
$p(1)$	0.3	0	0	0.9	0	0.5	0.9	0.5	0
$p(2)$	0.7	0.8	0	0.1	0.5	0	0	0	0.5
$p(3)$	0	0	0.5	0	0	0.5	0	0.5	0.5
$p(4)$	0	0	0.5	0	0.5	0	0.1	0	0
$p(5)$	0	0.2	0	0	0	0	0	0	0

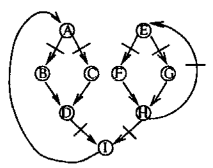
(b)

Fig. 1. An example of a 9-node graph

Intuitively, such a graph can be probabilistically retimed in order to enhance the parallelism of the graph. In other words, for every cycle or iteration, the graph can finish computing in a shorter time. The measurement of each cycle is also called *cycle* period, denoted by $\Psi(G)$. By using probabilistic retiming, the placement of the delays in the graph is rearranged in such a way that the probability or confidence level of the desired computation time is acceptable. Examining the example in Figure 1, if we consider the worst case of each computation time of the nodes in the graph, the cycle period of this graph will be 14. This result can be verified by looking at the sum of the worst execution time of nodes E, G, H, and I; however, this situation might rarely happen.

[1]Each value represents a possible computation time of a task

By using retiming, the cycle period of this graph can be reduced as presented in Figure 2(a). After observing the graph, we found that with confidence level being higher than 90%, this graph will have the cycle period smaller than or equal to 6. To verify this fact, Figure 2(b) presents the probability of the sum of the computation time of the nodes in paths containing no delays. We know that these paths cause the cycle period of the graph. We can compute the set of possible maximum values out of these distributions which is the set $\{4, 5, 6, 7\}$. The probability associated with these possible values are presented in the last row of Figure 2(b). Notice that the probability of the possible maximum $T = 7$ is less than 10%, $\mathcal{P}(T = 7) < 0.1$, i.e., it is possible that, for more than 90% of the iterations, this graph will have a computation time less than 7.



(a)

	$p(2)$	$p(3)$	$p(4)$	$p(5)$	$p(6)$	$p(7)$
$B \overset{p}{\leadsto} D$	0	0.72	0.08	0	0.18	0.02
$C \overset{p}{\leadsto} D$	0	0	0.45	0.5	0.05	0
$F \overset{p}{\leadsto} H$	0.25	0	0.5	0	0.25	0
$G \overset{p}{\leadsto} H$	0.45	0	0.45	0.05	0	0.05
$I \overset{p}{\leadsto} A$	0	0.15	0.5	0.35	0	0
max(T)	0	0	0.158	0.3836	0.3895	0.069

(b)

Fig. 2. The retimed graph and the distribution of no-delay paths

By observing the retimed graph in Figure 2(a), we know that, with no constraints on the number of processing elements (PEs), we can assign the tasks to the PEs as presented in Figure 3(a). Also, according to the new dependencies of the transformed graph, we can compute a schedule table considering a limited number of processing elements. Figure 3(b) shows a schedule for a system with only 3 PEs. Such a schedule will be able to guarantee that higher than 92% of the time, the entire set of tasks will be computed in less than 11 time units. In order to explain how to schedule the tasks according to a given confidence level of the execution time, 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 in this section. The probabilistic retiming is discussed in Section 3. Section 4 presents the scheduling under resource constraints algorithm. Experimen-

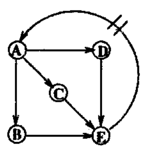
PE1	PE2	PE3	PE4	PE5
B	C	I	F	G
D	E	A	H	

(a)

PE1	PE2	PE3
B	C	E
F	G	I
H	D	A

(b)

Fig. 3. Scheduling with different number of processing elements



(a) PG

$v \in V$	$T_v = x$		$t(x) = \mathcal{P}(T_v = x)$		
	x_1	x_2	$t(x_1)$	$t(x_2)$	$\sum t(x)$
A	2	3	0.8	0.2	1
B	1	2	0.5	0.5	1
C	1	2	0.7	0.3	1
D	1	2	0.9	0.1	1
E	1	3	0.9	0.1	1

(b) probability distribution

Fig. 4. The probabilistic graph example

tal results are discussed in Section 5. Finally, Section 6 concludes the contributions of this research.

2 Preliminaries

We now introduce the graph model which is us d to represent tasks that are characterized by the uncertainty of the computation time.

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 the tasks to be executed, E is the set of edges representing the data dependencies between vertices, d is a function from E to $\mathbb{Z}^+$, the set of positive integers, representing the dependency distance between two nodes connected by an edge, and T_v is a random variable representing the computation time a node $v \in V$.*

Note that an ordinary DFG is a special case of the PG. A *probability distribution* of T is assumed to be discrete in this paper. The granularity of the resulting probability distribution, if necessary, depends on the need of accuracy. The probability that the random variable T assumes the value x, denoted by $\mathcal{P}(T = x)$, is used to represent each of the possible computation time of a task. Each vertex $v \in V$ is weighted with a probability distribution of the computation time, $\mathcal{P}(T_v = x)$, where T_v is the discrete random variable associated with the set of possible computation times of the vertex v. For any vertex, the summation of the probability of all possible values is one, i.e., $\sum_{\forall x} \mathcal{P}(T_v = x) = 1$. In this paper, we assume that all random variables are independent of each other. That is the basic random variable representing each node's computation time is independent and the resulting random variable, obtained by computing the function of any two random variables, are also independent. As an example, probability distribution for the set of vertices $V = \{A, B, C, D, E\}$ in the graph of Figure 4(a) are presented in Figure 4(b).

An edge $e \in E$, from vertices u to v, is denoted by $u \xrightarrow{e} v$ and a path p starting from a node u and ending at a node v is indicated by the notation $u \overset{p}{\rightsquigarrow} v$. The register count of a path $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{k-1}} v_k$ is $d(p) = \sum_{i=0}^{k-1} d(e_i)$. As an example, Figure 1(a) has the set of edges $E = A \xrightarrow{e_1} B, A \xrightarrow{e_2} C, A \xrightarrow{e_3} D, B \xrightarrow{e_4} E, C \xrightarrow{e_5} E, D \xrightarrow{e_6} E$, and $E \xrightarrow{e_7} A$. The delays of the edges are $d(e_7) = 2$, and $d(e_i) = 0$, for $1 \leq i \leq 6$.

The retiming method is known as a circuit optimization technique developed by Leiserson and Saxe [10]. The retiming of a graph $G = \langle V, E, d, t \rangle$ where t is a function representing the exact computation time of any node in V, is a transformation function $r : V \mapsto \mathbb{Z}$. The optimization goal of that methodology is to reduce the *clock period* or *cycle period* $\Phi(G)$ which is defined by the equation $\Phi(G) = \max\{t(p) : d(p) = 0\}$ 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 $d(p) = \sum_{i=0}^{k-1} d(e_i)$. That technique is also applicable in the high-level synthesis area.The retiming function r tells us the movement of delays with respect to a vertex so as to transform G into a new graph $G_r = \langle V, E, d_r, t \rangle$ where d_r is a new register count function for G_r.

3 Retiming with Uncertain Time

In this paper, since the computation time of each node along the path is a discrete random variable, the sum of computation times of all nodes connected in a path $u \overset{p}{\rightsquigarrow} v$ with no delays establishes a *varying* clock period. As mentioned in the previous section, the summation of the computation time along a path can be obtained by adding each random variable, e.g., $A = T_1 + T_2$, where T_1 and T_2 represent the computation time of two vertices v_1 and v_2 connected by an edge e. Furthermore, the maximum among n random variables, $A_1, \ldots, A_n$, can also be computed in pairs, repeatedly. In other words, the clock period of the PG becomes a random variable, called *maximum reaching time* (mrt). $\mathsf{mrt}(u, v)$ represents the probabilistic clock period for the portion of the graph between nodes u and v.

By considering the DAG portion of the PG, edges with non-zero registers are omitted. The overall clock period of the graph, denoted by $\mathsf{mrt}(G)$, is computed by $\mathsf{mrt}(v_s, v_d)$, where v_s is the source node connected to all root-nodes, nodes that have no incoming edges, of the DAG and v_d is the sink node connected to all leaf-nodes, nodes that have no outgoing edges, of the same graph. Note that the requirements for a probabilistic graph are usually described by an acceptable computation time for the final graph, denoted by c, and a confidence level $\theta = 1 - \delta$, where δ is the acceptable probability of not achieving the required performance. In this paper, the requirement is expressed as $\mathcal{P}(\mathsf{mrt}(G) \leq c) \geq \theta$, or $\mathcal{P}(\mathsf{mrt}(G) > c) > \delta$. The goal of probabilistic retiming is to transform a PG such that the requirement can be satisfied.

The following presents an algorithm to calculate the mrt of a graph. In order to simplify the calculation, two dummy vertices with zero computation time, v_s and v_d, are added to the graph. A set of zero delay edges is used to connect vertex v_s to all root-nodes, and to connect all leaf-nodes to vertex v_d. Therefore, the $\mathsf{mrt}(v_s, v_d)$ gives the overall maximum reaching time of the graph. The timing constraint for the PG is that $\mathcal{P}(\mathsf{mrt}(v_s, v_d) < c) >$

θ, where c is the desired clock period of the graph and θ is the confidence probability. Such a requirement can be rewritten as $\mathcal{P}(\mathsf{mrt}(v_s, v_d) > c) \leq \delta$, where δ is $1 - \theta$. Furthermore, because the mrt is equal to the maximum of the total computation time of any path routing through zero delays to the same destination node, one can calculate the mrt by operating merely on the graph that has zero delay edges, i.e., DAG.

Algorithm 3.1 (Maximum reaching time)

Input : PG $G = \langle V, E, d, T\rangle$
Output: $\mathsf{mrt}(G) = \mathrm{temp}_{\mathsf{mrt}}(v_s, v_d)$

```
$G_0 = \langle V_0, E_0, d, T\rangle$ such that $V_0 = V + \{v_s, v_d\}$,
$E_0 = E - \{e \in E | d(e) \neq 0\} + \{v_s \xrightarrow{e} v \in V_r, u \in V_l \xrightarrow{e} v_d\}$
$\forall\ u \in V_0, \mathrm{temp}_{\mathsf{mrt}}(v_s, u) = 0, T_{v_s} = T_{v_d} = 0, Queue = v_s$
while $Queue \neq \emptyset$ do
    $get(u, Queue)$
    $\mathrm{temp}_{\mathsf{mrt}}(v_s, u) = \mathrm{temp}_{\mathsf{mrt}}(v_s, u) + T_u$
    foreach $u \xrightarrow{e} v$ do
      $indegree(v) = indegree(v) - 1$
      $\mathrm{temp}_{\mathsf{mrt}}(v_s, v) = \max(\mathrm{temp}_{\mathsf{mrt}}(v_s, u), \mathrm{temp}_{\mathsf{mrt}}(v_s, v))$
      if $indegree(v) = 0$ then $put(v, Queue)$ fi
    od
od
```

Lines 1 and 2 produce DAG G_0 from G which contains only edges $e \in E$, where $d(e) = 0$. Additional zero delay edges connect v_s to every root node $v \in V_r$ of G and the other set of these egdes connect every leaf node $u \in V_l$ of G to v_d. Line 3 initializes the $\mathrm{temp}_{\mathsf{mrt}}(v_s, u)$ 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 4–13 traverse the graph in topological order and compute the mrt of each node v with respect to v_s. The $\mathrm{temp}_{\mathsf{mrt}}$ for node v is originally set to zero. When the first parent of v is dequeued, v has its indegree reduced by one (Line 9) and also has its $\mathrm{temp}_{\mathsf{mrt}}$ updated (Line 10). 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 11). Node v will be eventually dequeued by Line 5. Line 6 will then add T_v to the $\mathrm{temp}_{\mathsf{mrt}}$ of node v producing the final mrt with respect to all paths reaching node v.

Note that the initial computation time are fixed point values and the probabilities associated with the computation time being greater than c are accumulated as one value in the algorithm. Only $\mathcal{O}(cn)$ values need to be stored for each vertex, where n is a constant depending on the number of fixed points. Therefore, the time complexity for calculating the summation (Line 6), or the maximum (Line 10) of two vertices is $\mathcal{O}((cn)^2)$. Since the algorithm computes the result in a breadth first fashion, the running time of Algorithm 3.1 is $\mathcal{O}((cn)^2|V||E|)$, while the space complexity is bounded by $\mathcal{O}(c|V|n)$.

Algorithm 3.2 presents the probabilistic retiming algorithm. 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 non-zero delay 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. Lines 21–31 traverse the DAG in a breath-first search manner and update the $\mathrm{temp}_{\mathsf{mrt}}$ for each node as in Algorithm 3.1. After updating a vertex, the resulting $\mathrm{temp}_{\mathsf{mrt}}$ is tested to see if the requirement, $\mathcal{P}(\mathrm{temp}_{\mathsf{mrt}}(G) > c) \leq \delta$, is met. Line 28, 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.

Intuitively, Line 28 pushes a delay onto all incoming edges of a node that violates the timing constraint. Since all descendents of this node will also be retimed, Line 28 in essence moves a delay 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 delay. The algorithm stops when there exists a retiming function satisfying the requirement (*modified_flag* $=$ **false**). That is the probability of the mrt from v_s to every other vertices, including v_d, being greater than c is less than δ. 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 $\mathcal{O}((cn)^2|V|^2|E|)$ while the space complexity remains the same as in the maximum reaching time algorithm.

Algorithm 3.2 (Probabilistic retiming)

Input: PG $G = \langle V, E, d, t\rangle$, a desired clock period c, and a probability δ.
Output: Retiming function r.

```
$\forall$ vertex $v \in V$, set $r(v) = 0$
for $i = 1$ to $|V|$ do
    $G_r = Retime(G, r)$; $modified\_flag$ = false
    $G_0 = \langle V_0, E_0, d, T\rangle$ where $V_0 = V + \{v_s, v_d\}$,
    $E_0 = E - \{e \in E | d(e) \neq 0\} + \{v_s \xrightarrow{e1} v \in V_r, u \in V_l \xrightarrow{e2} v_d\}$
    $\forall u \in V_0, \mathrm{temp}_{\mathsf{mrt}}(v_s, u) = 0, Queue = v_s, T_{v_s} = T_{v_d} = 0$
    while $Queue \neq \emptyset$ do
          $get(u, Queue)$
          $\mathrm{temp}_{\mathsf{mrt}}(v_s, u) = \mathrm{temp}_{\mathsf{mrt}}(v_s, u) + T_u$
          foreach $u \xrightarrow{e} v$ do
             $indegree(v) = indegree(v) - 1$
             $\mathrm{temp}_{\mathsf{mrt}}(v_s, v) = \max(\mathrm{temp}_{\mathsf{mrt}}(v_s, u), \mathrm{temp}_{\mathsf{mrt}}(v_s, v))$
             if $\mathcal{P}(\mathrm{temp}_{\mathsf{mrt}}(v_s, v) > c) > \delta$ and $u$ has not been retimed
               then $r(u) = r(u) - 1$; $modified\_flag$ = true; fi
             if $indegree(v) = 0$ then $put(v, Queue)$ fi
          od
    od
    if $modified\_flag$ = false then Report $r$, break fi
od
```

Notice that the selected c may not be either feasible for the algorithm to find the retiming values or tight enough. Therefore, one might need to increase or decrease the target clock period and rerun Algorithm 3.2 to get a feasible or tighter solution.

4 Scheduling under Resource Constraints

The retimed graph, obtained from the previous section, partially gives out some nice properties for scheduling the resulting graph to the systems. In other words, with enough number of processing elements, one can guarantee that the timing requirement (cycle period) will be met. By investigating such a graph, a schedule under resource constraints can be produced. Notice that the delays in the graph are reallocated. Therefore, it reduces the chance of having possible long paths. A DAG to be used in the scheduling process, can be obtained by ignoring edges with non-zero delays of the retimed graph, i.e., $E_{\mathrm{dag}} = \{e \in E : d(e) = 0\}$. The following algorithm presents a graph scheduling under resource constrained environment.

Algorithm 4.1 (Probabilistic Task Scheduling)

Input : PG $G = \langle V, E_{\mathrm{dag}}, d, T\rangle$, number of resources *numP*
Output: Schedule S

```
foreach $v \in V$ do
  if indegree($v$) = 0 then put($v$, Queue) fi od
for $i = 1$ to numP do count($i$) = 0 od
$V_m = \emptyset$; $E_m = \emptyset$
while Queue $\neq \emptyset$ do
      get($u$, Queue); $V_m = V_m \cup \{u\}$
      $E_m = E_m \cup \{v \xrightarrow{e} u : v \in V_m, e \in E_{\mathrm{dag}}\}$
      $G_{\mathrm{sav}} = G = \langle V_m, E_m, d, T \rangle$; $T_p = \infty$;
      for $i = 1$ to numP do
         if count($i$) $\neq 0$ then $E_m = E_m \cup \{node(count(i)) \xrightarrow{e} u\}$ fi
         $\Psi(G_m) = \mathrm{mrt}(G_m)$
         if mrt_less($\Psi(G_m), T_p$)
           then $G_{\mathrm{final}} = G_m$; Resource($u$) = $i$; $T_p = \Psi(G_m)$ fi
         $G_m = G_{\mathrm{sav}}$
      od
      foreach $u \xrightarrow{e} v$ do
        indegree($v$) = indegree($v$) − 1
        if indegree($v$) = 0 then put($v$, Queue) fi
      od
      count(Resource($u$)) = $u$
      $G_m = G_{\mathrm{final}}$
od
```

Algorithm 4.1 assigns nodes from the retimed graph to the existing resources. It operates on the graph in a *topological order* by attempting to map nodes in the same topological level to the target processors. Lines 40–41 construct a graph of scheduled nodes by inserting a tentative node u to V_m and adding an edge coming from the previous node $count(i)$ inside the same partition to the node u as well as preserving its dependency edges coming from already scheduled predecessors of the node u. $count(i)$ stores the last node scheduled in resource i. By doing that, we can compute the *probabilistic schedule length* of those scheduled nodes by using the concept of the mrt. Algorithm 4.1, now, can use the mrt to compare whether or not a node should be assigned to a processor. Lines 43–49 in Algorithm 4.1 present the processor selection part in which, in each iteration of the algorithm, it selects the resource which yields the *shortest* mrt among all possible resources. A function called *mrt_less* is introduced to handle this comparison. The following defines such a function.

Definition 4.1 *Given two random variables T_a and T_b with their probability distributions $p(x_i) = \mathcal{P}(T_a = x_i)$, where $x_i < x_{i+1} : i = 1 \ldots m$, and $p(y_i) = \mathcal{P}(T_b = y_i)$, where $y_i < y_{i+1} : i = 1 \ldots n$, $\underline{T_a \text{ is less than } T_b}$ under an acceptable probability δ if, for some integer α and β, $x_\alpha < y_\beta$ such that $(\sum_{i=\alpha+1}^{m} p(x_i) \leq \delta < \sum_{i=\alpha}^{m} p(x_i)) \bigwedge (\sum_{i=\beta+1}^{n} p(y_i) \leq \delta < \sum_{i=\beta}^{n} p(y_i))$, or, for $x_\alpha = y_\beta$ and $\sum_{i=\alpha}^{m} p(x_i) \leq \sum_{i=\beta}^{n} p(y_i)$.*

For example, if T_a has $\mathcal{P}(T_a = x_1 = 1) = 0.7, \mathcal{P}(T_a = x_2 = 2) = 0.18, \mathcal{P}(T_a = x_3 = 3) = 0.1$, and $\mathcal{P}(T_a = x_4 = 4) = 0.02$, and T_b has $\mathcal{P}(T_b = y_1 = 2) = 0.7, \mathcal{P}(T_b = y_2 = 4) = 0.15$, and $\mathcal{P}(T_b = y_3 = 5) = 0.1$. If $\delta = 0.2$, then $\sum_{i=3}^{4} \mathcal{P}(T_b = x_i) \leq \delta < \sum_{i=2}^{4} \mathcal{P}(T_b = x_i)$ and $\sum_{i=3}^{3} \mathcal{P}(T_b = y_i) \leq \delta < \sum_{i=2}^{4} \mathcal{P}(T_b = y_i)$, with $x_2 = 2$ and $y_2 = 4$. According to Definition 4.1, therefore, $T_a < T_b$.

Note that a schedule produced by Algorithm 4.1 is an execution order in which nodes in each partition or resource are ordered. This complies with the fact that each node in the schedule does not have an exact computation time, i.e., the notion of global clock tick or control step is not feasible. Therefore, an asynchronous model can be applied in this case.

5 Experimental Results

In this section we presents the experimental results obtained from using our algorithm to schedule some well-known benchmarks where the tasks are atomic operations that may present different execution time due to the fabrication process of a functional unit. The benchmarks which include the biquadratic IIR filter, 3-stage IIR filter, 4th-order Jaunmann wave digital filter, 5th-elliptic filter, all-pole lattice filter, and volterra filter are scheduled to two system configurations: 2 adders/1 multiplier and 2 adders/2 multipliers. The distributions of the execution time for the basic components (adder and multiplier) were obtained from [5]. Such a timing information consists of the minimum, typical and maximum values. For the experiment purpose, these values are broken down and generalized to fit in the normal probability distribution. In practice, probabilistic task scheduling algorithm can be applied to any one-dimensional loop body. The information about computation time of a task can simply be obtained by either using direct examination of the code or the use of profile information collected by the earlier runs of the program [13].

In each experiment, for a given confidence level θ and a PG, Algorithm 3.2 is used to search for the best clock period of the graph. In order to do this, the current desired clock period is varied based on whether or not a feasible solution is found. For instance, if the current desired clock period c is too small, the algorithm will report that no feasible solution exists. In this case, c will be increased and Algorithm 3.2 will be re-applied. Until a feasible c is found, this process is repeated. Table 1 shows the results obtained from running traditional retiming using worst-case computation time assumptions and the probabilistic model with varying confidence levels. In particular, Column worst in the table presents the optimized clock period obtained from applying traditional retiming using the worst-case computation time of each adder (24ns) and each multiplier (30ns). Columns 4–9 show the best clock period c for the confidence levels 0.9 down to 0.7 and percent reduction (%) over the clock periods of the benchmarks considering worst-case scenario.

			$\mathcal{P}(\mathrm{mrt}(v_s, v_d) \leq c) \geq 1 - \delta = \theta$					
	num.	c	$\theta = 0.9$		$\theta = 0.8$		$\theta = 0.7$	
Benchmark	nodes	worst	c	%	c	%	c	%
1. Biquad IIR	8	78	60	23	57	26	56	28
2. 3-stage IIR	12	54	44	19	41	24	40	26
3. WDF	17	156	116	26	112	28	109	30
4. Lattice	15	157	120	24	117	25	115	27
5. Volterra	27	276	216	22	212	23	208	25
6. Elliptic	34	330	240	28	236	29	233	30

Tab. 1. Probabilistic retiming versus worst case traditional retiming

Tables 2 and 3 compare the resulting schedule length obtained from running list scheduling algorithm on the original graph and the retimed graph using the information from Table 1 while considering the exact values (largest one) for the computation time and the probabilistic computation time. Assume that the synchronization cost, occurring when two dependent nodes are assigned to different functional units, is negligible. Column "worst" shows the schedule length of the benchmarks under the worst-case scenario. Column "Bef" presents the schedule length obtained by applying list scheduling for varying confidence level to those benchmarks.

Column "Aft" illustrates the optimized schedule length after probabilistic task scheduling is applied. In order to quantify the improvement of the probabilistic retiming algorithm, each of column "%" in the tables lists the percent reduction of the schedule length reduction compared with the schedule length obtained without applying probabilistic retiming.

		Schedule length given θ, (2 adds. 1 mult.)								
		$\theta = 0.9$			$\theta = 0.8$			$\theta = 0.7$		
Ben	wst	Bef	Aft	%	Bef	Aft	%	Bef	Aft	%
1.	144	125	91	27	121	90	26	118	102	26
2.	204	188	151	20	184	147	20	181	143	21
3.	186	157	142	10	154	138	10	151	136	10
4.	312	229	176	23	225	172	24	222	169	24
5.	750	526	509	3	519	502	3	514	497	3
6.	414	318	298	6	314	294	6	310	290	6

Tab. 2. Scheduling to the systems with 2 adders

We have also tested some benchmarks by running the traditional retiming algorithm using the average case computation time, although these results are not shown here. Most of the result retimed graphs induce the longer schedule length than using our approach. For example, in the biquadratic IIR filter, with $\theta = 0.7$, its schedule length is 105 while considering the probability during the retiming results in the schedule with length 90 in the 2-adder and 1-multiplier system. Likewise, in the 5^{th}-elliptic filter, the schedule length considering the average-case value is 304 with $\theta = 0.9$ while our approach gives the schedule with the length 298 for the same system.

By using our method, the obtained schedule length l is ensured with a confidence level greater than 90%. For example, in the system with two adders and one multiplier, after running the probabilistic task scheduling algorithm on the Biquad IIR filter, the obtained result guarantees that, if the confidence level is 0.1, the schedule length will be less than 91, which is 27% shorter than the result obtained from considering the worst case scenario. With such a high confidence probability, the results show that, using our approach, the schedule length of these benchmarks can be much less than the schedule length obtained when considering worst-case scenario.

		Schedule length given θ, (2 adds. 2 mults.)								
		$\theta = 0.9$			$\theta = 0.8$			$\theta = 0.7$		
Ben	wst	Bef	Aft	%	Bef	Aft	%	Bef	Aft	%
1.	108	85	66	22	83	62	25	80	63	21
2.	162	124	87	30	120	82	32	118	82	31
3.	180	137	135	1	133	132	1	131	130	1
4.	312	229	153	33	225	149	34	222	143	36
5.	510	359	342	5	353	336	5	349	332	5
6.	372	288	272	6	284	268	6	282	265	6

Tab. 3. Scheduling to the system with 2 adders and 2 multipliers

6 Conclusion

In some applications in high-level synthesis area, a task may have uncertain computation time. Such tasks normally occur due to hardware manufacturing process or the estimation during system design process. A probability distribution can be used to mirror these uncertainties. We have presented the theoretical foundation and experimental results for a new scheduling framework, called probabilistic task scheduling, which reduces the chance that the system will take a long time to finish computing all iterations. Considering the realistic case, our algorithm can effectively schedule those tasks while utilizing the task probability information and the designer confidence probability θ. Based on the framework presented in the paper, it is likely that the probabilistic optimization algorithms may have a great impact to various problems in high-level synthesis.

References

[1] U. Banerjee. Unimodular transformations of double loops. In *Proc. of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pages 192–219, 1990.

[2] L. Chao, A. LaPaugh, and E. Sha. Rotation scheduling: A loop pipelining algorithm. In *Proc. of the 30th DAC*, pages 566–572.

[3] L.-F. Chao and E. H.-M. Sha. Static scheduling for synthesis of DSP algorithms on various models. *J. of VLSI Signal Processing*, pages 207–223, October 1995.

[4] E. M. Girczyc. Loop winding—a data flow approach to functional pipeline. In *Proc. of the ISCAS*, pages 382–385, May 1987.

[5] Texas Instruments. *The TTL data book*, volume 2. Texas Instruments Incorporation, 1985.

[6] R. A. Kamin, G. B. Adams, and P. K. Dubey. Dynamic list-scheduling with finite resources. In *Proc. of the 1994 ICCD*, pages 140–144.

[7] I. Karkowski and R. H. J. M. Otten. Retiming synchronous circuitry with imprecise delays. In *Proc. of the 32nd DAC*, pages 322–326.

[8] A. A. Khan, C. L. McCreary, and M. S. Jones. A comparison of multiprocessor scheduling heuristics. In *Proc. of the 1994 ICPP*, pages 243–250.

[9] M. Lam. Software pipelining. In *Proc. of the ACM SIGPLAN'88 Conf. on Programming Language Design and Implementation*, pages 318–328.

[10] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[11] 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.

[12] P. Mozumder and A. Strojwas. Statistical control for VLSI fabrication processes. In W. Moore, W. Maly, and A. Strojwas, editors, *Yield Modeling and Defect Tolerance in VLSI*. Adam Hilger, Bristol and Philadelphia, 1987.

[13] D. A. Patterson and J. L. Hennessy. *Computer architecture: A Quantitative approach*. Morgan-Kaufman, 1996.

[14] M. E. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.

[15] M. E. Wolfe and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[16] L. A. Zadeh. Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets and Systems*, 1:3–28, 1978.