IPSJ Transactions on System LSI Design Methodology Vol.7 81–90 (Aug. 2014)

[DOI: 10.2197/ipsjtsldm.7.81]

Regular Paper

A Delay-variation-aware High-level Synthesis Algorithm for RDR Architectures

Yuta Hagio[1,a)] Masao Yanagisawa[2] Nozomu Togawa[1,b)]

Received: December 5, 2013, Revised: March 13, 2014,
Accepted: April 26, 2014, Released: August 4, 2014

Abstract: As device feature size drops, interconnection delays often exceed gate delays. We have to incorporate interconnection delays even in high-level synthesis. Using RDR architectures is one of the effective solutions to this problem. At the same time, process and delay variation also becomes a serious problem which may result in several timing errors. How to deal with this problem is another key issue in high-level synthesis. In this paper, we propose a delay-variation-aware high-level synthesis algorithm for RDR architectures. We first obtain a *non-delayed* scheduling/binding result and, based on it, we also obtain a *delayed* scheduling/binding result. By adding several extra functional units to *vacant* RDR islands, we can have a delayed scheduling/binding result so that its latency is not much increased compared with the non-delayed one. After that, we *similarize* the two scheduling/binding results by repeatedly modifying their results. We can finally realize non-delayed and delayed scheduling/binding results simultaneously on RDR architecture with almost no area/performance overheads and we can select either one of them depending on post-silicon delay variation. Experimental results show that our algorithm successfully reduces delayed scheduling/binding latency by up to 42.9% compared with the conventional approach.

Keywords: process and delay variation, post-silicon tuning, high-level synthesis, distributed-register architecture

1. Introduction

High-level synthesis is one of the important and reliable techniques from the viewpoints of cost and time reduction. High-level synthesis starts with an abstract behavioral specification and finds a register-transfer-level structure that realizes a given behavior.

In the deep submicron era, interconnection delays often exceed gate delays and then we have to incorporate interconnection delays even in high-level synthesis. In Ref. [2], regular-distributed-register architecture (RDR architecture) is proposed, which gives one of the very strong solutions to this problem. RDR architectures divide a chip into uniform-sized islands and arrange functional units, a register file, and a controller in each island. **Figure 1** shows an example of an RDR architecture which has 3×2 islands. It is very easy to predict interconnection delays even in high-level synthesis stage.

At the same time, process and delay variation is also getting a serious problem which may result in several timing errors. Currently we usually assign some amount of timing margins to high-level synthesis based on statistical static timing analysis [1]. A post-silicon circuit tuning approach is proposed such as in Ref. [8] but high-level synthesis taking into account both interconnection delays and delay variation is not proposed.

In this paper, we propose a delay-variation-aware high-level synthesis algorithm for RDR architectures [*1]. We first perform a *non-delayed* scheduling/binding and obtain its scheduling/binding result as well as functional unit floorplanning. Based on its result, we then perform *delayed* scheduling/binding. By adding several extra functional units to *vacant* RDR islands, we have a delayed scheduling/binding result so that its latency is not much increased compared with the non-delayed one. After that, we *similarize* the two scheduling/binding results by repeatedly re-scheduling/re-binding. We can finally realize non-delayed and delayed scheduling/binding results simultaneously on RDR architecture with almost no area/performance overheads and we can select either one of them depending on post-silicon delay variation. Experimental results show that our algorithm successfully reduces delayed scheduling/binding latency by up to 42.9% compared with the conventional approach.

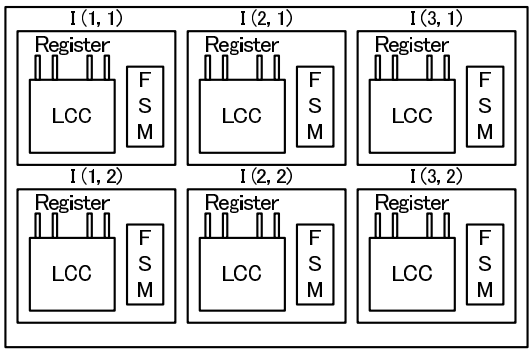


Fig. 1 RDR architecture (3×2).

[1] Department of Computer Science and Engineering, Waseda University, Shinjuku, Tokyo 169–8555, Japan
[2] Department of Electronic and Photonic Systems, Waseda University, Shinjuku, Tokyo 169–8555, Japan
[a)] yuta.hagio@togawa.cs.waseda.ac.jp
[b)] togawa@togawa.cs.waseda.ac.jp

[*1] Preliminary version of this paper appeared in Ref. [4].

This paper is organized as follows: Section 2 introduces an RDR architecture and defines our delay-variation-aware high-level synthesis problem; Section 3 proposes our delay-variation-aware high-level synthesis algorithm for RDR architectures; Section 4 demonstrates several experimental results; Section 5 gives concluding remarks.

2. Problem Definition

In high-level synthesis, a behavioral description is given by a data-flow graph (DFG) as input. A DFG $G = (V, E)$ is represented by a directed graph, where V is a set of operation nodes and E is a set of edges which denote data dependencies. Hereafter, we use a DFG as input for simplicity. Note that the discussion can also be extended to a control-data flow graph.

In this section, we introduce an RDR architecture [2] and define our delay-variation-aware high-level synthesis problem.

2.1 RDR Architecture

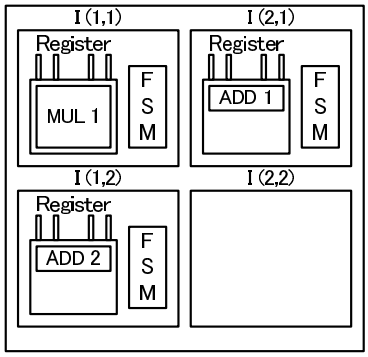
The RDR architecture divides the entire chip into $N \times M$ array of islands as shown in **Fig. 2** (a). Let $I(x, y)$ be the island on the position (x, y) of the array, where $1 \le x \le N$ and $1 \le y \le M$. Every island is assumed to be in a square shape. A functional unit (FU) fu is allocated to one of the islands and has a delay of d_{fu}. Each island $I(x, y)$ has the local register $R(I(x, y))$. RDR island is designed so that interconnection delays inside each island can be ignored.

We use the interconnection delay model used in Ref. [5], [7] and an interconnection delay $D_c(i_1, i_2)$ between the two islands i_1 and i_2 is proportional to the square of their distance and given by

$$D_c(i_1, i_2) = C_d \times (|x_1 - x_2| + |y_1 - y_2|)^2 \quad (1)$$

where C_d shows the constant interconnection delay coefficient.

Our algorithm can also use another interconnection delay model, e.g., an interconnection delay model where wire delay is proportional to the Manhattan distance, because the essential part of our interconnection delay model is given by the data-transfer table. Once just the data-transfer table is constructed, we may use any interconnection delay model.



(a) FU floorplanning in RDR.

	ADD 1	ADD 2	MUL 1
ADD 1	0	1	0
ADD 2	1	0	0
MUL 1	0	0	0

(b) *Non-delayed* data-transfer table.

	ADD 1	ADD 2	MUL 1
ADD 1	0	2	0
ADD 2	2	0	0
MUL 1	0	0	0

(c) *Delayed* data-transfer table.

Fig. 2 Data-transfer table.

Let fu_1 be one of the FUs allocated to the island i_1. Assume that the output of fu_1 is used by the island i_2. Let T_{clk} be the given clock period and we assume $d_{fu_1} < T_{clk}$. If $T_{clk} \ge d_{fu_1} + D_c(i_1, i_2)$, executing fu_1 and storing its output into the register $R(i_2)$ are done in a single control step. On the other hand, if $T_{clk} < d_{fu_1} + D_c(i_1, i_2)$ holds, we only execute the fu_1 operation and store its output into the register $R(i_1)$ at the first control step. After that, we transfer the output of fu_1 from $R(i_1)$ to $R(i_2)$ using

$$\left\lceil \frac{D_c(i_i, i_2)}{T_{clk}} \right\rceil \quad (2)$$

control steps. Similarly we can also define data-transfer control steps when $d_{fu_1} \ge T_{clk}$.

Every island has the capacity C and every FU fu has the capacity cost c_{fu}. Let $FU(i)$ be a set of FUs allocated to the island $i = I(x, y)$. Any island i satisfies:

$$C \ge \sum_{fu \in Fu(i)} c_{fu}. \quad (3)$$

Equation (3) is called a *capacity constraint*.

In RDR architecture, we can predict interconnection delays between any two FUs very accurately even in high-level synthesis since they can be calculated by the FU positions in RDR islands.

2.2 Data-transfer Table

When we obtain an FU floorplanning result for a given RDR architecture, we can construct a *data-transfer table* which shows how many control steps every data transfer requires between any two FUs allocated to the RDR architecture. Consider the data transfer between two FUs fu_1 and fu_2. Assume that fu_1 is allocated to the island i_1 and fu_2 is allocated to to the island i_2. We further assume $d_{fu_1} < T_{clk}$. Based on the previous discussion, if $T_{clk} \ge d_{fu_1} + D_c(i_1, i_2)$, then the table value is zero, i.e, we do not need control steps for data transfer. If $T_{clk} < d_{fu_1} + D_c(i_1, i_2)$, the table value is given by Eq. (2), i.e., we need control steps only for data transfer. Similarly we can also define a table value when $d_{fu_1} \ge T_{clk}$.

Figure 2 (b) shows an example of a data-transfer table based on the FU floorplanning shown in Fig. 2 (a). As shown here, a data transfer from ADD2 in $I(1, 2)$ to ADD1 in $I(2, 1)$ requires one control step. This means that, when we execute ADD2 and ADD1 in this order, the execution of ADD2 operation and data transfer from the ADD2 output to the ADD1 input cannot be done in a single clock cycle. Data transfer from the ADD2 output to the ADD1 input requires one clock cycle since $I(1, 2)$ and $I(2, 1)$ are apart from each other.

2.3 Process/Delay Variations in RDR Architectures and Problem Definition

Process and delay variation may occur in any wire and interconnection as well as logic gates but we must consume too much area and power if we cope with all possible delay variations in high-level synthesis, which we consider must be almost impossible. An approach dealing with delay variations in specific interconnections is one of the feasible solutions to this problem.

Logic delay variation is also very important. But, we only deal with the simplest delay variation model in this paper and then we do not deal with logic delay variation here. Extending our algorithm so as to tackle logic delay variations is one of our very important future works.

This research work is the first step of delay-variation-aware high-level synthesis and thus we only deal with the simplest delay variation model in this paper. Using RDR architectures, we can estimate the interconnection length very easily in high-level synthesis stage and thus focusing on interconnection between RDR islands is one of the simplest targets. A long interconnection must have a large number of contact holes or vias, which has relatively high resistance and capacity [9]. This means that they can affect largely delay variations.

Based on the above discussion, we focus only on the *longest* interconnections and deal with their delay variations here. Delay variation focused on in this paper is defined as follows:

Definition 1. *Delay variation is assumed to occur in the longest interconnections in the target RDR architecture. If the delay variation occurs there, we require one more control step than expected for data transfer when using these interconnections.* □

There may be a situation where an extra cycle is necessary in data transfer other than the longest interconnection with large delay variation. However, as we motioned earlier, we only focus here on the longest interconnection simply because this research work is the first step of delay-variation-aware high-level synthesis. How many extra cycles are required to accommodate delay variation in the longest interconnections is another concern. There may be several discussions but we simply assume that we require one extra clock cycle when delay variation occurs in the longest interconnection.

Let us consider FU floorplanning as shown in Fig. 2 (a). The longest interconnections are the one between $I(1,1)$ and $I(2,2)$ and the one between $I(2,1)$ and $I(1,2)$. If the delay variation occurs there, a data-transfer table is modified from Fig. 2 (b) to Fig. 2 (c). As shown in Fig. 2 (c), the data transfer from ADD2 in $I(1,2)$ to ADD1 in $I(2,1)$ requires two control steps, which is increased by one as compared to Fig. 2 (b). Figure 2 (b) is called a *non-delayed* data-transfer table and Fig. 2 (c) is called a *delayed* data-transfer table.

Now we define a delay-variation-aware high-level synthesis problem for RDR architectures as follows:

Definition 2. *For a given DFG $G = (V, E)$, an RDR architecture, a capacity constraint C, FU library, and clock period constraint T_{clk}, our delay-variation-aware high-level synthesis problem for the RDR architecture is to arrange FUs to the RDR islands and to assign each operation node to a control step and a functional unit with and without considering delay variation. The objective is to minimize the latency.* □

In our high-level synthesis problem above, we map an input DFG based on a non-delayed data-transfer table onto RDR islands (e.g., Figs. 7 (a) and 13 (a)) and also we map the same input DFG based on a delayed data-transfer table onto the same RDR islands simultaneously (e.g., Figs. 7 (a) and 13 (b)). Then we select either one of the two mapping results depending on post-silicon delay variation. Detailed discussions how to obtain these results will be explained in the next section.

3. A Delay-variation-aware High-level Synthesis Algorithm for RDR Architectures

One of the simplest ways to cope with the delay variation given by Definition 1 is that, every data transfer using the longest interconnections always require one more control step in high-level synthesis. However, this approach is too pessimistic and always requires extra clock cycles even when delay variation does not occur. How to deal with non-delayed scheduling/binding as well as delayed scheduling/binding simultaneously is the important key and we must consider the following two points:

(1) We can have a minimized non-delayed scheduling/binding latency using the conventional approach as in Ref. [2]. We also have to minimize a delayed scheduling/binding latency.

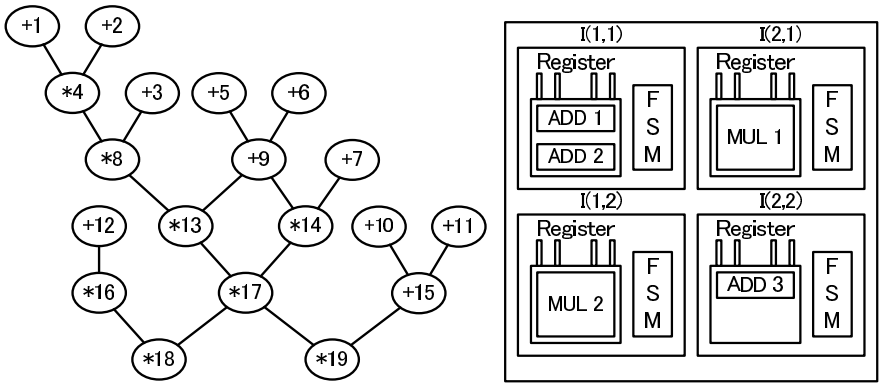
(2) If we schedule/bind a given DFG using non-delayed data-transfer table and delayed one independently, we will require too many RDR resources, such as controllers and MUXs. We have to modify the two scheduling/binding results so that they can be similar to each other and share as many RDR resources as possible (which is called *similarization*).

Since high-level synthesis for RDR architectures is composed of scheduling, binding, FU floorplanning and FU assignment, we consider each of these processes to give an efficient solution to Points (1) and (2) as below:

- **Scheduling/Binding**
  We first perform scheduling/binding using the conventional RDR synthesis algorithm, MCAS [2]. This result is called a *non-delayed scheduling/binding result*. MCAS can obtain a good result in terms of latency. In this step, we also have its associated FU floorplanning in RDR islands. Using this FU floorplanning, we next perform scheduling/binding based on a delayed data-transfer table. This result is called a *delayed scheduling/binding result*. In this scheduling/binding, we also use MCAS to obtain a latency-optimized result without considering similarization of the two scheduling/binding results.

  After that, we similarize the two scheduling/binding results. This is done by repeating re-scheduling/re-binding based on non-delayed and delayed data-transfer tables several times.
- **Floorplan/Allocation**
  We do not change the original FU floorplanning in RDR islands obtained at non-delayed scheduling and binding, since changing it may increase the non-delayed scheduling/binding latency. However, RDR architectures have regular structure and often have *vacant* islands. Then we allocate several extra functional units to vacant RDR islands so that we can minimize the delayed scheduling/binding latency.

Based on the above strategy, we propose a delay-variation-aware high-level synthesis algorithm for RDR architectures. **Figure 3** shows our synthesis flow. The initial step (Step 0: non-delayed scheduling/binding) can be performed by just using MCAS. Then our algorithm is mainly composed of the two steps: delayed scheduling/binding (Step 1) and similarization of non-delayed and delayed scheduling/binding results (Step 2).



(a) Input DFG. (b) FU floorplanning.

Fig. 4 Input DFG and its FU floorplanning.

	ADD 1	ADD 2	ADD 3	MUL 1	MUL 2
ADD 1	0	0	0	0	0
ADD 2	0	0	0	0	0
ADD 3	0	0	0	0	0
MUL 1	0	0	0	0	0
MUL 2	0	0	0	0	0

(a) Non-delayed data-transfer table.

	ADD 1	ADD 2	ADD 3	MUL 1	MUL 2
ADD 1	0	0	1	0	0
ADD 2	0	0	1	0	0
ADD 3	1	1	0	0	0
MUL 1	0	0	0	0	1
MUL 2	0	0	0	1	0

(b) Delayed data-transfer table.

Fig. 5 Data-transfer table.

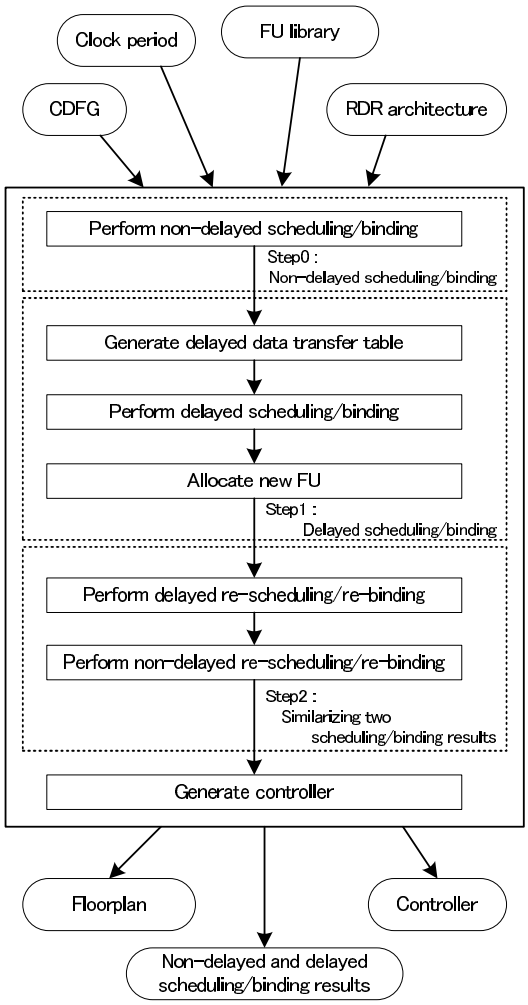


Fig. 3 The synthesis flow.

3.1 Delayed Scheduling/Binding (Step 1)

Step 1 generates a delayed scheduling/binding result. Step 1 is composed of:

Step (1.1): Generate a delayed data-transfer table

Step (1.2): Perform delayed scheduling/binding

Step (1.3): Allocate new FUs.

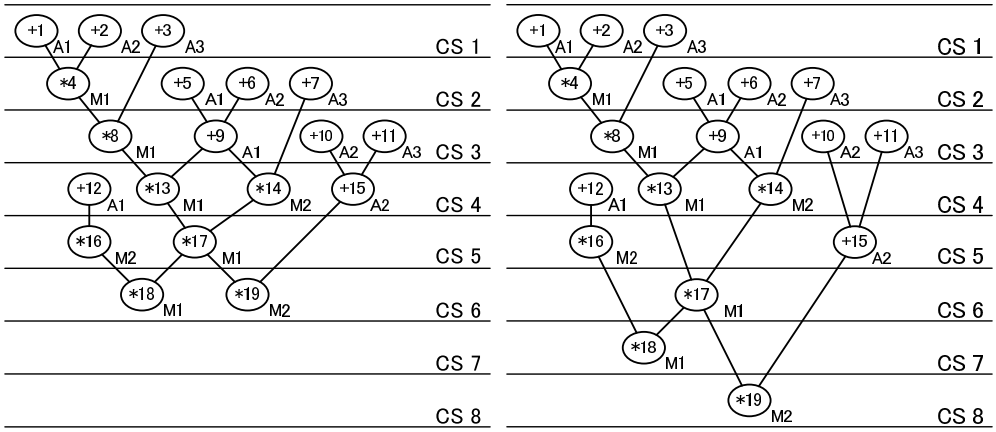
In Step (1.1), we generate a delayed data-transfer table based on the non-delayed one. The non-delayed data-transfer table here can be obtained by FU floorplanning given by Step 0. In the delayed data-transfer table, every data transfer using the longest interconnections requires one more control step than those in the non-delayed one.

In Step (1.2), we perform scheduling/binding based on the delayed data-transfer table. In this step, we use initial FU floorplanning and do not change it, since we try to similarize the two scheduling/binding results in the later step.

In Step (1.3), we allocate new FUs so that we can minimize the delayed scheduling/binding latency. In this step, we try all the possible patterns of allocating a new FU into a vacant RDR island, one by one, and perform scheduling/binding for each of them. Then we accept the new FU giving the minimum delayed scheduling/binding latency. We repeat this step until no further vacant island exists nor further latency improvement can be seen.

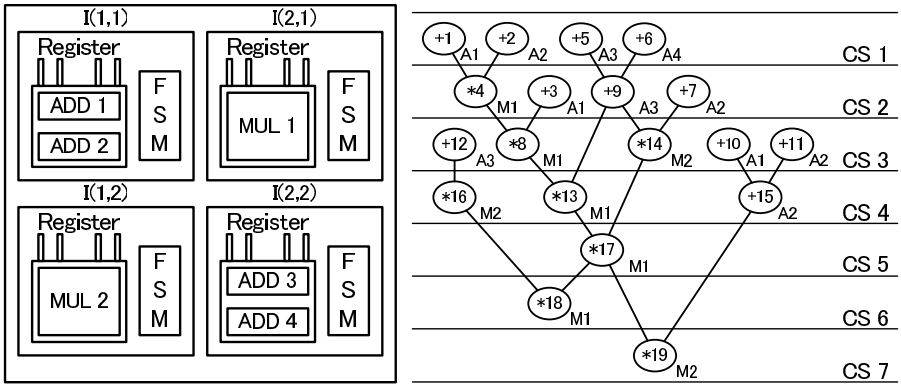
Example 1. *Let us consider a DFG as depicted in* ***Fig. 4*** *(a). Assume that the RDR architecture has* 2×2 *islands. In Step 0, FUs are placed as in Fig. 4 (b) and the non-delayed data-transfer table as in* ***Fig. 5*** *(a) can be obtained.* ***Figure 6*** *(a) shows the non-delayed scheduling/binding result. In this example, all the operations can be executed in one control step.*

We generate a delayed data-transfer table as shown in Fig. 5 (b) based on the non-delayed one in Fig. 5 (a). Then we perform scheduling/binding based on Fig. 5 (b). Here we also use MCAS. Figure 6 (b) shows the delayed scheduling/binding result.



(a) Non-delayed scheduling/binding. (b) Delayed scheduling/binding.

Fig. 6 Scheduling/binding result.



(a) The layout. (b) Delayed scheduling/binding result.

	ADD 1	ADD 2	ADD 3	ADD4	MUL 1	MUL 2
ADD 1	0	0	1	1	0	0
ADD 2	0	0	1	1	0	0
ADD 3	1	1	0	0	0	0
ADD 4	1	1	0	0	0	0
MUL 1	0	0	0	0	0	1
MUL 2	0	0	0	0	1	0

(c) Delayed data-transfer table.

Fig. 7 After allocating a new FU.

After that, we allocate additional new FUs to vacant RDR islands to minimize the delayed scheduling/binding latency, in a one-by-one manner. ***Figure 7*** *(a) shows the FU floorplanning after allocating the new FU, ADD4, to* $I(2,2)$*. Figures 7 (c) and 7 (b) show its updated delayed data-transfer table and updated delayed scheduling/binding result, respectively.* □

3.2 Similarizing Two Scheduling/Binding Results (Step 2)

In Step 2, we similarize the two scheduling/binding results; non-delayed one and delayed one. By similarizing their results, control signals for registers and FUs can be shared and then the controller size and the number of MUXs can be decreased.

Let $v \in V$ be a node in the input DFG $G = (V, E)$. Let $S(v)$ and $B(v)$ be the non-delayed control step assigned to v and its FU bound to v, respectively. Similarly, let $S_D(v)$ and $B_D(v)$ be the delayed control step assigned to v and its FU bound to v, respectively. Similarizing the two scheduling/binding results is to maximize the number of such nodes as $S(v) = S_D(v)$ and $B(v) = B_D(v)$.

Step 2 is composed of:

Step (2.1): Perform delayed re-scheduling and re-binding

Step (2.2): Perform non-delayed re-scheduling and re-binding

3.2.1 Step (2.1): Delayed Re-scheduling and Re-binding

In Step (2.1), we modify the delayed scheduling/binding result. In this step, we do not change $S(v)$ and $B(v)$ but we modify $S_D(v)$ and $B_D(v)$ so that the non-delayed and delayed scheduling/binding results can be similarized. We use the delayed data-transfer table when we perform delayed re-scheduling/re-binding.

Our re-scheduling algorithm of Step (2.1) is as follows:

(RS1) For every node $v \in V$ from the top to the bottom, if $S_D(v) = S(v)$, we fix $S_D(v)$ to $S(v)$ and unchange it hereafter.

(RS2) Repeat (RS3)–(RS5) below until no further delayed re-scheduling can be performed.

(RS3) For every node $v \in V$ from the top to the bottom, if

$S_D(v) \neq S(v)$ and $S_D(v)$ can be assigned to $S(v)$, we re-assign $S_D(v)$ to $S(v)$ and unchange it hereafter. We repeat this step several times from the top to the bottom until no further re-assignment is done.

(RS4) For every node $v \in V$ from the bottom to the top, if $S_D(v) \neq S(v)$ and $S_D(v)$ can be assigned to $S_D(v)+1$ without modifying other nodes' scheduling and without increasing the latency, we re-assign $S_D(v)$ to $S_D(v)+1$. At that time, if $S_D(v)$ becomes $S(v)$, we fix $S_D(v)$ to $S(v)$ and unchange it hereafter. We repeat this step several times from the bottom to the top until no further re-assignment is done.

(RS5) For every node $v \in V$ from the top to the bottom, if $S_D(v) \neq S(v)$ and $S_D(v)$ can be assigned to $S_D(v)-1$ without modifying other scheduling, we assign $S_D(v)$ to $S_D(v)-1$. At that time, if $S_D(v)$ becomes $S(v)$, we fix $S_D(v)$ to $S(v)$ and unchange it hereafter. We repeat this step several times from the top to the bottom until no further re-assignment is done.

We iterate Steps (RS3)–(RS5) until no further improvement is done. Stopping criteria is as follows: Let $n_{fixedS_D}(i)$ be the number of the nodes whose delayed scheduling is fixed at the iteration i. If $n_{fixedS_D}(i-1) < n_{fixedS_D}(i)$, then we perform the $(i+1)$-th iteration. If not, we stop the iteration. Finally, we expect that we can similarize the delayed scheduling result to the non-delayed one.

Example 2. *In Step (RS3), we try to re-assign $S_D(v)$ to $S(v)$. Let us consider the two scheduling/binding results as shown in **Fig. 8**. In this figure, Node "+1" and "+2" are executed by the FU "A1," and Node "+3" is executed by the FU "A2." **Figure 9** show the two data-transfer tables corresponding to Fig. 8. Data transfer between "A1" and "A2" needs one control step when delay variations occur. In this case, Node "+2" of the delayed scheduling result is re-assigned to CS2 and fixed to it. Node "+3" of the delayed scheduling result cannot be re-assigned to CS2 and remains at CS3 since we require one control step from "A1" to "A2" in the delayed scheduling.* □

Example 3. *Let us consider the two scheduling/binding results, delayed and non-delayed, as shown in **Figs. 10** (a) and 10 (b). We now assume that Node "+3" in Fig. 10 (b) cannot be assigned to CS5, because the FU "A1" is used by another operation and cannot be used. In Step (RS4), we re-assign $S_D(v)$ to $S_D(v)+1$, one by one, repeatedly. In this case, Node "+3" is firstly re-assigned to CS4. After that, Node "+2" is re-assigned to CS3, where we have S_D("+2") = S("+2"). Finally, we have the delayed scheduling result as in Fig. 10 (c).* □

Our re-binding algorithm of Step (2.1) is as follows:

(RB1) For every node $v \in V$ from the top to the bottom, if $B_D(v) = B(v)$, we fix $B_D(v)$ to $B(v)$ and unchange it hereafter.

(RB2) Repeat (RB3)–(RB4) until no further re-binding can be performed.

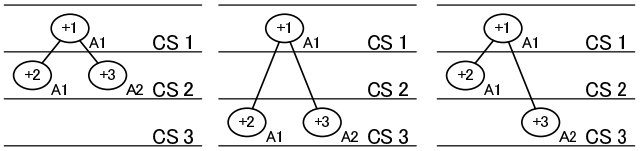
(RB3) For every node $v \in V$ from the top to the bottom, if $B_D(v) \neq B(v)$ and $B_D(v)$ can be bound to $B(v)$ without modifying other nodes' binding, we bind $B_D(v)$ to $B(v)$ and unchange it hereafter. We repeat this step several times from the top to the bottom until no further re-binding is done.

(RB4) For every node $v \in V$ from the top to the bottom, if $B_D(v) \neq B(v)$ and there exists another node $w \in V$ such that $B_D(w) = B(v)$ and $B_D(v)$ can be swapped for $B_D(w)$, we swap $B_D(v)$ for $B_D(w)$. At that time, we fix $B_D(v)$ to $B(v)$ and unchange it hereafter. We repeat this step several times from the top to the bottom until no further swap is done.

We iterate Steps (RB3)–(RB4) until no further improvement is done. Stopping criteria is as follows: Let $n_{fixedB_D}(i)$ be the number of the nodes whose delayed binding is fixed at the iteration i. If $n_{fixedB_D}(i-1) < n_{fixedB_D}(i)$, then we perform the $(i+1)$-th iteration. If not, we stop the iteration. Finally, we expect that we can similarize the delayed binding result to the non-delayed one.

Example 4. *In Step (RB3), we try to bind $B_D(v)$ to $B(v)$. Let us consider the two scheduling/binding results as shown in **Fig. 11**. In Fig. 11 (a), Nodes "+1" and "+3" are bound to the FU "A1" and Node "+2" is bound to the FU "A2." In Fig. 11 (b), Nodes "+1", "+2" and "+3" are bound to the FU "A1." In this case, Node "+2" in Fig. 11 (b) can be assigned to the FU "A2" if it is not used there. Then, in this case, we re-bind Node "+2" to the FU "A2" in Step (RB3).* □

Example 5. *Let us consider the two scheduling/binding results as shown in **Fig. 12**. In this case, we cannot directly re-bind Node "+1" in Fig. 12 (b) to the FU "A2" since it is used by Node "+2." In Step (RB4), we try to swap $B_D(v)$ for $B_D(w)$ and, in this case,*



(a) Non-delayed. (b) Delayed. (c) Delayed (modified).

Fig. 8 An example of Step (RS3).

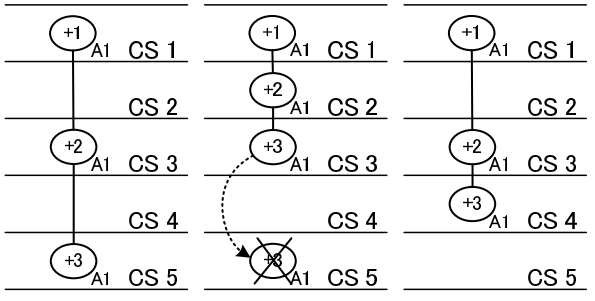
	A1 (ADD 1)	A2 (ADD 2)
A1 (ADD 1)	0	0
A2 (ADD 2)	0	0

(a) Non-delayed.

	A1 (ADD 1)	A2 (ADD 2)
A1 (ADD 1)	0	1
A2 (ADD 2)	1	0

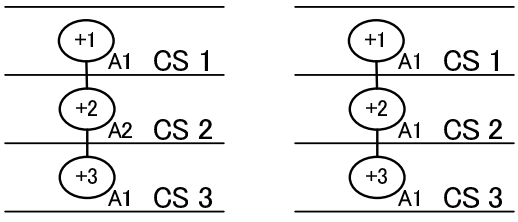
(b) Delayed.

Fig. 9 Data-transfer tables of Fig. 8.



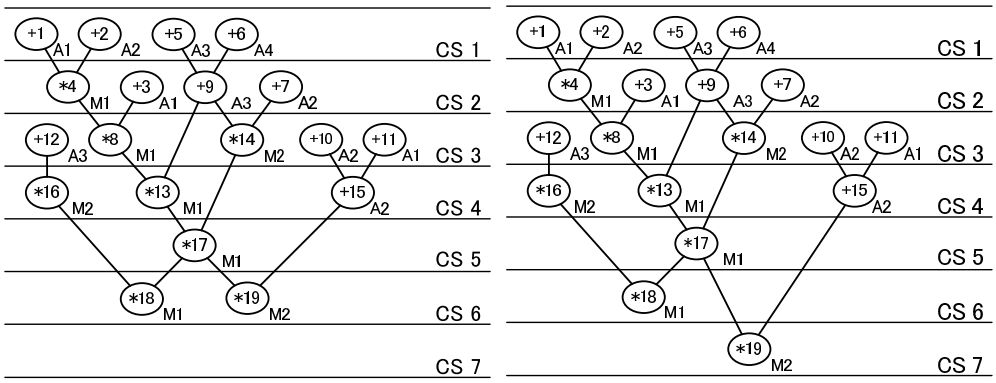
(a) Non-delayed. (b) Delayed. (c) Delayed (modified).

Fig. 10 An example of Step (RS4).



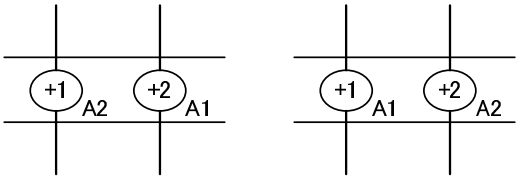
(a) Non-delayed biding. (b) Delayed binding.

Fig. 11 An example of Step (RB3).



(a) Non-delayed scheduling/binding. (b) Delayed scheduling/binding.

Fig. 13 Re-scheduling/re-binding results.



(a) Non-delayed binding. (b) Delayed binding.

Fig. 12 An example of Step (RB4).

we swap B_D("+1") for B_D("+2"). We can have the same binding result for non-delayed one and delayed one. □

We repeatedly perform these re-scheduling and re-binding several times until no further re-scheduling and re-binding can be done for delayed scheduling/binding results.

3.2.2 Step (2.2): Non-delayed Re-scheduling and Re-binding

In Step (2.2), we modify the non-delayed re-scheduling/re-binding result. In this step, we do not change $S_D(v)$ and $B_D(v)$ but we modify $S(v)$ and $B(v)$ so that the non-delayed and delayed scheduling/binding results can be similarized. We use the same algorithm described in the previous section. We use the non-delayed data-transfer table when we perform non-delayed re-scheduling/re-binding.

Example 6. *After our re-scheduling/re-binding steps are applied to the two scheduling/binding results shown as in Figs. 6 (a) and 7 (b), we have* ***Figs. 13*** *(a) and 13 (b). As can be seen in these figures, they are similar to each other.* □

Finally, we generate controllers. Then we can select either one of the two scheduling/binding results depending on post-silicon delay variation.

4. Experimental Results

We have implemented our algorithm in C++. We applied our algorithm to DCT (48 nodes), EWF3 (102 nodes), FIR (75 nodes), and PARKER (22 nodes, including conditional branches). All the FUs are assumed to have 16-bit width under the 90 nm technology node. We set the clock period to be $T_{clk} = 3.0$ ns and the capacity to be $C = 2$. According to Ref. [7], we set adder cost to be 1, subtracter cost to be 1, multiplier cost to be 2, and comparator cost to be 1. We also set the interconnection delay coefficient to be $C_d = 1.0$ ns. We used Synopsys Design Compiler for controller synthesis.

We have compared our algorithm with the following four algorithms:

Original MCAS: We have run the original MCAS [2]. It only deals with a non-delayed data-transfer table and then does not cope with the delay variation.

Modified MCAS (1): One of the simplest ways to cope with the delay variation given by Definition 1 is that, every data transfer using the longest interconnections always requires one more control step in MCAS, i.e., we always consider a delayed data-transfer table. In this case, we always have a delayed scheduling/binding result but it can satisfy the delay variation given by Definition 1. Note that, in this case we allocate a new FU which is the same as our proposed algorithm to the same island position to compare it fairly with our proposed algorithm.

Modified MCAS (2): Another simple way to cope with the delay variation given by Definition 1 is that, we assume to use error-detecting flipflops such as Refs. [3], [6] as RDR local registers and wait for correct data to arrive its destination for one clock cycle if delay variation occurs. In this case, we can always use a non-delayed scheduling/binding result generated by the original MCAS and, if delay variation occurs, we wait for the data to arrive at its destination and stall all the working operations for one clock cycle. How to implement error-detecting flipflops is another concern and then we just hand-calculate how many clock cycles are required when delay variation occurs in the longest interconnections. We performed our experiment "Modified MCAS (2)" changing delay variation probability (25%, 50%, 75%, and 100%). For example, if delay variation probability is 25%, we require an extra clock cycle in the longest interconnection at the probability of 25%. We performed 100 trials in this experiment and obtained average control step counts.

Our algorithm without Step 2 (Ours w/o Step 2): In order to evaluate the effectiveness of our similarization step, we have run our algorithm without Step 2.

Table 1 shows the experimental results. In this table, non-delay control steps refer to the number of required control steps to schedule/bind a non-delayed control-data flow graph. Similarly, delayed control steps refer to the number of required control steps to schedule/bind a delayed control-data flow graph. Since every RDR island has the same size and it is determined by the

Table 1 Experimental results.

App.	Islands	FUs	Algorithm	Non-delayed control steps	Delayed control steps	Allocated new FUs	Area [μm^2]	CPU time [sec]
DCT	2×2	ADD×4 MUL×2	MCAS [2]	11	–	NA	7,194	99.3
			Modified MCAS (1)	12	12	–	6,901	99.8
			Modified MCAS (2) [3], [6] (Prob=25%)	11	15.46	NA	–	–
			Modified MCAS (2) [3], [6] (Prob=50%)	11	18.06	NA	–	–
			Modified MCAS (2) [3], [6] (Prob=75%)	11	19.82	NA	–	–
			Modified MCAS (2) [3], [6] (Prob=100%)	11	21	NA	–	–
			Ours w/o Step2	11	12	–	7,239	100.1
			Ours	11	12	–	6,930	100.5
EWF3	2×2	ADD×3 MUL×2	MCAS [2]	52	–	NA	9,897	106.3
			Modified MCAS (1)	54	54	–	9,908	106.1
			Modified MCAS (2) [3], [6] (Prob=25%)	52	55.82	NA	–	–
			Modified MCAS (2) [3], [6] (Prob=50%)	52	59.02	NA	–	–
			Modified MCAS (2) [3], [6] (Prob=75%)	52	61.76	NA	–	–
			Modified MCAS (2) [3], [6] (Prob=100%)	52	64	NA	–	–
			Ours w/o Step2	52	54	–	10,087	108.3
			Ours	52	54	–	9,315	109.0
FIR	3×2	ADD×3 MUL×3	MCAS [2]	19	–	NA	6,427	123.9
			Modified MCAS (1)	19	19	–	6,427	125.2
			Modified MCAS (2) [3], [6] (Prob=100%)	19	19	NA	–	–
			Ours w/o Step2	19	19	–	6,427	124.2
			Ours	19	19	–	6,427	124.4
PARKER	2×2	ADD×2 SUB×2 COMP×1	MCAS [2]	10	–	NA	3,479	55.3
			Modified MCAS (1)	10	10	ADD×1	3,467	55.0
			Modified MCAS (2) [3], [6] (Prob=25%)	10	10.26	NA	–	–
			Modified MCAS (2) [3], [6] (Prob=50%)	10	10.55	NA	–	–
			Modified MCAS (2) [3], [6] (Prob=75%)	10	10.68	NA	–	–
			Modified MCAS (2) [3], [6] (Prob=100%)	10	11	NA	–	–
			Ours w/o Step2	10	10	ADD×1	3,467	57.5
			Ours	10	10	ADD×1	3,467	57.9

Table 2 The configuration in each RDR island (DCT).

Algorithm	Island	FUs	Controller area [μm^2]	MUXs	Registers	Area [μm^2]
MCAS	**(1, 1)**	**MUL**	**309**	**7**	**5**	**7,194**
	(2, 1)	MUL	301	5	4	6,674
	(1, 2)	ADD×2	630	24	6	5,610
	(2, 2)	ADD×2	583	25	7	5,963
Modified MCAS (1)	**(1, 1)**	**MUL**	**304**	**7**	**4**	**6,901**
	(2, 1)	MUL	288	7	4	6,773
	(1, 2)	ADD×2	704	29	6	6,244
	(2, 2)	ADD×2	536	21	7	5,468
Ours w/o Step 2	**(1, 1)**	**MUL**	**354**	**7**	**5**	**7,239**
	(2, 1)	MUL	346	5	4	6,719
	(1, 2)	ADD×2	659	24	6	5,639
	(2, 2)	ADD×2	613	24	7	5,881
Ours	**(1, 1)**	**MUL**	**333**	**7**	**4**	**6,930**
	(2, 1)	MUL	328	6	4	6,701
	(1, 2)	ADD×2	674	24	6	5,654
	(2, 2)	ADD×2	575	20	7	5,395

maximum-sized RDR island, we also show maximum-sized RDR island area at the column of "Area." CPU time shows the run time to perform each algorithm.

Our proposed algorithm successfully reduced delayed scheduling/binding latency by up to 42.9% (from 21 steps to 12 steps in DCT) compared with Modified MCAS (2). In most of the cases, our algorithm can decrease the control step count with almost no area and CPU time overhead. Further, our algorithm increases the control step count in none of the cases. Our proposed algorithm reduced the maximum RDR island area by up to 7.7% compared with ours w/o Step 2. Similarization step is important for our algorithm to optimize the area. Consequently, our proposed algorithm gives one of the most effective and feasible solutions to the delay variation in high-level synthesis.

In order to show how our algorithm affects maximum island size, we also show the configuration in each RDR island for our application programs used in this experiment in **Tables 2**, **3**, **4**, and **5**, where every row written in bold face shows the maximum-sized island. As in these tables, maximum-sized island area obtained by our proposed algorithm is almost equal to the ones obtained by MCAS and Modified MCAS (1) while it is reduced by up to 7.7% compared with ours w/o Step 2 (from 10,087 μm^2 to 9,315 μm^2 in EWF3). Note that the number of registers used in our proposed algorithm is slightly decreased compared with MCAS and Modified MCAS (1) in DCT and EWF3. The maximum-sized island is changed from $I(2,1)$ to $I(1,1)$ in PARKER. This is just because our proposed algorithm re-binds FUs and thus register assignment is changed.

Since our algorithm allocates new FUs into a vacant island, i.e., non-maximum-sized island, adding a new FU into an island

Table 3 The configuration in each RDR island (EWF3).

Algorithm	Island	FUs	Controller area [μm^2]	MUXs	Registers	Area [μm^2]
MCAS	(1, 1)	ADD	527	8	3	2,569
	(2, 1)	MUL	454	3	1	5,627
	(1, 2)	MUL	503	3	2	6,076
	(2, 2)	**ADD×2**	**1,477**	**47**	**9**	**9,897**
Modified MCAS (1)	(1, 1)	ADD	433	7	2	2,075
	(2, 1)	MUL	340	1	1	5,289
	(1, 2)	MUL	372	3	2	5,833
	(2, 2)	**ADD×2**	**1,488**	**47**	**9**	**9,908**
Ours w/o Step 2	(1, 1)	ADD	604	7	3	2,534
	(2, 1)	MUL	449	3	1	5,622
	(1, 2)	MUL	489	2	2	5,950
	(2, 2)	**ADD×2**	**1,555**	**48**	**9**	**10,087**
Ours	(1, 1)	ADD	567	8	3	2,609
	(2, 1)	MUL	452	4	1	5,737
	(1, 2)	MUL	522	3	2	6,095
	(2, 2)	**ADD×2**	**1,519**	**44**	**8**	**9,315**

Table 4 The configuration in each RDR island (FIR).

Algorithm	Island	FUs	Controller area [μm^2]	MUXs	Registers	Area [μm^2]
MCAS	**(1, 1)**	**MUL**	**342**	**6**	**3**	**6,427**
	(2, 1)	ADD	589	19	7	4,792
	(3, 1)	MUL	348	5	3	6,321
	(1, 2)	MUL	325	4	3	6,186
	(2, 2)	ADD×2	690	28	5	5,830
	(3, 2)	–	0	0	0	0
Modified MCAS (1)	**(1, 1)**	**MUL**	**342**	**6**	**3**	**6,427**
	(2, 1)	ADD	589	19	7	4,792
	(3, 1)	MUL	348	5	3	6,321
	(1, 2)	MUL	325	4	3	6,186
	(2, 2)	ADD×2	690	28	5	5,830
	(3, 2)	–	0	0	0	0
Ours w/o Step 2	**(1, 1)**	**MUL**	**342**	**6**	**3**	**6,427**
	(2, 1)	ADD	589	19	7	4,792
	(3, 1)	MUL	348	5	3	6,321
	(1, 2)	MUL	325	4	3	6,186
	(2, 2)	ADD×2	690	28	5	5,830
	(3, 2)	–	0	0	0	0
Ours	**(1, 1)**	**MUL**	**342**	**6**	**3**	**6,427**
	(2, 1)	ADD	589	19	7	4,792
	(3, 1)	MUL	348	5	3	6,321
	(1, 2)	MUL	325	4	3	6,186
	(2, 2)	ADD×2	690	28	5	5,830
	(3, 2)	–	0	0	0	0

Table 5 The configuration in each RDR island (PARKER).

Algorithm	Island	FUs	Controller Area [μm^2]	MUXs	Registers	area [μm^2]
MCAS	(1, 1)	SUB, COMP	168	12	5	3,355
	(2, 1)	**ADD×2**	**243**	**11**	**5**	**3,479**
	(1, 2)	SUB	170	2	2	1,286
	(2, 2)	–	0	0	0	0
Modified MCAS (1)	(1, 1)	SUB, COMP	168	12	5	3,355
	(2, 1)	**ADD×2**	**270**	**12**	**5**	**3,618**
	(1, 2)	SUB, ADD	172	2	3	1,576
	(2, 2)	–	0	0	0	0
Ours w/o Step 2	**(1, 1)**	**SUB, COMP**	**174**	**13**	**5**	**3,467**
	(2, 1)	ADD×2	228	7	5	3,016
	(1, 2)	SUB, ADD	234	5	4	2,544
	(2, 2)	–	0	0	0	0
Ours	**(1, 1)**	**SUB, COMP**	**174**	**13**	**5**	**3,467**
	(2, 1)	ADD×2	228	7	5	3,016
	(1, 2)	SUB, ADD	234	5	4	2,544
	(2, 2)	–	0	0	0	0

does not directly lead to increasing the RDR island size. In fact, we added a new FU (ADD) to the Island $I(1,2)$ in PARKER (Table 5) but it does not become a maximum-sized island.

5. Conclusion

In this paper, we proposed a delay-variation-aware high-level synthesis algorithm for RDR architectures. Our proposed algorithm reduced delayed scheduling/binding latency by up to 42.9% compared with the conventional algorithm. Our proposed algorithm gives one of the most effective and feasible solutions to the delay variation in high-level synthesis.

In the future, we will consider extensive delay variations and develop an improved version of our proposed algorithm.

Acknowledgments This research is partially supported by NEDO.

References

[1] Amin, C.S., Menezes, N., Killpack, K., Dartu, F., Choudhury, U., Hakim, N. and Ismail, Y.I.: Statistical static timing analysis: How simple can we get?, *Proc. 42nd Design Automation Conference*, pp.652–657 (2005).

[2] Cong, J., Fan, Y., Han, G., Yang, X. and Zhang, Z.: Architecture and synthesis for on-chip multicycle communication, *IEEE Trans. Computer-Aided Design*, Vol.23, No.4, pp.550–564 (2004).

[3] Ernst, D., Kim, N.S., Das, S., Pant, S., Rao, R., Pham, T., Ziesler, C., Blaauw, D., Austin, T., Flautner, K. and Mudge, T.: Razor: A low-power pipeline based on circuit-level timing speculation, *Proc. 36th IEEE/ACM International Symposium on Microarchitecture*, pp.7–18 (2003).

[4] Hagio, Y., Yanagisawa, M. and Togawa, N.: High-level synthesis with post-silicon delay tuning for RDR architectures, *Proc. International SoC Design Conference* (*ISOCC*), pp.194–197 (2013).

[5] Kawamura, K., Yanagisawa, M. and Togawa, N.: A thermal-aware high-level synthesis algorithm for RDR architectures through binding and allocation, *IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences*, Vol.E96-A, No.1, pp.312–321 (2013).

[6] Shi, Y., Igarashi, H., Togawa, N. and Yanagisawa, M.: Suspicious timing error prediction with in-cycle clock gating, *Proc. 14th International Symposium on Quality Electronic Design*, pp.335–340 (2013).

[7] Tanaka, S., Yanagisawa, M., Ohtsuki, T. and Togawa, N.: A fault-secure high-level synthesis algorithm for RDR architectures, *IPSJ Trans. System LSI Design Methodology*, Vol.4, pp.150–165 (2011).

[8] Yoshida, H. and Fujita, M.: An energy-efficient patchable accelerator for post-silicon engineering changes, *Proc. 9th International Conference on Hardware/Software Codesign and System Synthesis*, pp.13–20 (2011).

[9] Weste, N.H.E. and Harris, D.M.: *CMOS VLSI Design: A Circuits and Systems Perspective*, Addison-Wesley Publishing Company (2010).

Yuta Hagio received his B.E. degree from Waseda University in 2013 in computer science and engineering. Currently he is working toward M.E. degree there. His research interest includes variation-aware high-level synthesis of LSIs.

Masao Yanagisawa received his B.E., M.E., and Dr.E. degrees from Waseda University in 1981, 1983, and 1986, respectively, all in electrical engineering. He was with University of California, Berkeley from 1986 through 1987. In 1987, he joined Takushoku University. In 1991, he left Takushoku University and joined Waseda University, where he is presently a Professor in the Department of Electronic and Photonic Systems. His research interests are combinatorics and graph theory, computational geometry, LSI design and verification, and network analysis and design. He is a fellow of IEICE and a member of IEEE and ACM.

Nozomu Togawa received his B.E., M.E., and Dr.E. degrees from Waseda University in 1992, 1994, and 1997, respectively, all in electrical engineering. He is presently a Professor in the Department of Computer Science and Engineering, Waseda University. His research interests are LSI design, graph theory, and computational geometry. He is a member of IEEE and IEICE.

(Recommended by Associate Editor: *Atsushi Takahashi*)