

# Design Exploration with Imprecise Latency and Register Constraints

C. Chantrapornchai, W. Surakampolthorn *Senior Member, IEEE*, and E. H-M. Sha *Senior Member, IEEE*

**Abstract**—We propose a design exploration framework which considers impreciseness in design specification. In high-level synthesis, imprecise information is often encountered. Two types of impreciseness are considered in the paper: impreciseness underlying on functional unit specifications and impreciseness due to the system constraints: latency and register constraints. The framework is iterative and based on a core scheduling called, *Register-Constrained Inclusion Scheduling*. An example on how the scheduling algorithm works is shown. We demonstrate the effectiveness of our framework for imprecise specification by exploring a design solution for three well-known benchmarks, *Discrete Cosine Transform*, *Voltera Filter*, and *Fast Fourier Transform*. The selected solution meets the acceptability criteria while minimizing the total number of registers.

**Index Terms**—Imprecise Design Exploration, Scheduling/Allocation, Multiple design attributes, Imprecise information, Register constraint, Inclusion Scheduling

## I. 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

Manuscript received April 20, 2005; revised September 23, 2005, January 12, 2006. This work was supported by the TRF under grant number MRG4680115, Thailand, TI University Program, NSF EIA 0103709, Texas ARP-009741-0028-2001 and NSF CCR-0309461, USA.

C. Chantrapornchai is with Department of Computing, Faculty of Science, Silpakorn University, Thailand, ctana@su.ac.th

W. Surakumpolthorn is with Department of Electrical Engineer, Faculty of Engineering, King Mongkut's Institute of Technology, Thailand, kswanlop@kmitl.ac.th

E.H-M. Sha is with Department of Computer Science, University of Texas at Dallas, U.S.A., edsha@utdallas.edu

Copyright (c) 2006 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from IEEE by sending an email to pubs-permissions@ieee.org.

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 propose a design exploration framework which considers imprecise information underlying in system specification and requirements. Particularly, we are interested in the latency and register constraints. However, the approach can be extended to handle other multiple design criteria. The system characteristics are modeled based on the fuzzy set theory. Register count is considered as another dimension of imprecise system requirement. The work in [1], [2] is used as a scheduling core in the iterative design refinement process. The imprecise schedule which minimizes the register usage is generated. If the schedule meets acceptability criteria, the design solution is selected. Otherwise, the resources are adjusted and the process is repeated. Our input system is modeled using a data flow graph with imprecise timing parameters. 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.

Many research results are available for design space exploration [3], [4], [5], [6]. 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. Karkowski and Otten introduced a model to handle the imprecise propagation delay of events [7], [8]. 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 [9], [10], [11]. These approaches, however, neither consider multiple design attributes nor impreciseness in system characteristics.

Many research works in the scheduling and register allocation area exist in high-level synthesis and compiler optimization area for VLIW architecture. Varatkar et. al. proposed a scheduling algorithm for multiprocessor systems which considers minimizing the total system energy [12]. Shao et. al. presented instruction scheduling for loop applications which considers minimizing the total switching activity [13]. These work, however, do not consider register usage minimization. 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 [15], [16]. Akturan and Jacome also proposed a scheduling algorithm which considers minimizing register usage for software pipelining [17]. The algorithm uses retiming and force directed scheduling and explores the trade-off between code size, performance, and register requirements. Wong et. al. developed a strategy

to insert objective functions during scheduling and register allocation steps [18]. Their algorithm is called, scheduling FLOF, which attempts to minimize the register usage subjected to the latency and resource constraints. Dani et. al. also presented a heuristic which uses stage scheduling to minimize register requirement. They also targeted at instruction level scheduling [19]. Zalamea et. al. presented hardware and software approach to minimize the register's usage targeting VLIW architecture [20], [21], [22]. On the software side, they proposed an extended version of modulo scheduling which considers a register constraint, and register spilling. However, these works focus on loop scheduling and do not consider handling the imprecise system characteristics or specification.

In [1], the inclusion scheduling which takes the imprecise system characteristic was proposed. The algorithm was expanded and used in design exploration under imprecise system requirement as well as the estimation of resource bounds [23], [24], [25]. However, it does not take register criteria in creating a schedule.

In this paper, we particularly consider both imprecise latency and register constraints. We develop a design exploration framework under imprecise specification and constraints. The framework is iterative and based on the developed scheduling core, RCIS, *Register-Constrained Inclusion Scheduling* that takes imprecise information into account. Experimental results show that we can achieve an acceptable design solution with minimized number of registers.

This paper is organized as follows: Section II describes our models. It also presents some backgrounds in fuzzy set. Section III presents the iterative design framework. Section IV presents the scheduling core (RCIS) used in the design exploration framework. It also addresses some issues when the register count is calculated during scheduling. An example how the scheduling algorithm works is shown in Section V. Section VI displays some experimental results. Finally, Section VII draws a conclusion from our work.

## II. 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.

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 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 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 numbers 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  $\mathcal{Q}$  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 perfect system would have both latency and register count being less than or equal to 100 and 1 respectively. An acceptability function,  $\mathcal{Q}(a_1, a_2)$  for such a specification is formally defined as:

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

where  $F$  is assumed to be linear functions, e.g.,  $F(a_1, a_2) = -0.002439(a_1 + 2a_2) + 1.248780$  which returns the acceptability between  $(0,1)$ . 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.

In general, one may model any criteria by

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

In Equation 2,  $F$  can be any function. For example, we can simply replace the register constraint by others such as power. Figure 2(b) depicts another example of the specification. Figure 2(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  $\mathcal{Q}$  function to a latency-acceptability plane. An inner curve (tighter latency constraint) corresponds to larger power values. Based on the acceptability model, a design with high acceptability implies an optimized design towards certain goals.

Notice that  $\mathcal{Q}$  represents both design constraint and design goal. For a given system, a designer can specify their interested criteria and the constraint for each criteria. Rather than giving the crisp value for a constraint, the constraint is a fuzzy set. The shape of the function can be selected based on the preference of the designer or can be obtained using a typical method to calculate the membership values [26]. When considering more than one design criteria, these criteria can be correlated. One may express the design constraint and goal as a multiple objective functions which can be implied by the  $\mathcal{Q}$  function as well.

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

*Given a specification containing  $S = (\mathcal{F}, \mathcal{A}, \mathcal{Q})$ ,  $G = (\mathcal{V}, \mathcal{E}, \beta)$ , find a schedule under functional unit and register constraints for each  $f$  in  $\mathcal{F}$  which maximizes the acceptability level.*

In this work, the term ‘‘schedule’’ refers to finding the execution sequence of operations in the DFG as well as finding the functional unit binding of the operations.

In this research, we use some of the fuzzy set concepts. Next, we give a quick review of fuzzy set theory as it relates to our work.

Fuzzy sets, proposed by Zadeh, represent a set with imprecise boundary [27], [28]. 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  $R$ . 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 [29]:

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

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

Fuzzy arithmetic is used to compute an arithmetic operation between two fuzzy numbers. Figure 3(a) shows an example of fuzzy number  $A$ . 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* is 1. Similarly, Figure 3(b) shows the fuzzy number with the confidence interval  $(3, 7)$  representing  $B$ .

Figure 3(c) demonstrates a graphical result of adding two fuzzy numbers defined on the integer line from Figures 3(a)–3(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 [30]. In addition, the defuzzified value can be used to represent the fuzzy set. Several defuzzified methods can be found in [26]. 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. For example, a given a functional unit  $f$  may require 10, 20, 35 and 70 units of execution time with the possibility of .2, .4, 1, .7 respectively. We describe its possible characteristic set as  $A_f$ . and its associated possibility as  $\mu_f(a) \in [0, 1], \forall a \in A_f$ . Then, a possibility function of the fuzzy set of timing characteristic of a functional unit  $f$  is  $\mu_f(10) = .2, \mu_f(20) = .4, \mu_f(35) = 1, \mu_f(70) = .7$ .

### III. ITERATIVE DESIGN FRAMEWORK

Figure 4 presents an overview of our iterative design process for finding a satisfactory solution. One may estimate the initial design configuration with any heuristic for example using ALAP, and/or ASAP scheduling [25]. The RCIS scheduling and allocation process produces the imprecise schedule attributes which are used to determine whether or not the design configuration is acceptable.

RCIS is a scheduling and allocation process which 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 given acceptability function is then checked against the derived attributes of the schedule.

In order to determine whether or not the resource configuration satisfies the objective function, we use the *acceptability threshold*. If the schedule attributes result in the acceptability level being greater than the threshold, the process stops. Otherwise, the resource configuration is adjusted using a heuristic and this process is repeated until the schedule cannot be improved or the design solution that satisfies the threshold is found. Note that if we are able to find the schedule that satisfies the given acceptability threshold, the threshold can be increased and the whole process can be repeated.

Generally, the constraints can be obtained in the similar way as in the traditional scheduling problem. After computing the precise maximum timing value and register that one could have from the DFG, the fixed value can be expanded as a range of acceptable values associated with various possibility degrees. For an acceptability threshold, one would prefer a perfect value 1. In practice, it is usually impossible to obtain the perfect value. Thus the threshold may be reduced to 0.9 or 0.8 accordingly. On the other hand, one may compute the maximum number of functional units and types for a given DFG. By varying the functional unit configuration and performing the proposed scheduling algorithm, the designer can find the configuration that leads to the best acceptability level.

### IV. REGISTER-CONSTRAINT INCLUSION SCHEDULING

In this section, we present the register-constraint inclusion scheduling (RCIS) algorithm. The algorithm is based on the *inclusion scheduling* core presented in Algorithm 1. The algorithm evaluates the quality of the schedule by considering imprecise register criteria. which is represented by the model in Section II.

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 is used to compute possible latency from the given functional specification. Then, using a fuzzy scheme, latencies 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 [31]. Algorithm 1 presents a list-based inclusion scheduling framework.

**Algorithm 1 (Register-Constrained Inclusion scheduling)**

**Input:**  $G = (\mathcal{V}, \mathcal{E}, \beta)$ ,  $Spec = (F, \mathcal{A}, \mathcal{Q})$ , 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  $\neq$  empty do
3      Q = prioritized(Q)
4      u = dequeue(Q);
5      mark u scheduled
6      good_S = NULL;
7      foreach f  $\in$  {fj : fj is able to perform  $\beta(u)$ ,  $1 \leq j \leq N$ } do
8          temp_S = assign_heuristic(S, u, f)      // assign u at FU f
9          if Eval_Schedule_with_Reg(good_S, temp_S, G, Spec)
10             then good_S = temp_S fi od
11     S = good_S                                // keep good schedule
12     foreach v : (u, v)  $\in$  E do
13         indegree(v) = indegree(v) - 1
14         if indegree(v) = 0 then enqueue(Q, v) fi od
15 od
16 return(S)

```

After node  $u$  is assigned to  $f$ , the imprecise attributes of the intermediate schedule, is computed. *Eval\_Schedule\_with\_Reg* compares the current schedule with the “best” one found in previous iterations. The better one is then chosen and the process is repeated for all nodes in the graph.

In Algorithm 1, fuzzy arithmetic simply takes place in routine *Eval\_Schedule\_with\_Reg* (Line 9). In this routine, we also consider the register count used in the schedule. Since an execution time of a node is imprecise, the life time of a node is imprecise. 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 the 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 3).

A. *Imprecise Timing Attributes*

In the following, we present basic terminologies used in the algorithm which calculates register usage under impreciseness.

**Definition 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  $y$ .

Without loss of generality, for nodes that are executed at the first time step in each functional unit,  $FST(u) = \{(1, 1)\}$ , which implies that the start time of  $u$  is at 1 with possibility 1.

**Definition 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 the earliest start time of a node,  $FST(v) = \max_i(FFT(u_i)) + 1, \forall u_i \rightarrow v$ .

The general idea of using fuzzy numbers is depicted in Figure 5 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 5(b). When a node occupies a resource at a certain time step, a possibility value is associated with the resource 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  $MFFT(u)$ .

**Definition 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  $MFFT(u) = FFT(u) + \max(FST(v_i))$ , where  $u \rightarrow v_i \in \mathcal{E}$  and  $+$ ,  $\max$  are fuzzy addition and fuzzy maximum respectively.

In other words,  $MFFT(u)$  is the fuzzy set of maximum time for node  $u$  to hold a value for its successors. Given  $FLT(u)$ , let  $\min\_st$  be the minimum time step from  $FST(u)$  whose  $\mu_{FST(u)}$  is nonzero, and  $\max\_st$  be the maximum time

step from  $FST(u)$  whose  $\mu_{FST(u)}$  is nonzero. Similarly, let  $min\_fin$  be the minimum time step from  $MFFT(u)$  whose  $\mu_{MFFT(u)}$  is nonzero, and let  $max\_fin$  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  $[min\_st...max\_st]$  to a real value in  $[0..1]$ , calculating 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 4–5.

**Definition 4**  $G = (\mathcal{V}, \mathcal{E}, \beta)$ , a given schedule,  $[min\_st...max\_st]$  and  $FLT(u)$

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

**Definition 5**  $G = (\mathcal{V}, \mathcal{E}, \beta)$ , a given schedule,  $[min\_fin...max\_fin]$  and  $FLT(u)$

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

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 4–5 give the following properties.

**Property 1** The possibility of  $IFST(u)$  is in nondecreasing order.

**Property 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 6.

**Definition 6**  $G = (\mathcal{V}, \mathcal{E}, \beta)$ , a given schedule,  $IFST(u)$  and  $IMFFT(u)$ .

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

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

## B. Register Usage Calculation

Once a scheduled DFG is created,  $FST(u)$  and  $MFFT(u)$  must be calculated for all  $u \in V$ . Figure 6 displays the meaning of fuzzy life time implied by Definitions 4–5.  $SA$  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.

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 6 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. although the intersection of  $IFLT(A)$  and  $IFLT(B)$  may not be 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.

## C. Algorithms

Algorithm 2 presents a framework in evaluating fuzzy latency and register counts of a schedule. This algorithm is called after Line 8 in Algorithm 1 which already assigns the start time for each node.

In Algorithm 2, Line 5 invokes Algorithm 3 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 3, 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 8 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 11 to select the best schedule.

### Algorithm 2 (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 3
6   Let  $W$  is a set of leaves in  $G_0$ 
7    $latency[S_i] = \text{fuzzymax\_time}(W)$ 
8    $quality[S_i] = \text{Combine}(latency[S_i], Reg[S_i])$ 
9 od
10 // comparing the overall attributes of both schedules
11 return( $\text{compare}(quality[S_1], quality[S_2])$ )

```

### Algorithm 3 (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:**  $Reg[S]$  contains register counts needed and its possibility

```

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

```

In Algorithm 3,  $RegAt$  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 4.

### Algorithm 4 (Count\_Node)

**Input:**  $IFLT, G_0, cs$   
**Output:** # registers needed and its possibility at  $cs$

```

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

```

```

20 od
21 return ( $reg, poss$ )

```

In Algorithm 4, 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 time step or not. If it is counted, it means that one more register is needed. 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 is dependent by using  $\text{FindPath}$  in Line 12. In the loop, it selectively marks descendant nodes in the current step.

Let us consider the complexity of Algorithm 4. The time complexity is dominated by Lines 6–20, which is  $O(|V|^2(|V| + |E|))$ , since for DAG,  $\text{FindPath}$  takes  $O(|V| + |E|)$ .

In Algorithm 3, 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 whose upper bound is of  $O(N_1|V||E|)$ . The computation for  $IFLT(u)$  is simply a double loop for each node. In overall, Algorithm 3 runs in polynomial time.

## V. EXAMPLE

We integrate Algorithm 2 into Algorithm 1. 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.

Consider the simple DFG presented in Figure 7. Assume that there are four general functional units available, where FU1 and FU3 have the same characteristics as well as FU2 and FU4 as shown in Table I. In the figure, Columns “(lat,poss)” 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.

Given the system specification shown in Figure 8, where register axis contains discrete values ranged in [1..7] and latency axis contains values 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.

Consider how Algorithm 1 works. Algorithm 1 is based on list scheduling, in this DFG, the root set contains  $\{A, E, F\}$ .

We assume no priority function here. First node  $A$  is picked to schedule. After placing node  $A$  in the schedule, node  $E$  is scheduled next in the queue  $Q$ . Note that  $E$  can be placed at FU1 after  $A$  and at FU3 as the first node. To choose where to place  $E$ , Algorithm 1, *Eval\_Schedule\_with\_Reg*, is called. *Eval\_Schedule\_with\_Reg* compares both schedules by considering latency and register usage. It computes the register usage for each schedule in Line 5 (Algorithm 2). Next, Line 7 calculates the fuzzy latency of the schedule using the fuzzy maximum operator. Then both attributes are combined using a heuristic function at Line 8. In this example, we use the weighted sum of  $w_1(\sum_i l_i p_{l_i}) + w_2(\sum_i r_i p_{r_i})$  where  $w_1 = 1$  and  $w_2 = 10$  and  $l_i$  and  $p_{l_i}$  are various latency values and their associated possibilities, and  $r_i$  and  $p_{r_i}$  are various register counts and their associated possibilities of the schedule.

To compute the fuzzy set of register usage of the schedule ( $r_i$  and  $p_{r_i}$ ), Algorithms 3–4 are applied. Let consider an example of comparing the two schedules depicted in Figure 9(a) and Figure 9(b). When first placing node  $A$ , we compute  $FST(A) = \{(1, 1)\}$ . This means node  $A$  starts at time 1 with possibility 1. Since  $FFT(A) = FST(A) + EXEC(A)$ .  $FFT(A) = \{(5, 0.05), (10, 1.00), (15, 0.90), (23, 0.10)\}$ . After placing  $E$  after  $A$ , we compute  $FST(E) = \{(6, 0.05), (11, 1.00), (16, 0.90), (24, 0.10)\}$  since  $FST(v) = \max_i(FFT(u_i)) + 1, \forall u_i \rightarrow v$ , where  $u_i$  is  $A$  in this case. Next, compute  $FFT(E) = FST(E) + EXEC(E)$  which becomes  $\{(10, 0.05), (15, 0.05), (20, 1.00), (25, 0.90), (28, 0.05), (30, 0.90), (33, 0.10), (38, 0.10), (46, 0.10)\}$ . For this schedule, the maximum time step  $\max_{cs}$  is 46. Figure II presents  $IFLT(A)$  and  $IFLT(E)$ . Column  $cs$  presents the discretized time steps. Column FU1 presents nodes that are executed in the resource. In this case we schedule node  $A$  and  $E$  after node  $A$ . The value associated with the node name, calculated by Definition 6, is the possibility that the node is occupied at the time step. Since the boundaries of fuzzy start time and finished time are fuzzy, we can see that at time steps 6–23, the two nodes occupy the resource. However, when  $E$  is placed after  $A$  in the schedule, an extra edge is constructed from  $A$  to  $E$  to show the sequence of execution in FU1. During these time steps, Algorithm 4 (Count\_Node) considers that only one register will be needed. The last column “(RegAt(cs),poss)” shows the summarized register count and its possibility at each time step computing by Line 5 in the algorithm. From the table, we will need 1 register for every time step in [1..46]. We perform the similar calculation for the schedule in Figure 9(b). We can see that Figure 9(a) gives the weighted sum valued 26.716419 while

Figure 9(b) gives the weighted sum valued 14.707317. Hence, the position in FU3 is chosen for node  $E$ .

Following the approach, the final schedule is constructed as in Figure 10(a). The register count and its possibility value for each time step as shown in Figure 11. This is computed by Algorithm 2. From the figure, we can 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 schedule given by the original inclusion scheduling (as shown in Figure 10(b)) whose average latency is 41 and the maximum register count is 3.

We have tried to experiment on larger graphs. For instance, we expand the graph in Figure 7 by adding nodes, and following edges  $\{H \rightarrow J, I \rightarrow J\}$ . Using the same constraint as in Figure 8, and the same functional unit specification in Table I 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 value.

## VI. EXPERIMENTAL RESULTS

We consider the experiments on exploring design solutions for Discrete Cosine Transform (DCT) [32], Volterra filter benchmark [25], and Fast Fourier Transform.

### A. Discrete Cosine Transform

Consider a well-known benchmark, Discrete Cosine Transform, containing 48 nodes. Assume the same functional unit specification for both adders and multipliers and the constraint in Figure 12 where the register axis is [1..12] and the latency axis is [1..500]. We assume the functional unit characteristics similar to the example in the previous section. This is shown in Table III. We compare the results obtained from various cases of varying the number of functional units. The results are shown in Table IV. Columns “RCIS” and “IS” compare the performance of the schedule by Register-Constrained Inclusion Scheduling and the original inclusion scheduling (IS) [1]. 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 values in Row “Avg latency” and “Max Reg”. Row “Max Latency” presents the maximum latency values and Row “Avg Weight” presents weighted sum values 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 V–VI shows the summarized possibility values for using certain register counts for RCIS and IS respectively [2]. 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 1.8GHz, 1GB RAM.

### B. Votera filter

We present experimental results on Voltera filter benchmark, containing 27 nodes, where 10 nodes require adder units and the rest requires multiplier units. Assume that we have two types of functional units: adder and multiplier, whose latencies are as shown according to Table I.

Assume the constraint is depicted in Figure 13 where the register axis is [1..7] and the latency axis is [200..700]. In the experiment, we consider various design configurations by varying the number of functional units. We compare the results obtained by RCIS and the original inclusion scheduling as a scheduling core in the design exploration. Due to the characteristic of the filter, increasing the number of multipliers will help reduce the overall latency. Suppose that we set the acceptability threshold to be 0.8. The results are shown in Table VII. Recall that  $w_1 = 1$  and  $w_2 = 10$ . RCIS attempts to create a schedule which minimizes the total weighted sum of  $w_1x + w_2y$  where  $x$  and  $y$  are the weighted latency and weighted register counts of the resulting schedule. Figure 14 depicts acceptability values for each design configuration based on RCIS. When we increase the number of functional units the latency decreases while the number of register counts needed increases. However, when the number of multipliers becomes 4 or more, RCIS can create a schedule which gives the maximum acceptability value 0.84 (which is greater than the threshold defined at 0.8). By inspecting the resulting schedule, we conclude that 4 multipliers would be sufficient and adding more multipliers will be wasteful. Compared this

the schedule generated by IS, we found that since IS does not consider the register criteria, IS attempts to utilize all available resources to minimize the overall latency values. Thus, the latency of schedule generated by IS keep decreasing and the number of register counts keep increasing. This will finally decrease the acceptability value according the constraint.

From the results, we can see that to achieve the acceptability threshold 0.8, using RCIS will give a better design solution using fewer number of registers. Consider the running time. For all the cases, the maximum running time is approximately 2.8 seconds to achieve the results for 1 adder and 5 multipliers on the same computer.

### C. Fast Fourier Transform

We experiment the approach on Fast Fourier Transform benchmark containing 36 nodes, where 12 nodes require multiplier units and 24 nodes require adder units. Assume that we use the same type of functional units as in Table III. Assume the constraint is similar to that of Figure 13, where the latency axis is [200..500] and the register axis is [1..10]. In this experiment, we set  $w_1 = 1$  and  $w_2 = 10$ . Based on the characteristics of the benchmark, increasing the adders will help improve the schedule. However, from the results shown in Table VIII, we found that RCIS attempts to reduce the number of registers; thus, the schedules’ maximum latency is longer than those of IS while the number of registers is fewer. When increasing the number of adders, RCIS still attempts to reduce the number of registers. However, the register counts may not be reduced further. Overall, for RCIS, we can see that the configuration of 5 adders and 2 multipliers gives the same performance as the other configurations in the Table. It gives the acceptability value 0.87 while IS yields the schedule with acceptability value 0.84. Therefore, 5 adders and 2 multipliers would be sufficient for this benchmark.

## VII. CONCLUSION

In architecture synthesis, imprecise information is unavoidable. In this paper, we propose a design exploration framework considering impreciseness in both system specification and constraint. The framework is based on the scheduling core, RCIS which considers the impreciseness and attempts to create a schedule which minimizes both latency and register usages. From the experiments on well-known benchmarks, by considering both imprecise latency and register constraints, RCIS can find a schedule which gives the same performance while saving upto 37% of number of registers. Hence, using the framework, we can select design solutions with specified acceptability levels.

## REFERENCES

- [1] 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*, 1998, pp. 272–275.
- [2] C. Chantrapornchai, W. Surakumpolthorn, and E. Sha, "Efficient scheduling for design exploration with imprecise latency and register constraints," in *Lecture Notes in Computer Science: 2004 International Conference on Embedded and Ubiquitous Computing (EUC)*, 2004, pp. 259–270.
- [3] I. Ahmad, M. K. Dhodhi, and C. Chen, "Integrated scheduling, allocation and module selection for design-space exploration in high-level synthesis," *IEEE Proc.-Comput. Digit. Tech.*, vol. 142, pp. 65–71, January 1995.
- [4] S. Chaudhuri, S. A. Blythe, and R. A. Walker, "An exact methodology for scheduling in 3D design space," in *Proceedings of the 1995 International Symposium on System Level Synthesis*, 1995, pp. 78–83.
- [5] H. Esbensen and E. S. Kuh, "Design space exploration using the genetic algorithm," in *Proceedings of the 1996 International Symposium on Circuits and Systems*, 1996, pp. 500–503.
- [6] 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*, 1996, pp. 166–170.
- [7] I. Karkowski, "Architectural synthesis with possibilistic programming," in *HICSS-28*, January 95.
- [8] I. Karkowski and R. H. J. M. Otten, "Retiming synchronous circuitry with imprecise delays," in *Proceedings of the 32nd Design Automation Conference*, San Francisco, CA, 1995, pp. 322–326.
- [9] S. Chaudhuri and R. Walker, "Computing lower bounds on functional units before scheduling," in *Proceedings of the International Symposium on System Level Synthesis*, 1994, pp. 36–41.
- [10] 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*, vol. 13, no. 6, June 1994.
- [11] A. Sharma and R. Jain, "Estimating architectural resources and performance for high-level synthesis applications," *IEEE Transactions on VLSI systems*, vol. 1, no. 2, pp. 175–190, June 1993.
- [12] G. Varatkar and R. Marculescu, "Communication-aware task scheduling and voltage selection for total systems energy minimization," in *IEEE/ACM Intl. Conf. on Computer Aided Design*, November 2003.
- [13] Z. Shao, Q. Zhuge, M. Liu, B. Xiao, and E. H.-M. Sha, "Switching activity minimization on instruction-level loop scheduling for VLIW DSP applications," in *Proceedings of ASAP*, 2004.
- [14] F. Chen, S. Tongshima, and E. H. Sha, "Loop scheduling algorithm for timing and memory operation minimization with register constraint," in *Proceedings of SiP'98*, 1998.
- [15] A. Eichenberger and E. S. Davidson, "Register allocation for predicated code," in *Proceeding of MICRO*, 1995.
- [16] A. E. Eichenberger and E. S. Davidson, "Stage scheduling: A technique to reduce the register requirements of a modulo schedule," in *Proceedings of MICRO-28*, 1995, pp. 338–349.
- [17] C. Akturan and M. F. Jacome, "RS-FDRA - a register sensitive software pipelining algorithm for embedded VLIW processors," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 12, no. 21, pp. 1395–1415, December 2002.
- [18] J. L. Wong, S. Megerian, and M. Potkonjak, "Forward-looking objective functions: Concepts & applications in high level synthesis," in *Proceedings of Design Automation Conference*, 2002.
- [19] 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*, April 1998, pp. 194–198.
- [20] J. Llosa, E. Ayguade, A. Gonzalez, M. Valero, and J. Eckhardt, "Lifetime-sensitive modulo scheduling in a production environment," *IEEE Transactions on Computers*, vol. 50, no. 3, pp. 234–249, 2001.
- [21] J. Llosa, M. Valero, and E. Ayguade, "Heuristics for register-constrained software pipelining," in *International Symposium on Microarchitecture*, 1996, pp. 250–261.
- [22] 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.
- [23] 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*, vol. 11, no. 4, pp. 1047–1071, 2000.
- [24] —, "Efficient acceptable design exploration based on module utility selection," *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 19–29, Jan. 2000.
- [25] C. Chantrapornchai and S. Tongshima, "Resource estimation algorithm under impreciseness using inclusion scheduling," *Intl. J. on Foundation of Computer Science, Special Issue in Scheduling*, vol. 12, no. 5, pp. 581–598, 2001.
- [26] T. J. Ross, *Fuzzy Logic with Engineering Applications*, 1st ed. McGrawHill, 1995.
- [27] L. A. Zadeh, "The concept of a linguistic variable and its application to approximate reasoning, Part I," *Information Science*, vol. 8, pp. 199–249, 1975.
- [28] —, "Fuzzy Logic," *Computer*, vol. 1, pp. 83–93, 1988.
- [29] K. Gupta, *Introduction to fuzzy arithmetics*. Van Nostrand, 1985.
- [30] A. Kaufmann and M. M. Gupta, *Fuzzy mathematical models in engineering and management science*. North-Holland, 1988.
- [31] 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*, 1999, pp. 128–131.
- [32] 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*, vol. 14, no. 8, pp. 934–944, August 1995.



**Chantana Chantrapornchai** Chantana Chantrapornchai received her Ph.D. degree in Computer Science and Engineering, University of Notre Dame, USA, in 1999. She received her bachelor degree in Computer Science from Thammasat University, Bangkok, Thailand in 1992 and M.S. degree in Computer Science from Northeastern University, Boston, Massachusetts in 1994. Currently she is currently an associate professor in Department of Computig,

Faculty of Science, Silpakorn University, Thailand. Her research interests include architecture-level synthesis, real-time and embedded systems, fuzzy systems, and wireless systems.



**Edwin Hsing-Mean Sha** Edwin Hsing-Mean Sha (S'88-M'92-SM'04) received the B.S.E. degree in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 1986; he received the M.A. and Ph.D. degree from the Department of Computer Science, Princeton University, Princeton, NJ, in 1991 and 1992, respectively. From August 1992 to August 2000, he was with the Department of Computer Science and Engineering

at University of Notre Dame, Notre Dame, IN. He served as Associate Chair from 1995 to 2000. Since 2000, he has been a tenured full professor in the Department of Computer Science at the University of Texas at Dallas.

He has published more than 200 research papers in refereed conferences and journals. He has served as an editor for many journals, and as program committee members and Chairs in numerous international conferences. He received Oak Ridge Association Junior Faculty Enhancement Award, Teaching Award, Microsoft Trustworthy Computing Curriculum Award, and NSF CAREER Award.

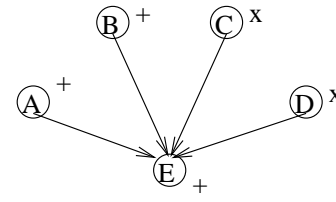
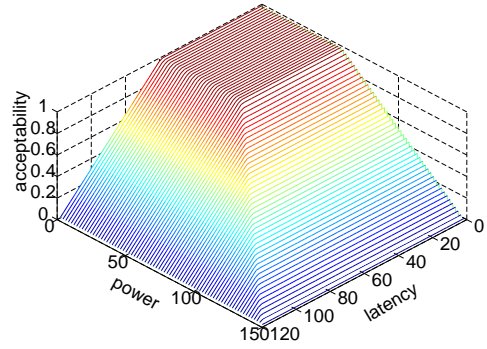
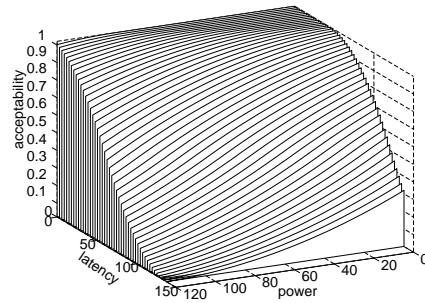


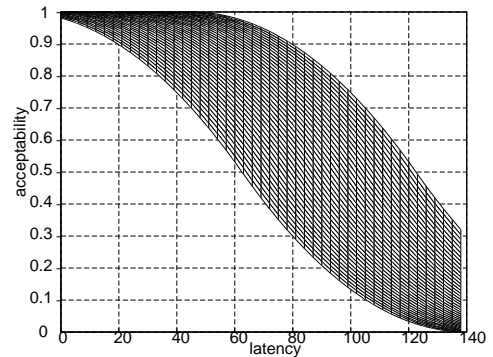
Fig. 1. Test1: Data flow graph example.



(a) Linear acceptability



(b) z curve acceptability with trade-off



(c) Latency acceptability curves corresponding to different power constraints derived from Figure 2(b)

Fig. 2. Various kinds of acceptability functions.

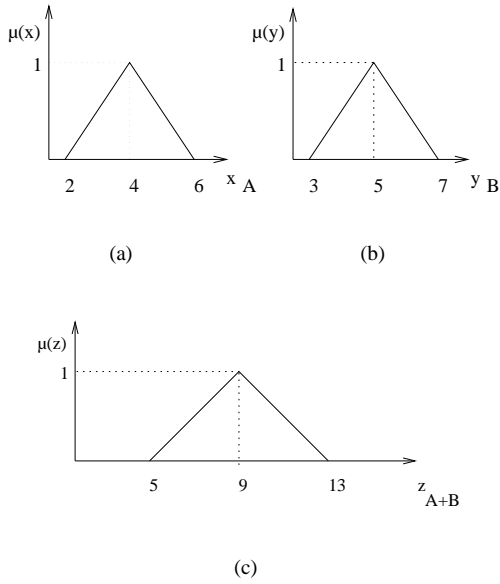


Fig. 3. Adding two fuzzy numbers, A and B. (a) A (b) B (c) A + B.

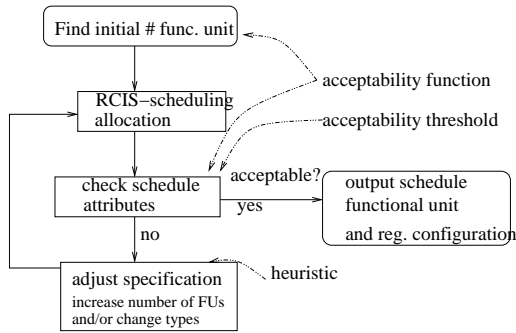


Fig. 4. Design solution finding process using RCIS.

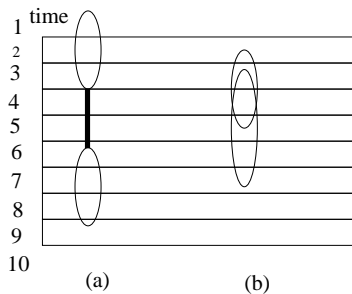


Fig. 5. A view of fuzzy start time and finished time.

FU1	FU2	FU3	FU4
A	-	-	-
E	-	-	-

(a)

FU1	FU2	FU3	FU4
A	-	E	-

(b)

Fig. 9. (a) Schedule obtained RCIS for Figure 7 (b) Schedule obtained using the original inclusion scheduling.

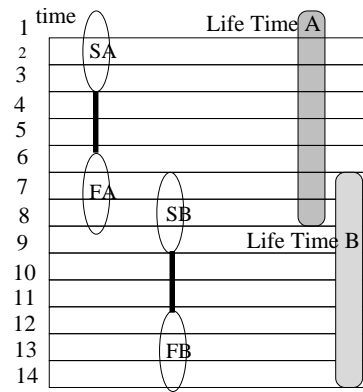


Fig. 6. Relationship between scheduled nodes and life time.

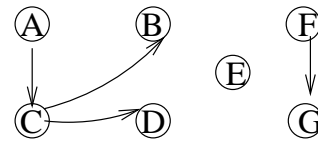


Fig. 7. A simple DFG example.

TABLE I  
FUNCTIONAL UNIT CHARACTERISTICS.

FUs	(lat,poss)		(lat,poss)		(lat,poss)		(lat,poss)	
	lat	poss	lat	poss	lat	poss	lat	poss
FU1,FU3	5	0.05	10	1	15	0.9	23	0.1
FU2,FU4	7	0.5	12	0.7	17	1	29	0.05

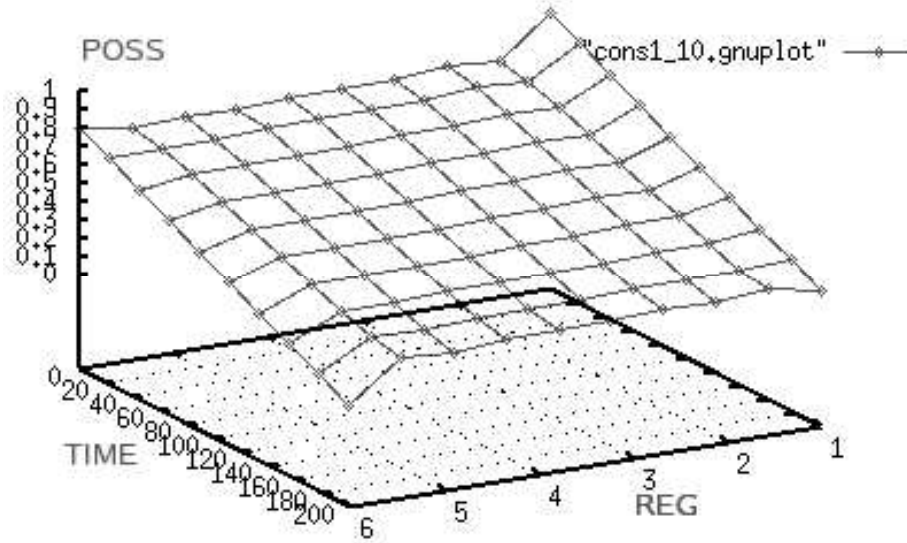


Fig. 8. Constraint for Figure 7.

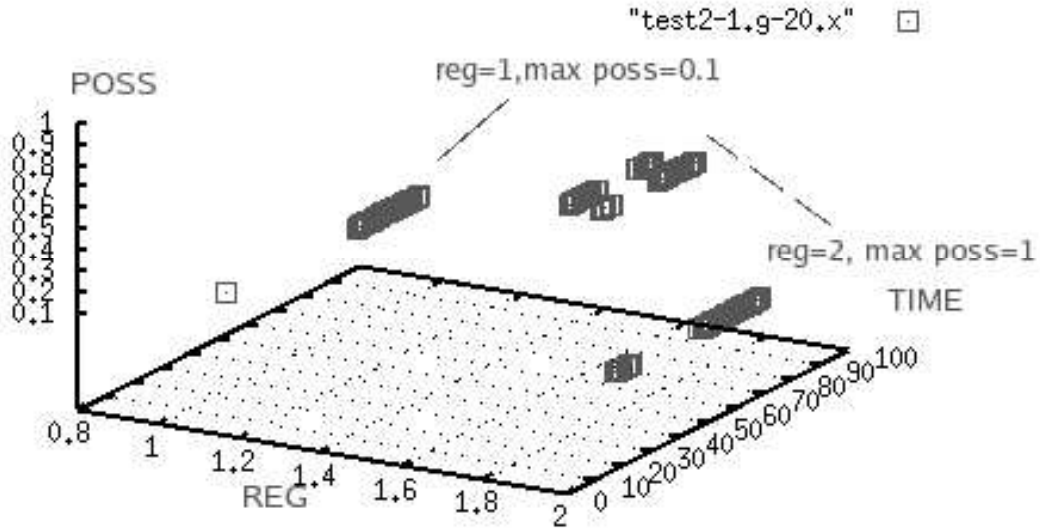


Fig. 11. Register counts and possibility each time step.

TABLE IV

COMPARISON OF RCIS AND IS WHEN VARYING THE NUMBER OF FUNCTIONAL UNITS FOR DISCRETE COSINE TRANSFORM.

	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	7	10	8	10	7	10	7	11
Acceptability	<b>0.719</b>	0.704	0.69	0.69	0.694	0.691	0.699	0.683	0.691	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

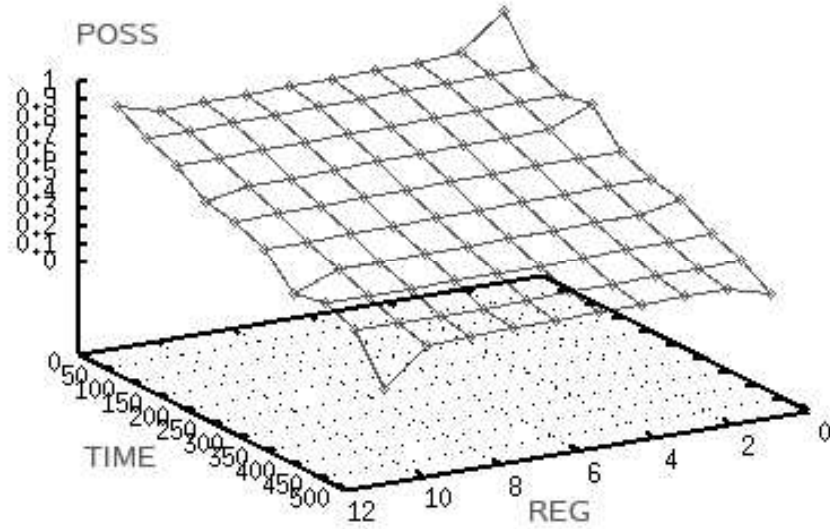


Fig. 12. Constraint for DCT.

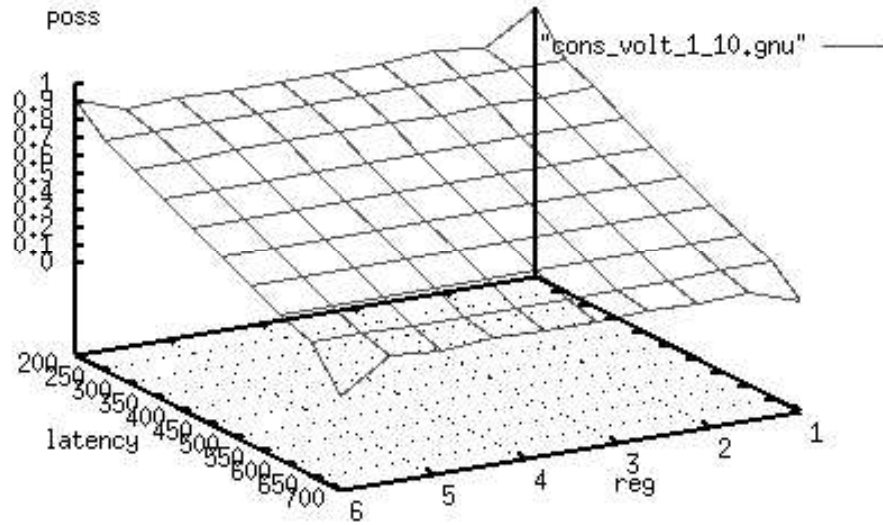


Fig. 13. Constraint for Volterra filter.

TABLE VII  
EXPLORING VARIOUS NUMBER OF FUNCTIONAL UNITS USING RCIS AND IS FOR VOLTERA FILTER.

	1 add 2 muls		1 add 3 muls		1 add 4 muls		1 add 5 muls	
	RCIS	IS	RCIS	IS	RCIS	IS	RCIS	IS
Avg Latency	308	300	267	270	260	260	260	246
Max Reg	2	2	3	3	<b>4</b>	4	4	5
Acceptability	0.78	0.80	0.84	0.84	<b>0.84</b>	0.84	0.84	0.84
Max Latency	561	561	474	477	445	445	445	416

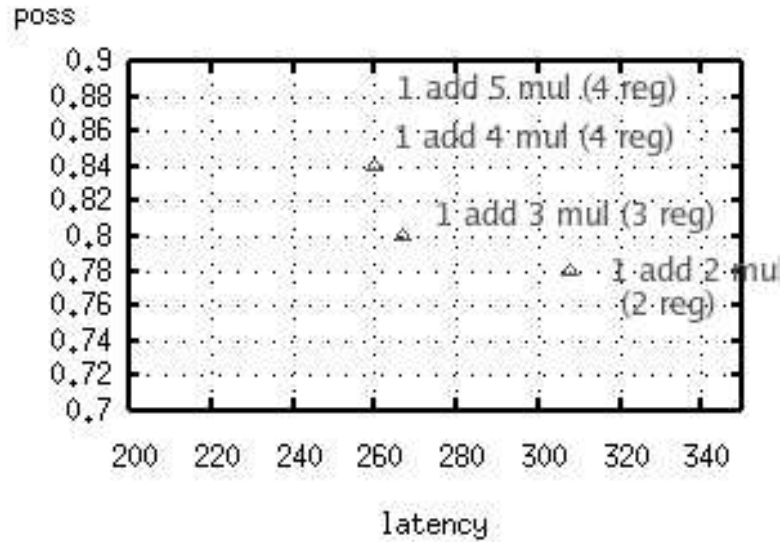


Fig. 14. Acceptability values for each configuration.

TABLE VIII

EXPLORING VARIOUS NUMBER OF FUNCTIONAL UNITS USING RCIS AND IS FOR FAST FOURIER TRANSFORM BENCHMARK.

	5 adds 2 muls		5 adds 3 muls		5 adds 4 muls		6 adds 4 muls	
	RCIS	IS	RCIS	IS	RCIS	IS	RCIS	IS
Avg Latency	134	109	109	107	120	105	120	105
Max Reg	<b>6</b>	7	6	7	6	7	6	7
Acceptability	<b>0.87</b>	0.84	0.87	0.84	0.87	0.84	0.87	0.84
Max Latency	266	220	214	214	225	202	248	202

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

(a)

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

(b)

Fig. 10. (a) Schedule obtained RCIS for Figure 7 (b) Schedule obtained using the original inclusion scheduling.

TABLE II  
SCHEDULE TABLE FROM FIGURE 9(A) TOGETHER WITH IFLT .

cs	FU1	(RegAt(cs),poss)
1	A (1.00)	(1,1.000000)
2	A (1.00)	(1,1.000000)
3	A (1.00)	(1,1.000000)
4	A (1.00)	(1,1.000000)
5	A (1.00)	(1,1.000000)
6	A (1.00) E (0.05)	(1,1.000000)
7	A (1.00) E (0.05)	(1,1.000000)
8	A (1.00) E (0.05)	(1,1.000000)
9	A (1.00) E (0.05)	(1,1.000000)
10	A (1.00) E (1.00)	(1,1.000000)
11	A (0.90) E (1.00)	(1,0.900000)
12	A (0.90) E (1.00)	(1,0.900000)
13	A (0.90) E (1.00)	(1,0.900000)
14	A (0.90) E (1.00)	(1,0.900000)
15	A (0.90) E (1.00)	(1,0.900000)
16	A (0.10) E (1.00)	(1,0.100000)
17	A (0.10) E (1.00)	(1,0.100000)
18	A (0.10) E (1.00)	(1,0.100000)
19	A (0.10) E (1.00)	(1,0.100000)
20	A (0.10) E (1.00)	(1,0.100000)
21	A (0.10) E (1.00)	(1,0.100000)
22	A (0.10) E (1.00)	(1,0.100000)
23	A (0.10) E (1.00)	(1,0.100000)
24	E (1.00)	(1,1.000000)
25	E (0.90)	(1,0.900000)
26	E (0.90)	(1,0.900000)
27	E (0.90)	(1,0.900000)
28	E (0.90)	(1,0.900000)
29	E (0.90)	(1,0.900000)
30	E (0.90)	(1,0.900000)
31	E (0.10)	(1,0.100000)
32	E (0.10)	(1,0.100000)
33	E (0.10)	(1,0.100000)
34	E (0.10)	(1,0.100000)
35	E (0.10)	(1,0.100000)
36	E (0.10)	(1,0.100000)
37	E (0.10)	(1,0.100000)
38	E (0.10)	(1,0.100000)
39	E (0.10)	(1,0.100000)
40	E (0.10)	(1,0.100000)
41	E (0.10)	(1,0.100000)
42	E (0.10)	(1,0.100000)
43	E (0.10)	(1,0.100000)
44	E (0.10)	(1,0.100000)
45	E (0.10)	(1,0.100000)
46	E (0.10)	(1,0.100000)

TABLE VI  
POSSIBILITY VALUES OF REGISTER COUNTS FOR CASE 7 ADDERS AND 5  
MULTIPLIERS FOR IS.

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

TABLE III  
ADDER AND MULTIPLIER CHARACTERISTICS

FUs	(lat,poss)		(lat,poss)		(lat,poss)		(lat,poss)	
	lat	poss	lat	poss	lat	poss	lat	poss
Adder	5	0.05	10	1	15	0.9	23	0.1
Multiplier	7	0.5	12	0.7	17	1	29	0.05

TABLE V  
POSSIBILITY VALUES OF REGISTER COUNTS FOR CASE 7 ADDERS AND 5  
MULTIPLIERS FOR RCIS.

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