

# Efficient Scheduling for Design Exploration with Imprecise Latency and Register Constraints

C. Chantrapornchai<sup>†</sup>      W. Surakumpolthorn      E. H-M. Sha<sup>‡</sup>  
Dept. of Mathematics      Dept. of Electronics      Dept. of Computer Science  
Silpakorn University, Thailand      KMITL, Thailand      U. of Texas at Dallas, U.S.A.

## Abstract

In architectural synthesis, scheduling and resource allocation are important steps. During the early stage of the design, imprecise information is unavoidable. Under the imprecise system characteristics and constraints, this paper proposes a polynomial-time scheduling algorithm which minimizes both functional units and registers while scheduling. The algorithm can be used in design exploration for exploring the tradeoff between latency and register counts and selecting a solution with satisfactory performance and cost. The experiments show that we can achieve a schedule with the same acceptable degree while saving register upto 37% compared to the traditional algorithm.

**Keywords:** Scheduling/Allocation, Multiple design attributes, Imprecise information, Register constraint, Inclusion Scheduling

## Corresponding author:

Chantana Chantrapornchai  
Faculty of Science, Silpakorn University, Nakorn Pathom, Thailand 73000

ctana@su.ac.th

---

<sup>†</sup>This work was supported in part by the TRF under grant number MRG4680115, Thailand.

<sup>‡</sup>This work was supported in part by TI University Program, NSF EIA 0103709, Texas ARP-009741-0028-2001 and NSF CCR-0309461, USA.

# 1 Introduction

In architectural level synthesis, imprecise information is almost unavoidable. For instance, an implementation of a particular component in a design may not be known due to several reasons. There may be various choices of modules implementing the functions or the component may have not been completely designed down to the geometry level. Even if it has been designed, variation in fabrication process will likely induce varying area and time measurements. Another kind of impreciseness or vagueness arises from the way a design is considered to be acceptable at architecture level. If a design with latency of 50 cycles is acceptable, what about a design with 51 cycles versus a design with 75 cycles? This even becomes imprecise especially when there are multiple conflicting design criteria. For example, is it worth to expand a latency by two cycles while saving one register and what about expanding 10 more cycles? Effective treatment of such impreciseness in high level synthesis can undoubtedly play a key role in finding optimal design solutions.

In this paper, we present an approach to handle certain imprecise specification and use them during architectural synthesis. The system characteristics are modeled based on the fuzzy set theory. Register count is considered as another dimension of imprecise system requirement. We extend the work in [2] to create a schedule subject to register constraints. An input system is modeled using a data flow graph with imprecise timing parameters. The imprecise schedule which minimizes the register usage is generated. The algorithm can be integrated into design exploration framework which considers the tradeoff between latency and register usage to find an acceptable solution. Such systems can be found in many digital signal processing applications, e.g., communication switches and real-time multimedia rendering systems. Imprecise specification on both system parameters and constraints can have a significant impact on component resource allocation and scheduling for designing these systems. Therefore, it is important to develop synthesis and optimization techniques which incorporate such impreciseness.

Most traditional synthesis tools ignore these vagueness or impreciseness in the specification. In particular, they assume the worst case (or sometimes typical case) execution time of a functional unit. The constraints are usually assumed to be a fixed precise value although in reality some flexibility can be allowed in the constraint due to the individual interpretation of an “acceptable” design. Such assumptions can be misleading, and may result in a longer design process and/or overly expensive design solutions. By properly considering the impreciseness up front in the design process, a good initial design solution can be achieved with provable degree of acceptance. Such a design solution can be used effectively in the iterative process of design refinement, and thus, the number of redesign cycles can be reduced.

Random variables with probability distributions may be used to model such uncertainty. Nevertheless, collecting the probability data is sometimes difficult and time consuming. Furthermore, some imprecise information may not be correctly captured by the probabilistic model. For example, certain inconspicuousness in the design goal/constraint specification, such as the willingness of the user to accept certain designs or the confidence of the engineer towards certain designs, cannot be described by probabilistic distribution.

Another effective technique for handling imprecise information is based on fuzzy set theory [32]. Instead of going through the statistical simulation process to collect the probability data, imprecise data can be easily modeled by fuzzy sets. For example, when the typical, minimum and maximum execution time values are given, they can easily be generalized into a triangular fuzzy number. (In some design tools, typical, minimum and maximum execution time values are already provided, or designers may quickly estimate the three values.) Similarly, the design goal/constraint

can be modeled by a fuzzy set, e.g., the latency constraint can be relaxed because of an ambiguous understanding of design solution acceptability. Therefore, applying the idea of fuzzy numbers and fuzzy arithmetics seems promising.

Many researchers have applied the fuzzy logic approach to various kinds of scheduling problem. In compiler optimization, fuzzy set theory has been used to represent unpredictable real-time events and imprecise knowledge about variables [16]. Lee et.al. applied the fuzzy inference technique to find a feasible real-time schedule where each task satisfies its deadline under resource constraints [21]. In production management area, fuzzy rules were applied to job shop and shop floor scheduling [30, 25]. Kaviani and Vranesic used fuzzy rules to determine the appropriate number of processors for a given set of tasks and deadlines for real-time systems [20]. Soma et.al. considered the schedule optimization based on fuzzy inference engine [29]. These approaches, however, do not take into account the fact that an execution delay of each job can be imprecise and/or multiple attributes of a schedule.

Many research results are available for design space exploration [13, 1, 24, 7]. All of these works differ in the techniques used to generate a design solution as well as the solution justification. These works, however, do not consider the impreciseness in the system attributes such as latency constraints and the execution time of a functional unit. Recently, Karkowski and Otten introduced a model to handle the imprecise propagation delay of events [18, 17]. In their approach, the fuzzy set theory was employed to model imprecise computation time. Their approach applies possibilistic programming based on the integer linear programming (ILP) formulation to simultaneously schedule and select a functional unit allocation under fuzzy area and time constraints. Nevertheless, the complexity of solving the ILP problem with fuzzy constraints and coefficients can be very high. Furthermore, they do not consider multiple degrees in acceptability of design solutions. Several papers were published on the resource estimation [28, 26, 8]. These approaches, however, neither consider multiple design attributes nor impreciseness in system characteristics.

Many research works related to register allocation exists in high-level synthesis and compiler optimization area for VLIW architecture. For example, Chen et. al. proposed a loop scheduling for timing and memory operation optimization under register constraint [14]. The technique is based on multi-dimensional retiming. Eichenberger et. al. presented an approach for register allocation for VLIW and superscalar code via stage scheduling [11, 12]. Dani et. al. also presented a heuristic which uses stage scheduling to minimize register requirement. They also target at instruction level scheduling [9]. Zalamea et. al. presented hardware and software approach to minimize the register's usage targeting VLIW architecture [22, 23, 34]. On the software side, they proposed an extended version of modulo scheduling which considers register constraint, and register spilling. However, these work focus on loop scheduling and do not consider handling the imprecise system characteristics or specification.

In [2], the inclusion scheduling which takes the imprecise system characteristic was proposed. The word "inclusion" stems from the fact that it collects various information while performing scheduling. The algorithm was expanded and used in design exploration under imprecise system requirement as well as the estimation of resource bounds [5, 4, 6]. However, it does not take register criteria in creating a schedule.

In this paper, we particularly consider both latency and register constraints. We propose an extended inclusion scheduling which considers the register usage while performing scheduling. The developed scheduling core, RCIS, *Register-Constrained Inclusion Scheduling*, takes imprecise information into account. Since the latency of the system specification is imprecise, the register usage of the schedule is imprecise. We study the imprecise register usage and propose a heuristic to estimate the register count in the imprecise schedule. Given a functional specification (in the form of a directed acyclic graph) and a number of available functional units, an inclusion schedule can be efficiently generated in polynomial time. Our proposed approach can efficiently be used in an iterative design cycle to find an

initial design to reduce the number cycles of design improvements. Experimental results show that, we can achieve a better design when the number of registers is limited while keeping the same satisfactory requirement.

This paper is organized as follows: Section 2 describes our models. It also presents some backgrounds in fuzzy set. Section 3 presents the iterative design framework which may integrate our scheduling approach. Section 4 presents the inclusion scheduling framework. It also addresses some issues when register counts are calculated during scheduling. Section 5 presents necessary definitions, properties, and heuristics which integrate register consideration into inclusion scheduling framework. Section 6 shows how the algorithm works and displays some experimental results. Finally, Section 7 draws a conclusion from our work.

## 2 Overview and Models

In this section, we first describe our model as well as problem description. Since in developing an inclusion schedule some fuzzy arithmetics is involved, we also review some basic concepts in fuzzy computation.

### 2.1 Model Descriptions

Operations and their dependencies in an application are modeled by a vertex-weighted directed acyclic graph, called a *Data Flow Graph*,  $G = (\mathcal{V}, \mathcal{E}, \beta)$ , where each vertex in the vertex set  $\mathcal{V}$  corresponds to an operation and  $\mathcal{E}$  is the set of edges representing data flow between two vertices. Function  $\beta$  defines the type of operation for node  $v \in \mathcal{V}$ . Figure 1

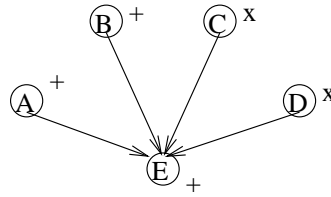


Figure 1: Test1: Data flow graph example

shows a five-node data flow graph, where  $\mathcal{V} = \{A, B, C, D, E\}$ ,  $\mathcal{E} = \{A \rightarrow E, B \rightarrow E, C \rightarrow E, D \rightarrow E\}$ , ( $u \rightarrow v$  defines a directed edge from  $u$  to  $v$ ),  $\beta(A) = \beta(B) = \beta(E) = \text{add}$ , and  $\beta(C) = \beta(D) = \text{multiply}$ .

Operations in a data flow graph can be mapped to different functional units which in turn can have varying characteristics. Such a system must also satisfy certain design constraints, for instance, power and cost limitations. These specifications are characterized by a tuple  $S = (\mathcal{F}, \mathcal{A}, \mathcal{M}, \mathcal{Q})$ , where  $\mathcal{F}$  is the set of functional unit types available in the system, e.g.,  $\{\text{add}, \text{mul}\}$ .  $\mathcal{A}$  is  $\{A_f : \forall f \in \mathcal{F}\}$ . Each  $A_f$  is a set of tuples  $(a_1, \dots, a_k)$ , where  $a_1$  to  $a_k$  represent attributes of particular  $f$ . In this paper, we use only use latency as an example attribute. (Note that our approach is readily applicable to include other constraints such as power and area). Hence,  $A_f = \{x : \forall x\}$  where  $x$  refers to the latency attribute of  $f$ .  $\mathcal{M}$  is  $\{\mu_f : \forall f \in \mathcal{F}\}$  where  $\mu_f$  is a mapping from  $A_f$  to a set of real number in  $[0,1]$ , representing a possible degree of using the value. Finally,  $\mathcal{Q}$  is a function that defines the degree of a system being acceptable for different system attributes. If  $\mathcal{Q}(a_1, \dots, a_k) = 0$  the corresponding design is totally unacceptable while  $\mathcal{Q}(a_1, \dots, a_k) = 1$ , the corresponding design is definitely acceptable.

Using a function  $Q$  to define the acceptability of a system is a very powerful model. It can not only define certain constraints but also express certain design goals. For example, one is interested in designing a system with latency under 500 and register count being less than 6 respectively. Also, the smaller latency and register count, the better a system is. The best system would have both latency and register count being less than or equal to 100 and 1 respectively. An acceptability function,  $Q(a_1, a_2)$  for such a specification is formally defined as:

$$Q(a_1, a_2) = \begin{cases} 0 & \text{if } a_1 > 500 \text{ or } a_2 > 6 \\ 1 & \text{if } a_1 \leq 100 \text{ and } a_2 \leq 1 \\ F(a_1, a_2) & \text{otherwise,} \end{cases} \quad (1)$$

where  $F$  is assumed to be linear functions, e.g.,  $F(a_1, a_2) = 1.249689(a_1 + 2a_2) - 0.001242$  which returns the acceptability between  $(0, 1)$ .

Figure 2 illustrates Equation (1) graphically. In this constraint, we express the weighted sum of the two criteria which gives the preference to minimizing register count twice as much as minimizing latency, that is latency: register count is 1:2. In other words, we are willing to spend two more latency cycles if one register can be saved.

Figure 3 shows the projection of Equation (1) on latency and possibility axis.

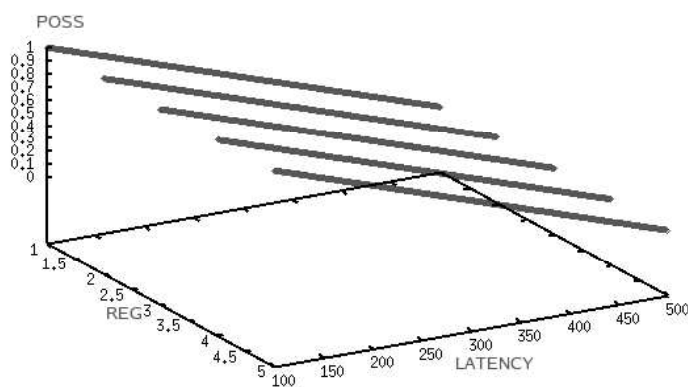


Figure 2: Imprecise constraint where Latency : Register is 1 : 2.

In general, one may model any criteria by

$$Q(a_1, a_2, \dots, a_n) = \begin{cases} 0 & \text{if } a_1 > p_{1_{max}} \dots \text{ or } a_n > p_{n_{max}} \\ 1 & \text{if } a_1 \leq p_{1_{min}} \dots \text{ and } a_n \leq p_{n_{min}} \\ F(w_1 a_1 + w_2 a_2 + \dots w_n a_n) & \text{otherwise.} \end{cases} \quad (2)$$

where

For example, we can simply replace the register constraint by others such as power. Figure 4(b) depicts an example of the specification concerning the tradeoff graphically where  $w_1 = 2, w_2 = 1$ . Hence,  $F$  refers to a z-shaped curve function which produces a smooth transition between two given points. Figure 4(c) shows the projection of the 3-dimensional acceptability model to the latency and acceptability plane. In this figure, each z curve represents a projection of  $Q$  function to a latency-acceptability plane. An inner curve (tighter latency constraint) corresponds to

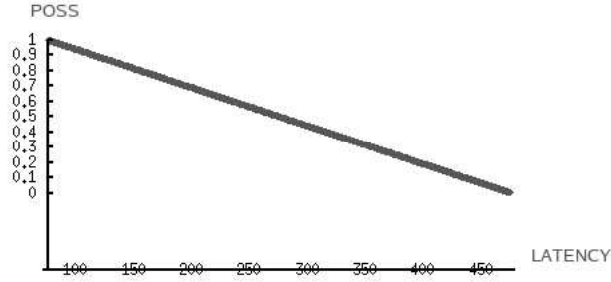


Figure 3: Projection of constraint in Figure 2.

larger power values. Based on the acceptability model, a design with high acceptability implies an optimized design towards certain goals.

Based on the above model, the combined scheduling/binding we intend to solve can be formulated as follows:

Given a specification containing  $S = (\mathcal{F}, \mathcal{A}, \mathcal{M}, \mathcal{Q})$ ,  $G = (\mathcal{V}, \mathcal{E}, \beta)$ , and acceptability level  $\alpha$ , find a schedule under functional unit and register constraints for each  $f$  in  $\mathcal{F}$  whose the acceptability degree is greater than or equal to  $\alpha$  subject  $Q$ .

## 2.2 Fuzzy Sets

In this section, we give a quick review of fuzzy set theory as it relates to our work. Readers familiar with the theory can skip this part.

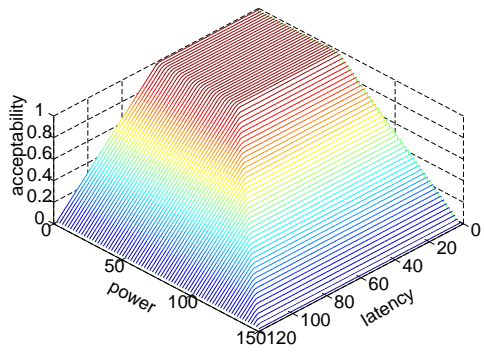
Fuzzy sets, proposed by Zadeh, represent a set with imprecise boundary [31, 33]. In classical (crisp) sets, an element can either be a member of a set or not at all; hence, its membership degree is either 1 or 0. A fuzzy set is defined by assigning each element in a universe of discourse its membership degree in the unit interval  $[0, 1]$ , conveying to what degree  $x$  is a member in the set. This membership value can be defined as a membership function of an element in the set,  $\mu(x) : x \rightarrow [0, 1]$ .

A fuzzy set is said to be *normal* if there exists at least one member in the set whose membership value is unity. A *convex* fuzzy set is defined as: for any  $x, y$ , and  $z$  in the fuzzy set  $A$ , the relation  $x < y < z$  implies that  $\mu_A(y) \geq \min(\mu_A(x), \mu_A(z))$ . A fuzzy number is a normal, convex fuzzy set defined on the real line  $\mathbb{R}$ . Let  $A$  and  $B$  be fuzzy numbers with membership functions  $\mu_A(x)$  and  $\mu_B(y)$ , respectively. Let  $*$  be a set of binary operations  $\{+, -, \times, \div, \min, \max\}$ . The arithmetic operations between two fuzzy numbers, defined on  $A * B$  with membership function  $\mu_{A*B}(z)$ , can use the extension principle, by [15]:

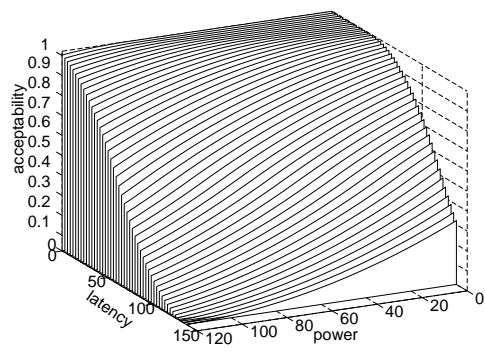
$$\mu_{A*B}(z) = \bigvee_{z=x*y} (\mu_A(x) \wedge \mu_B(y)) \quad (3)$$

where  $\bigvee$  and  $\wedge$  denote max and min operations respectively.

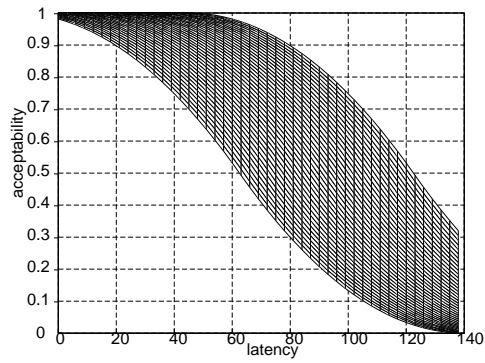
Fuzzy arithmetic is used to compute an arithmetic operation between two fuzzy numbers. Figure 5(a) shows a fuzzy number  $A$ , which is assumed to be normal triangular-shaped lied on an real line. In this figure, let  $A$  be assigned with the confidence interval  $(2, 6)$ . The most possible value of  $A$  is 4 since its *confidence level* or *presumption level*



(a) Linear acceptability



(b) z curve acceptability with trade-off



(c) Latency acceptability curves corresponding to different power constraints derived from Figure 4(b)

Figure 4: Various kinds of acceptability functions

is 1. Similarly, Figure 5(b) shows the fuzzy number with the confidence interval (3, 7) representing B. Figure 5(c)

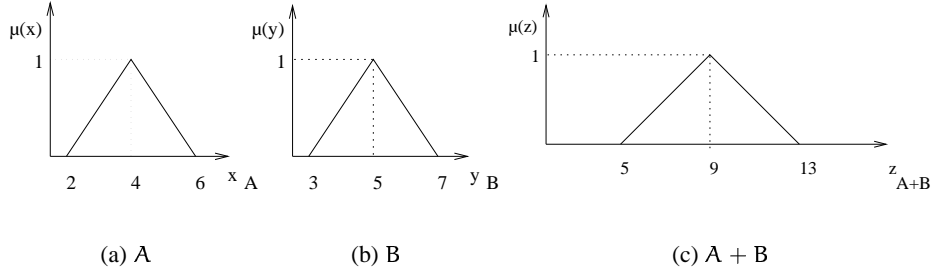


Figure 5: Adding two fuzzy numbers, A and B

demonstrates a graphical result of adding two fuzzy numbers defined on the integer line from Figures 5(a)–5(b), using Equation (3).

In order to compare two fuzzy numbers, several methods can be used. All of these methods are based on selecting a representative for each fuzzy number and compare the representatives [19]. One way to obtain the representatives is using the *removal* with respect to  $k$ , which is a measure of distance from  $k$ , computed by  $R(A, k) = \frac{1}{2}(R_l(A, k) + R_r(A, k))$ , where  $A$  is a fuzzy number,  $k$  is a reference position on the  $x$ -axis,  $R_l$  is the area bounded by the left side of the curve and the line  $x = k$  and similarly for the right side,  $R_r$ . Another can be *mode*, which uses the value  $x$  such that  $\mu(x) = \max_i\{\mu(x_i)\}$  for all  $x_i$  in the fuzzy set. *Divergence* is another way to calculate the representatives. It represents the width of the set which is computed by  $x_{max} - x_{min}$ . In addition, the defuzzified value can be used to represent the fuzzy set. Several defuzzified methods can be found in [27].

Based on the fuzzy set concept, we model the relationship between functional units and possible characteristics such that each functional unit is associated with a fuzzy set of characteristics. Given a functional unit  $f$  and its possible characteristic set  $A_f$  let  $\mu_f(a) \in [0, 1], \forall a \in A_f$ , describe a possibility of having attribute  $a$  for a functional unit  $f$ . Let us focus on the timing attribute. A fuzzy set of timing characteristic of a functional unit  $f$  may be  $\{ \frac{10}{.2}, \frac{20}{.4}, \frac{35}{1}, \frac{70}{.7} \}$ . That is  $f$  may use 10, 20, 35, and 70 time units to execute with different possibility, i.e.,  $\mu_f(10) = .2, \mu_f(20) = .4, \mu_f(35) = 1, \mu_f(70) = .7$ .

### 3 Iterative Design Process

Figure 6 presents an overview of our iterative design process for finding a satisfactory solution. One may estimate the initial design configuration with any heuristic. The scheduling and allocation process produces the imprecise schedule attributes which are used to determine whether or not the design configuration is acceptable. The dashed block elements contain the scheduling core which attempts to minimize both latency and register usage. Our scheduling and allocation process incorporates varying information of each operation. It takes an application modeled by a directed acyclic graph as well as the number of functional units that can be used to compute this application. Then, the schedule of the application is derived. This schedule shows an execution order of operations in the application based on the available functional units. The total attributes of the application can be derived after the schedule is computed. The

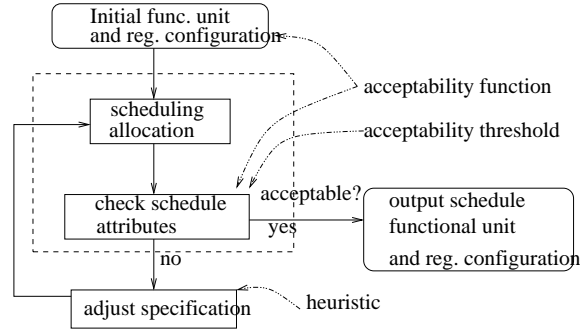


Figure 6: Iterative design solution finding process

given acceptability function is then checked with the derived attributes of the schedule. In order to determine whether or not the resource configuration is satisfied the objective function, we use the *acceptability threshold*. If the schedule attributes lead to the acceptability level being greater than the threshold, the process stops. Otherwise, the resource configuration is updated and this process is repeated until the design solution cannot be improved or the design solution is found.

## 4 Inclusion Scheduling Framework

In this section, we present the inclusion scheduling framework. Then we propose a framework to evaluate the quality of the schedule.

### 4.1 Latency-based Algorithm

Specifically, inclusion scheduling is a scheduling method which takes into consideration of fuzzy characteristics which in this case is fuzzy set of varying latency values associated with each functional unit. The output schedule, in turn, also consists of fuzzy attributes. In a nutshell, inclusion scheduling simply replaces the computation of accumulated execution times in a traditional scheduling algorithm by the fuzzy arithmetic-based computation. Hence, fuzzy arithmetics (specifically Equation (3)) is used to compute possible latency from the given functional specification. Then, using a fuzzy scheme, latency of different schedules are compared to select a functional unit for scheduling an operation. Though the concept is simple, the results are very informative. They can be used in many ways such as module selection [3]. Algorithm 4.1 presents a list-based inclusion scheduling framework.

#### **Algorithm 4.1 (Inclusion scheduling)**

**Input:**  $G = (\mathcal{V}, \mathcal{E}, \beta)$ ,  $Spec = (F, \mathcal{A}, \mathcal{M}, \mathcal{Q})$ , and  $N = \#FUs$

**Output:** A schedule  $S$ , with imprecise latency

- 1  $Q =$  vertices in  $G$  with no incoming edges // finding root nodes
- 2 **while**  $Q \neq \text{empty}$  **do**

```

3   Q = prioritized(Q)
4   u = dequeue(Q); mark u scheduled
5   good_S = NULL;
6   foreach f ∈ {fj : where fj is able to perform β(u), 1 ≤ j ≤ N} do
7       temp_S = assign_heuristic(S, u, f) // assign u at FU f
8       if Eval_Schedule(good_S, temp_S, G, Spec)
9           then good_S = temp_S fi od
10      S = good_S // keep good schedule
11      foreach v : (u, v) ∈ E do
12          indegree(v) = indegree(v) - 1
13          if indegree(v) = 0 then enqueue(Q, v) fi od
14      od
15      return(S)

```

In Algorithm 4.1, fuzzy arithmetic takes place in routine *Eval\_Schedule* (Line 8). In this routine, the fuzzy latency of the intermediate schedule, after node  $u$  is assigned to  $f$ , is computed. It also compares the current schedule with the “best” one found in previous iterations. The better one of the two is then chosen and the process is repeated for all nodes in the graph.

**Algorithm 4.2 (Eval\_Schedule)**

**Input:** schedules  $S_1, S_2$ ,  $G = (\mathcal{V}, \mathcal{E}, \beta)$ , and  $Spec = (F, \mathcal{A}, \mathcal{M}, \mathcal{Q})$

**Output:** 1 if  $S_1$  is better than  $S_2$ , 0 otherwise.

```

1 // construct a modified graph
2  $G_0 = (\mathcal{V}_0, \mathcal{E}_0, \beta)$  where  $\mathcal{V}_0 = \mathcal{V} - \{\text{unscheduled nodes}\}$ ,  $\mathcal{E}_0 = \emptyset$ 
3 foreach schedule  $S_i = S_1$  to  $S_2$  do
4      $\mathcal{E}_0 = \{(u, v) : u, v \in \mathcal{V}_0, \text{ if } u, v \text{ in same f.u. in } S_i$ 
5         and  $v$  is immediately after  $u\}$ 
6     Sort graph  $G$  according to the topological order
7     foreach level  $i$  of graph  $G$  in topological order do
8         //  $\gamma(u)$  returns the functional unit binding for  $u$ 
9          $u.attr = \text{fuzzyadd\_time}(u.attr, attr(\gamma(u)))$ 
10        foreach  $u : v \rightarrow u, \forall v \in V_i$  do
11             $u.attr = \text{fuzzymax\_time}(v.attr, u.attr)$ 
12        od
13    od
14    Let  $W$  is a set of leaves in  $G_0$ 
15    // merge all values in  $quality[S_i]$ 
16     $quality[S_i] = \text{fuzzymax\_time}(W)$ 
17    od
18 // comparing the overall attributes of both schedules
19 return(compare(quality[S1], quality[S2]))

```

The implementation of *Eval\_Schedule* is shown in Algorithm 4.2. It adds edges between two consecutive nodes in the same functional unit to construct a legal execution order of all nodes. Assuming the latency values are discretized,

using Equation (3) by replacing  $z$  with execution time, we compute the degree of each latency value. Lines 11 and 9 replace addition and maximum operations with fuzzy operations. Note that in Line 9, Line 11 and Line 19 can be modified to consider the other criteria.

As example, consider a DFG where  $\mathcal{V} = \{A, B, C\}$ ,  $\mathcal{E} = \{A \rightarrow C\}$ . Let us assume that  $\beta(A) = \beta(B) = \beta(C) = \text{add}$ . There exists  $\mathcal{F} = \{FU_1, FU_2\}$ , two general functional units.  $FU_1$  has latency  $\{(5, 0.05), (10, 0.7), (20, 0.8), (30, 1)\}$  and  $FU_2$  has latency  $\{(5, 0.05), (15, 0.5), (22, 0.9), (35, 1)\}$ . After performing scheduling,  $A$  is allocated to  $FU_1$ . Then  $B$  is allocated to  $FU_2$  since allocating  $B$  to  $FU_1$  would result in a schedule with worse overall latency. After that, allocating  $C$  to  $FU_1$  yields the overall weighted length 47.9 and allocating  $C$  to  $FU_2$  yields the weighted length 52.5. Hence,  $C$  is also allocated to  $FU_1$ . Note that this example considers only attempting to minimize latency constraint.

## 4.2 Issues in Considering Registers

From the previous algorithm, to compute fuzzy latency, it uses the original data flow graph and adds extra edges which connect consecutive nodes in the same function unit. Then a dummy sink node is created and connected to all leaves in the graph. The algorithm traverses the graph while calculating the fuzzy length of the schedule.

According to the scheme, we can see that the schedule table is not explicitly created. Thus, the notion of control step is implicit. This raises a few aspects. First, the nodes are assumed to start as early as possible. If the register constraint is considered, the ASAP approach may not give a good result. That is it may be good to start node later while latency is kept the same or even a little larger to maximize acceptability. Second, the graph is directly used to calculate fuzzy maximum latency. When considering only overall fuzzy latency, it is not important that nodes start ASAP or not, since in overall, the fuzzy length remains the same. However, when taking registers into account, the start time of a node becomes important since it can affect the register usage. Different start time can imply different register usage at each time step. As a result, a scheduling heuristic must be modified to consider register usage at each time step. Third, in order to consider this, since a node's execution time is a fuzzy number, the start time of the node and its successors become a fuzzy number. When the start time of the node is a fuzzy number, the finished time of the node is a fuzzy number. We need to define the fuzzy life time of a node. Hence at each control step, a node may occupy the functional unit with some possibility. To minimize the number of registers in this way, we must also minimize the possibility of using certain number of registers at each time step as well. In effect, it becomes not a trivial modification of inclusion scheduling to consider a register constraint.

In this paper, we establish a notion of fuzzy start time, fuzzy finished time, and fuzzy life time. We then propose an algorithm to calculate fuzzy register usage for a schedule. Next both register usage and latency characteristics of the schedule are compared to the system constraint and the best schedule is chosen. We show that comparing to traditional scheduling, we find a better solution with respect to the given system constraint, when both register and latency are taken into account simultaneously while performing scheduling.

## 5 Register Count Consideration under Impreciseness

In order to consider register constraint, we should count the number of registers used at each time step. In specific, when we place a node on a schedule, we consider a life time of the node in the schedule. Traditionally, a life time of a

node depends on the location of the node's successors in the schedule. That is the value produced by the node must be held until its successors have consumed it. For simplicity, let the successors consume the value right after they start.

Recall that a node's execution time is a fuzzy set, where the membership function is defined by  $\mu(x) = y$ . It implies that the node will take  $x$  time units with possibility  $y$ . Consequently, a start time and finished time of a node are fuzzy numbers. To be able to calculate fuzzy start time and finished time, we must assume that all nodes have been assigned to functional units already. We assume that resource binding and order of nodes executing in these resources are given based on the modified DFG (i.e., *scheduled DFG*). The modified DFG is just the original DFG where extra edges due to independent nodes executing in the same functional units are inserted (as constructed in Algorithm 5.2).

### 5.1 Fuzzy Timing Attributes in a Schedule

In the following, we present basic terminologies used in the algorithm which calculates register usage under imprecision. The algorithm based on the modified DFG is discussed in the next subsection.

**Definition 5.1** For  $G = (\mathcal{V}, \mathcal{E}, \beta)$ , and a given schedule, a fuzzy start time of node  $u \in \mathcal{V}$ ,  $FST(u)$  is a fuzzy set whose membership degree is defined by  $\mu_{FST(u)}(x) = y$ , i.e., node  $u$  may start at time step  $x$  with possibility  $y$ .

For nodes that are executed at time step 0 in each functional unit,  $FST(u) = 0$ , which is a crisp value.

**Definition 5.2** For  $G = (\mathcal{V}, \mathcal{E}, \beta)$ , and a given schedule, a fuzzy finished time of node  $u \in \mathcal{V}$ ,  $FFT(u)$  is a fuzzy set whose membership degree is defined by  $\mu_{FFT(u)}(x) = y$ , i.e., node  $u$  may finish at time step  $x$  with possibility  $y$ .

Hence,  $FFT(v) = FST(v) + EXEC(v)$ , where  $EXEC(v)$  is the fuzzy latency of  $v$ . When considering earliest start time of a node,  $FST(v) = \max_i(\tilde{FFT}(u_i)) + 1, \forall u_i \rightarrow v$ .

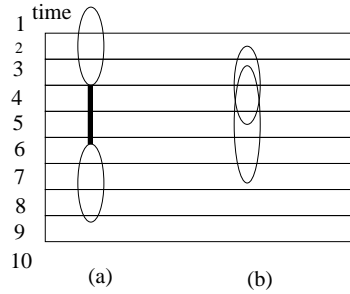


Figure 7: A view of fuzzy start time and finished time

The general idea of using fuzzy numbers is depicted in Figure 7 for both start time and finished time. Circles denote the fuzzy boundary which means that the start time and finished time boundary of a node is unclear. Indeed, they may also be overlapped as shown in Figure 7(b). When a node occupies a resource at a certain time step, a possibility value is associated with the assignment.

Traditionally, when the timing attribute is a crisp value, the start time  $x$  and finished time  $y$  of a node form an integer interval  $[x\dots y]$ , which will be used to compute the register usage in the schedule. In our case, a fuzzy life time for node  $u$  contains two fuzzy sets:  $FST(u)$  and  $MFFT(u)$ , the maximum of start time of all its successors.

**Definition 5.3** For  $G = (\mathcal{V}, \mathcal{E}, \beta)$ , and a given schedule, fuzzy life time of node  $u$ ,  $FLT(u)$  is a pair of  $[FST(u), MFFT(u)]$ , where  $\mu_{MFFT(u)} = FFT(u) \underset{\sim}{+} \underset{\sim}{\max}(FST(v_i))$ , where  $u \rightarrow v_i \in \mathcal{E}$  and  $\underset{\sim}{+}$ ,  $\underset{\sim}{\max}$  are fuzzy addition and fuzzy maximum respectively.

Given  $FLT(u)$ , let  $\text{min\_st}$  be the minimum time step from  $FST(u)$  whose  $\mu_{FST(u)}$  is nonzero, and  $\text{max\_st}$  be the maximum time step from  $FST(u)$  whose  $\mu_{FST(u)}$  is nonzero. Similarly, let  $\text{min\_fin}$  be the minimum time step from  $MFFT(u)$  whose  $\mu_{MFFT(u)}$  is nonzero, and let  $\text{max\_fin}$  be the maximum time step from  $MFFT(u)$  whose  $\mu_{MFFT(u)}$  is nonzero. Without loss of generality, assume that  $FST(u)$  and  $MFFT(u)$  are sorted in the increasing order of the time step. We create a fuzzy set  $IFST(u)$ , mapping for a discrete time domain  $[\text{min\_st} \dots \text{max\_st}]$  to a real value in  $[0..1]$ , showing the possibility that at time step  $x$ , node  $u$  will occupy a register for  $FST(u)$  and likewise for  $IMFFT(u)$  for  $MFFT(u)$  as in Definitions 5.4–5.5.

**Definition 5.4**  $G = (\mathcal{V}, \mathcal{E}, \beta)$ , a given schedule,  $[\text{min\_st} \dots \text{max\_st}]$  and  $FLT(u)$

$$\mu_{IFST(u)}(c) = \begin{cases} 0 & \text{if } c < \text{min\_st} \text{ or } c > \text{max\_st} \\ \max_{\substack{\forall x, \text{min\_st} \leq x < y \\ y = \max(FST(u)) \text{ and } y < c}} (\mu_{FST(u)}(x)) & \text{otherwise} \end{cases}$$

**Definition 5.5**  $G = (\mathcal{V}, \mathcal{E}, \beta)$ , a given schedule,  $[\text{min\_fin} \dots \text{max\_fin}]$  and  $FLT(u)$

$$\mu_{IMFFT(u)}(c) = \begin{cases} 0 & \text{if } c < \text{min\_fin} \text{ or } c > \text{max\_fin} \\ \max_{\substack{\forall x, y < x \leq \text{max\_fin} \\ y = \max(MFFT(u)) \text{ and } y < c}} (\mu_{MFFT(u)}(x)) & \text{otherwise} \end{cases}$$

From the above calculation, we assume that for any two starting time value  $a, b \in FST(u)$  where  $a < b$ , if node  $u$  starts at time  $a$ , it will be already started at time  $b$ . For  $MFFT(u)$ , when  $a < b$ ,  $a, b \in MFFT(u)$ , if the value for node  $u$  will not be needed at time  $a$ , it will not be needed at time  $b$  and vice versa. Thus, Definitions 5.4–5.5 give the following properties.

**Property 5.1** The possibility of  $IFST(u)$  is in nondecreasing order.

**Property 5.2** The possibility of  $IMFFT(u)$  is in non-increasing order.

From  $IFST(u)$  and  $IMFFT(u)$ , we merge the two sets to create a fuzzy interval for a node by defining Definition 5.6.

**Definition 5.6**  $G = (\mathcal{V}, \mathcal{E}, \beta)$ , a given schedule,  $IFST(u)$  and  $IMFFT(u)$ .

$$\mu_{IFLT(u)}(c) = \begin{cases} 0 & \text{if } c < \min(\text{min\_st}, \text{min\_fin}) \text{ or } c > \max(\text{max\_st}, \text{max\_fin}) \\ \max(\mu_{IFST(u)}(c), \mu_{IMFFT(u)}(c)) & \text{if } \text{min\_st} \leq c \leq \text{max\_st} \\ & \text{or } \text{min\_fin} \leq c \leq \text{max\_fin} \\ 1 & \text{otherwise} \end{cases}$$

After we compute the fuzzy life time interval for each node, we can start compute register usage for each time step.

## 5.2 Register Usage Calculation

Once a scheduled DFG is created,  $FST(u)$  and  $FFT(u)$  must be calculated for all  $u \in V$ . Figure 8 displays the meaning of fuzzy life time implied by Definitions 5.4–5.5.  $SA$  and  $FA$  denote the fuzzy start time and the fuzzy finished time of node  $A$  respectively. Similarly,  $SB$  and  $FB$  denote the start time and the fuzzy finished time of node  $B$ . The fuzzy life times of  $A$  and  $B$  are shown in the filled boxes on the right side.

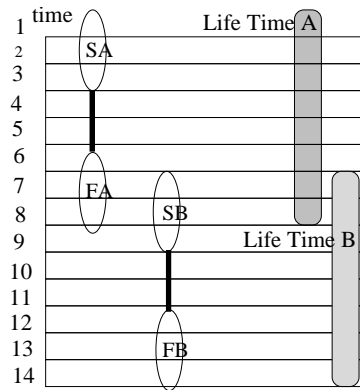


Figure 8: Relationship between scheduled nodes and life time

In the figure, the life time of  $A$  and the life time of  $B$  may overlap. Traditionally, when the timing attribute is precise, the overlapped interval implies that two registers are needed during these time steps. In particular, during time steps 7 and 8, two registers are needed.

When an execution time becomes a fuzzy number, each box still implies that one register is needed. However, the derived possibility associated with a time step indicates that that node may not actually exist during the time step. For example, node may start later or finished earlier. In other words, there is a possibility that a node may not use such a register. With this knowledge, the register may be shared with others with high possibility. Consider the overlap interval in Figure 8 at time step 7. One or two registers may be used with some possibility. This depends on whether the dependency between  $A \rightarrow B$  exists. If edge  $A \rightarrow B$  exists in the original data flow graph, the total register count would be one. Notice that in this case, the intersection of  $IFLT(A)$  and  $IFLT(B)$  is not empty. On the contrary, if  $A$  and  $B$  are independent, the total register count would be two although the intersection may not be empty as well. This issue must be considered in calculating register usages.

Algorithm 5.1 presents a framework in evaluating fuzzy latency and register counts of a schedule. This algorithm is called after Line 7 in Algorithm 4.1 which already assigns the start time for each node.

In Algorithm 5.1, Line 6 invokes Algorithm 5.2 to calculate the life time of all nodes in schedule and find the maximum register usage. The register usage is then kept in  $Reg[S_i]$  for a schedule  $S_i$ . Next, the latency of the whole schedule is then calculated. Note that after invoking Algorithm 5.2, necessary timing attributes for all nodes in  $G_0$

can be obtained. The latency of the schedule is obtained by just fuzzy maximizing the finished time of all leaves in  $G_0$ . Line 9 merges latency and register usage attributes of the schedule using some heuristic function. The combined attribute is denoted as a *quality* of the schedule. This quality is then compared in Line 13 to select the best one.

**Algorithm 5.1 (Eval\_Schedule\_with\_Reg)**

**Input:** schedules  $S_1, S_2$ ,  $G = (\mathcal{V}, \mathcal{E}, \beta)$ , and  $Spec = (F, \mathcal{A}, \mathcal{M}, \mathcal{Q})$

**Output:** 1 if  $S_1$  is better than  $S_2$ , 0 otherwise.

```

1  $G_0 = (\mathcal{V}_0, \mathcal{E}_0, \beta)$  where  $\mathcal{V}_0 = \mathcal{V} - \{\text{unscheduled nodes}\}$ ,  $\mathcal{E}_0 = \emptyset$ 
2 foreach schedule  $S_i = S_1$  to  $S_2$  do
3    $\mathcal{E}_0 = \{(u, v) : u, v \in \mathcal{V}_0, \text{ if } u, v \text{ in same f.u. in } S_i$ 
4     and  $v$  is immediately after  $u\}$ 
5   Calculate register usage for  $G_0$  using Algorithm 5.2
6   Let  $W$  is a set of leaves in  $G_0$ 
7    $latency[S_i] = \text{fuzzymax\_time}(W)$ 
8    $quality[S_i] = \text{Combine}(latency[S_i], \text{Reg}[S_i])$ 
9 od
10 // comparing the overall attributes of both schedules
11 return( $\text{compare}(quality[S_1], quality[S_2])$ )

```

**Algorithm 5.2 (Calculate\_Register\_Count)**

**Input:** Scheduled Graph  $G_0$  for schedule  $S$  and, original DFG  $G = (\mathcal{V}, \mathcal{E}, \beta)$   $Spec = (F, \mathcal{A}, \mathcal{M}, \mathcal{Q})$

**Output:**  $\text{Reg}[S]$  contains register counts needed and its possibility

```

1 Calculate FLT( $u$ )  $\forall u \in G_0$  by Definition 5.3
2 Calculate IFLT( $u$ )  $\forall u \in G_0$  by Definitions 5.4–5.5
3 Let  $max\_cs$  be max. finished time,  $\forall u \in G_0$ 
4 for  $cs = 1$  to  $max\_cs$  do
5   ( $\text{RegAt}[cs].reg, \text{RegAt}[cs].poss$ ) =  $\text{Count\_Node}(\text{IFLT}, cs, G_0)$  od
6  $\forall n, \text{FReg}[n] = 0$ 
7 for  $cs = 1$  to  $max\_cs$  do
8    $\text{FReg}[\text{RegAt}[cs].reg].reg = \text{RegAt}[cs].reg$ 
9    $\text{FReg}[\text{RegAt}[cs].reg].poss =$ 
10    $\max(\text{FReg}[\text{RegAt}[cs].reg].poss, \text{RegAt}[cs].poss)$  od
11  $\text{Reg}[S] = \text{FReg}$ 

```

In Algorithm 5.2,  $\text{RegAt}$  stores maximum number of registers needed at each  $cs$  and its associated possibility. The values are obtained by Algorithm  $\text{Count\_Node}$ . Lines 7–10 summarize the overall number of registers needed and its possibility. Algorithm  $\text{Count\_Node}$  is described in Algorithm 5.3.

**Algorithm 5.3 (Count\_Node)**

**Input:** IFLT,  $G_0$ ,  $cs$

**Output:** # registers needed and its possibility at  $cs$

```

1  node_set = {nodes occupy reg at cs}
2  set  $G_0$  in topological order
3  Let sorted_node be node_set sorted in by sorted  $G_0$ 
4  poss = 0, reg = 0
5   $\forall i \in \textit{sorted\_node}$ , i.ok = FALSE, i.count = FALSE
6  for every i  $\in$  sorted_node do
7      for j = i + 1 to last node in sorted_node do
8          if i.ok = TRUE and i.count = FALSE
9              then
10                 reg ++; poss = max(poss,  $\mu_{\text{IFLT}(i)}(cs)$ )
11                 i.count = TRUE fi
13             if FindPath(i, j)
14                 then j.ok = FALSE // don't count descendant fi
15         od
16     Let j be the last node in sorted_node
17     if j.ok = TRUE
18         then
19             reg ++; poss = max(poss,  $\mu_{\text{IFLT}(j)}(cs)$ )
20             j.count = TRUE fi
21     od
22 return (reg, poss)

```

In Algorithm 5.3, our heuristic only attempts to consider the ancestor at the current time step. In other words, we assume that the ancestor finishes first and then its descendants can start. Flag *ok* uses to indicate that the associated node should be counted at the current step or not. If it is a descendant of any of nodes in the current step, the flag will be disable. Since the schedule contains every node, the descendant will be started eventually. *reg* and *poss* store the current number of counted nodes and maximum possibility. At Line 3, the nodes currently in this time step indicated by IFLT are sorted in the topological order according to  $G_0$ . Then we extract each node in the sorted list to check if any pair are dependent by using *FindPath* in Line 13. In the loop, it selectively marks descendant nodes in the current step.

Let us consider the complexity of Algorithm 5.3. The time complexity is dominated by Lines 6–21, which is  $O(|V|^2(|V| + |E|))$ . Since for DAG, *FindPath* takes  $O(|V| + |E|)$ .

In Algorithm 5.2, the calculation for FLT ( $u$ ) depends on FST ( $u$ ) and MFFT ( $u$ ). Let  $N_1$  be the number of discrete points in FST ( $u$ ) and MFFT ( $u$ ). Lines 1–2 perform the calculation in the same manner to Lines 6–13 of Algorithm 4.2. Therefore, both have upper bound of  $O(N_1|V||E|)$ . The computation for IFLT ( $u$ ) is simply a double loop for each node. In overall, Algorithm 5.2 runs in polynomial time.

## 6 Experiments

We integrate Algorithm 5.1–5.3 into Algorithm 4.2. The new algorithm is called Register-Constrained Inclusion Scheduling (RCIS). In this section, we present an example which shows the calculation for FLT and the resulting schedule. Then we discuss the results on other benchmarks.

## 6.1 Example

Consider the simple DFG presented in Figure 9. Assume that there are four general functional units available. All of them have slightly different latency characteristics according to Table 1. In the figure, Columns “lat” and “pos” show the latency and its possibility of having the latency value if the nodes are executed in a functional unit. In this case, FU1 and FU3 have the same characteristics while FU2 and FU4 have the same characteristics.

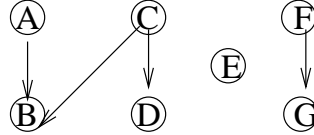


Figure 9: A simple DFG example.

FU1		FU2		FU3		FU4	
lat	pos	lat	pos	lat	pos	lat	pos
5	0.05	7	0.5	5	0.05	7	0.5
10	1	12	0.7	10	1	12	0.7
15	0.9	17	1	15	0.9	17	1
23	0.1	29	0.05	23	0.1	29	0.05

Table 1: Sample functional unit characteristics.

Given the system specification shown in Figure 10, where register axis contains a discrete value ranged in [1..7] and latency axis ranged in [1..200]. In the figure, we use the weighted sum as a criteria similar to Equation (1), where latency : register count is 1:10.

According to Section 5, Figure 11(a) shows the resulting schedule we obtain. We notice that FU1 and FU3 are preferable. To calculate FST( $u$ ), we assume a heuristic where a node starts as early as possible. From this schedule, consider node B. Figure 12(a) presents FLT( $A$ ) containing FST( $A$ ) and MFFT( $A$ ). Figure 12(b) presents FLT( $B$ ). For FST( $A$ ), possibility  $y$  at time  $x$  represents a possibility that node  $A$  occupies register at time  $x$  with respect to the schedule, denoted by rectangles. Notice that for  $A$ , there is only one possible start time whose possibility is one. Similarly for MFFT( $A$ ), possibility  $y$  at time  $x$  represents a possibility that the life time of node  $A$  ends at time  $x$ , denoted by triangles in the Figure. For two dependent nodes  $A, B$ , Figure 13 compares FLT( $A$ ) and FLT( $B$ ). We can see that FST( $B$ ) overlaps with MFFT( $A$ ). Figures 14(b)– 14(a) shows the FLT( $u$ ) all the nodes.

We summarize the register count and its possibility value for each time step as shown in Figure 15. Then we conclude that the register usage as following: **(1,0.1)** and **(2,1)**. It implies that at some control step, 1 register is needed with very low possibility, e.g. 0.05 and 0.1. The maximum possible finished time of the schedule is at 92 with possibility 0.1. With this schedule, the average weighted sum of latency and register is 79.53. Considering only the average latency, the value is 52. Compared to the constraint, with latency 52 and register count 2, the acceptability degree is 0.76. In fact, this gives the same acceptability level as the original inclusion scheduling whose average

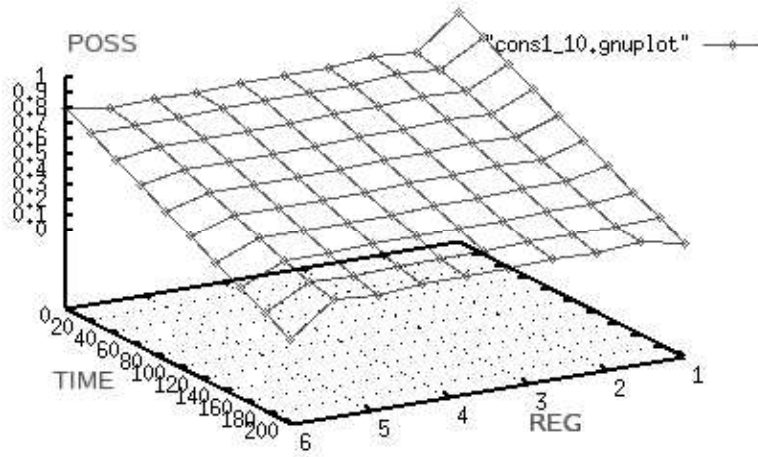


Figure 10: Constraint for Figure 9.

FU1	FU2	FU3	FU4
A	-	E	-
F	-	C	-
G	-	D	-
B	-	-	-

(a)

FU1	FU2	FU3	FU4
A	F	E	-
G	-	C	-
B	-	D	-

(b)

Figure 11: (a) Schedule obtained RCIS for Figure 9 (b) Schedule obtained using the original inclusion scheduling.

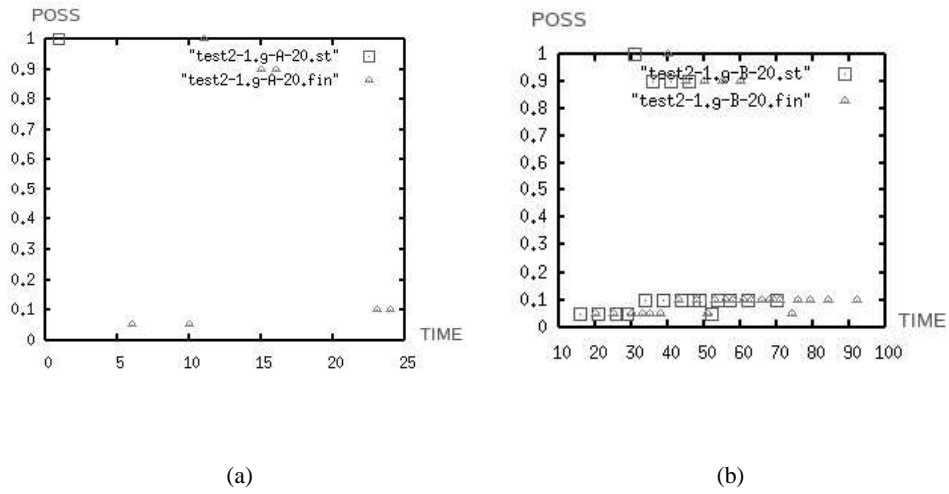


Figure 12: (a) FLT(A) (b) FLT(B).

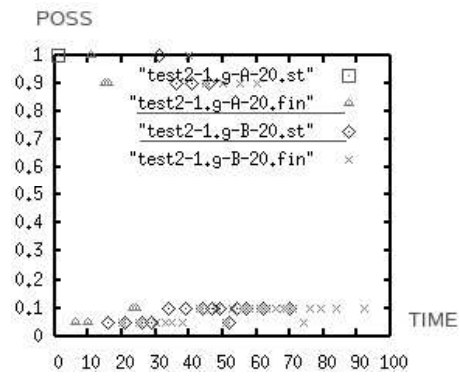
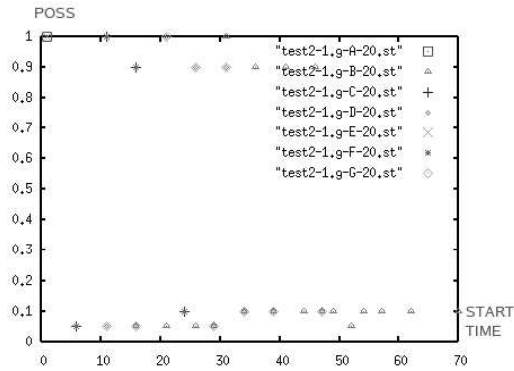
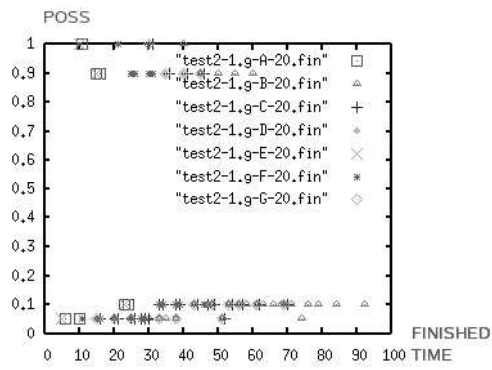


Figure 13: FLT(A) and FLT(B).



(a)



(b)

Figure 14: FLT(u), for all nodes in Figure 9 (a) FST(u) (b) MFFT(u).

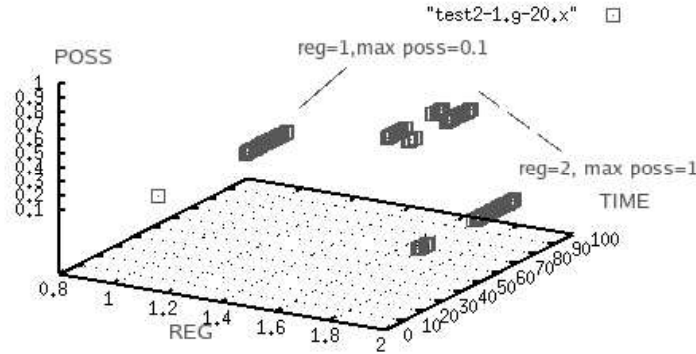


Figure 15: Register counts and possibility each time step.

latency is 41 and the maximum register count is 3. The schedule of this case is given in Figure 11(b).

## 6.2 Other Benchmarks

We have tried to experiment on larger graphs. For instance, we expand the graph in Figure 9 by adding nodes, and following edges  $\{H \rightarrow J, I \rightarrow J\}$  (See Figure 16). Using the same constraint as in Figure 10, and the same functional unit specification in Table 1 while allowing 4 functional units, RCIS yields the schedule with average latency 57 and maximum registers of 2. This yields acceptability value 0.74. Compared to the latency-based inclusion scheduling, it results in average latency 37 with the maximum register of 4 which also gives the same acceptability level.

Consider a well-known benchmark, discrete cosine transform [10], containing 48 nodes. Assume the same functional unit specification for both adders and multipliers and the constraint in Figure 17 where the register axis is [1..12] and the latency axis is [1..500]. We compare the results obtained from various cases of varying the number of functional units. The results are shown in Table 2. Columns “RCIS” and “IS” compare the performance of the schedule by Register-Constrained Inclusion Scheduling and the original inclusion scheduling (IS). Row “Avg Latency” shows the weighted sum of latency for each case. Row “Max Reg” displays the maximum number of registers. Row “Acceptability” shows the acceptability value obtained using the “Avg latency” and “Max Reg”. Row “Max Latency” presents the maximum latency values and Row “Avg Weight” presents weighted sum value for RCIS and IS. For RCIS, recall that  $w_1 = 1$  and  $w_2 = 10$  and for IS, this is the same value as shown in Row “Avg Latency” since we only consider minimizing latency. Tables 3–4 shows the summarized possibility values for using certain register counts for RCIS and IS respectively. It is obvious that IS attempts to minimize latency while not considering the register usage. From these tables, we can achieve about the same acceptability (and even better acceptability in some case) with fewer number of registers, which is upto **37%** saving for the number of registers for the case of 7 adders and 5 multipliers. Among all these cases, we see that the configuration with 5 adders and 4 multipliers should be the best. Consider the running time. For all the cases, the maximum running time is approximately 1 minute 50 seconds to achieve the results for 7 adders and 5 multipliers under Pentium 4 2.8GHz, 1GB RAM.

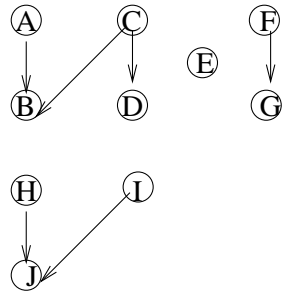


Figure 16: A simple DFG example 2.

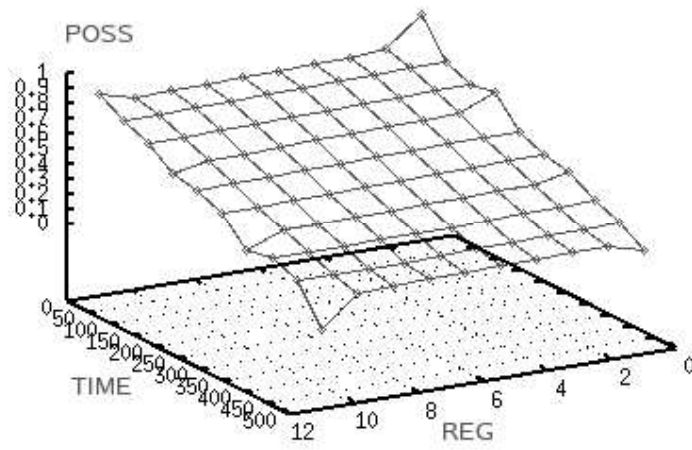


Figure 17: Constraint for DCT.

	5 adds 4 muls		6 adds 4 muls		6 adds 5 muls		7 adds 4 muls		7 adds 5 muls	
	RCIS	IS	RCIS	IS	RCIS	IS	RCIS	IS	RCIS	IS
Avg Latency	122	111	132	98	117	99	124	104	127	94
Max Reg	<b>6</b>	8	<b>7</b>	10	<b>8</b>	10	<b>7</b>	10	<b>7</b>	11
Acceptability	<b>0.719</b>	0.704	<b>0.69</b>	0.69	<b>0.694</b>	0.691	<b>0.699</b>	0.683	<b>0.691</b>	0.683
Max Latency	226	252	296	224	213	197	255	226	230	179
Avg Weight	188	111	198	98	206	99	210	104	209	94

Table 2: Comparison of RCIS and IS when varying the number of functional units.

#reg	2	3	4	5	6	7
poss	0.1	3	0.1	0.1	1	1

Table 3: Possibility values of register counts for case 7 adders and 5 multipliers for RCIS.

#reg	2	4	5	6	7	8	10	11
poss	0.05	0.05	1	0.05	1	0.1	1	1

Table 4: Possibility values of register counts for case 7 adders and 5 multipliers for IS.

## 7 Conclusion

We propose a polynomial-time scheduling algorithm which considers impreciseness in the system specification, constraint and attempts to create a schedule which minimizes both latency and register usages. The algorithm can be integrated into an iterative design process to find acceptable solutions. Our algorithm considers imprecise functional unit characteristic and system requirement. When the timing characteristic is imprecise, life time of a node in the schedule is imprecise. We investigate the imprecise life time of a node in the schedule and analyze the register usage. The algorithm can be integrated into a design exploration which explores an acceptable solution trading off latency cycles with register saving. The experiments show that the better and same quality schedule can be achieved using fewer number of registers compared to the traditional scheduling algorithm.

## References

- [1] I. Ahmad, M. K. Dhodhi, and C.Y.R. Chen. Integrated scheduling, allocation and module selection for design-space exploration in high-level synthesis. *IEEE Proc.-Comput. Digit. Tech.*, 142:65–71, January 1995.
- [2] C. Chantrapornchai, E. H. Sha, and X. S. Hu. Efficient scheduling for imprecise timing based on fuzzy theory. In *Proceedings of Midwest Symposium on Circuits and Systems*, pages 272–275, 1998.
- [3] C. Chantrapornchai, E. H. Sha, and X. S. Hu. Efficient algorithms for finding highly acceptable designs based on module-utility selections. In *Proceedings of the Great Lake Symposium on VLSI*, pages 128–131, 1999.
- [4] C. Chantrapornchai, E. H-M. Sha, and X. S. Hu. Efficient module selections for finding highly acceptable designs based on inclusion scheduling. *J. of System Architecture*, 11(4):1047–1071, 2000.
- [5] C. Chantrapornchai, E. H-M. Sha, and Xiaobo S. Hu. Efficient acceptable design exploration based on module utility selection. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, 19:19–29, Jan. 2000.
- [6] C. Chantrapornchai and S. Tongsima. Resource estimation algorithm under impreciseness using inclusion scheduling. *Intl. J. on Foundation of Computer Science, Special Issue in Scheduling*, 12(5):581–598, 2001.

- [7] S. Chaudhuri, S. A. Bylthe, and R. A Walker. An exact methodology for scheduling in 3D design space. In *Proceedings of the 1995 International Symposium on System Level Synthesis*, pages 78–83, 1995.
- [8] S. Chaudhuri and R. Walker. Computing lower bounds on functional units before scheduling. In *Proceedings of the International Symposium on System Level Synthesis*, pages 36–41, 1994.
- [9] A. Dani, V. Ramanan, and R. Govindarajan. Register-sensitive software pipelining. In *Proceedings of the Merged 12th International Parallel Processing and 9th International Symposium on Parallel and Distributed Systems*, pages 194–198, April 1998.
- [10] M. K. Dhodhi, F. H. Hielscher, R. H. Storer, and J. Bhasker. Datapath synthesis using a problem-space genetic algorithm. *IEEE Transactions on Computer-Aided Design of integrated circuits and systems*, 14(8):934–944, August 1995.
- [11] A. Eichenberger and E. S. Davidson. Register allocation for predicated code. In *Proceedings of MICRO*, 1995.
- [12] Alexandre E. Eichenberger and Edward S. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proceedings of MICRO-28*, pages 338–349, 1995.
- [13] H. Esbensen and E. S. Kuh. Design space exploration using the genetic algorithm. In *Proceedings of the 1996 International Symposium on Circuits and Systems*, pages 500–503, 1996.
- [14] F. Chen, S. Tongsima, and E. H. Sha. Loop scheduling algorithm for timing and memory operation minimization with register constraint. In *Proceedings of SiP'98*, 1998.
- [15] K. Gupta. *Introduction to fuzzy arithmetics*. Van Nostrand, 1985.
- [16] O. Hammami. Fuzzy scheduling in compiler optimizations. In *Proceedings of the ISUMA-NAFIPS*, 1995.
- [17] I. Karkowski. Architectural synthesis with possibilistic programming. In *HICSS-28*, January 95.
- [18] 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.
- [19] A. Kaufmann and M. M. Gupta. *Fuzzy mathematical models in engineering and management science*. North-Holland, 1988.
- [20] A. S. Kaviani and Z. G. Vranesic. On scheduling in multiprocess systems using fuzzy logic. In *Proceedings of the International Symposium on Multiple-valued Logic*, pages 141–147, 1994.
- [21] J. Lee, A. Tiao, and J. Yen. A fuzzy rule-based approach to real-time scheduling. In *Proceedings of FUZZ-94*, volume 2, 1994.
- [22] Josep Llosa, Eduard Ayguade, Antonio Gonzalez, Mateo Valero, and Jason Eckhardt. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Transactions on Computers*, 50(3):234–249, 2001.
- [23] Josep Llosa, Mateo Valero, and Eduard Ayguade. Heuristics for register-constrained software pipelining. In *International Symposium on Microarchitecture*, pages 250–261, 1996.
- [24] C. A. Mandal, P. O. Chakrabarti, and S. Ghose. Design space exploration for data path synthesis. In *Proceedings of the 10th International Conference on VLSI Design*, pages 166–170, 1996.

- [25] K. Mertins et al. Set-up scheduling by fuzzy logic. In *Proceedings of the International conference on computer integrated manufacturing and automation technology*, pages 345–350, 1994.
- [26] J. Rabaey and M. Potkonjak. Estimating implementation bounds for real time DSP application specific circuits. *IEEE Transactions on Computer-Aided Design of integrated circuits and systems*, 13(6), June 1994.
- [27] T. J. Ross. *Fuzzy Logic with Engineering Applications*. McGrawHill, 1 edition, 1995.
- [28] A. Sharma and R. Jain. Estimating architectural resources and performance for high-level synthesis applications. *IEEE Transactions on VLSI systems*, 1(2):175–190, June 1993.
- [29] H. Soma, M. Hori, and T. Sogou. Schedule optimization using fuzzy inference. In *Proceedings of FUZZ-95*, pages 1171–1176, 1995.
- [30] I.B. Turksen et al. Fuzzy expert system shell for scheduling. *SPIE*, pages 308–319, 1993.
- [31] L. A. Zadeh. The concept of a linguistic variable and its application to approximate reasoning, Part I. *Information Science*, 8:199–249, 1975.
- [32] L. A. Zadeh. Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets and Systems*, 1:3–28, 1978.
- [33] L. A. Zadeh. Fuzzy Logic. *Computer*, 1:83–93, 1988.
- [34] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Software and hardware techniques to optimize register file utilization in vliw architectures. In *Proceedings of the International Workshop on Advanced Compiler Technology for High Performance and Embedded Systems (IWACT)*, July 2001.