IPSJ Transactions on System LSI Design Methodology Vol. 4 150–165 (Aug. 2011)

Regular Paper

A Fault-Secure High-Level Synthesis Algorithm for RDR Architectures

Sho Tanaka,[†1] Masao Yanagisawa,[†2]
Tatsuo Ohtsuki[†1] and Nozomu Togawa[†1]

As device feature size decreases, the reliability improvement against soft errors becomes quite necessary. A fault-secure system, in which concurrent error detection is realized, is one of the solutions to this problem. On the other hand, average interconnection delays exceed gate delays which leads to a serious timing closure problem. By using *regular-distributed-register architecture* (*RDR architecture*), we can estimate interconnection delays very accurately and their influence can be much reduced even in behavioral-level design. In this paper, we propose a fault-secure high-level synthesis algorithm for an RDR architecture. In fault-secure high-level synthesis, a recomputation CDFG as well as a normal-computation CDFG must be scheduled to control steps and bound to functional units. Firstly, our algorithm re-uses vacant areas on RDR islands to allocate new function units additionally for the recomputation CDFG. Secondly, we propose an efficient edge-break algorithm which considers comparison nodes' scheduling/binding. We can have small-latency scheduling/binding for both the normal CDFG and recomputation CDFG. Our algorithm reduces the required control steps by up to 53% compared with the conventional approach.

1. Introduction

While advanced process technologies contribute to improve the device integration, speed up the operation and reduce the power consumption [2], soft errors and system design reliability are becoming important problems [10]. Previous works mainly focused on soft errors on memory elements. This is because they are more susceptible to soft errors than logical circuits, since soft errors on logical circuits could be masked by logical, electrical, and latching-window masking [9]. However, the soft error susceptibility in logical circuits will be comparable to that of mem-

†1 Department of Computer Science and Engineering, Waseda University
†2 Department of Electronic and Photonic Systems, Waseda University

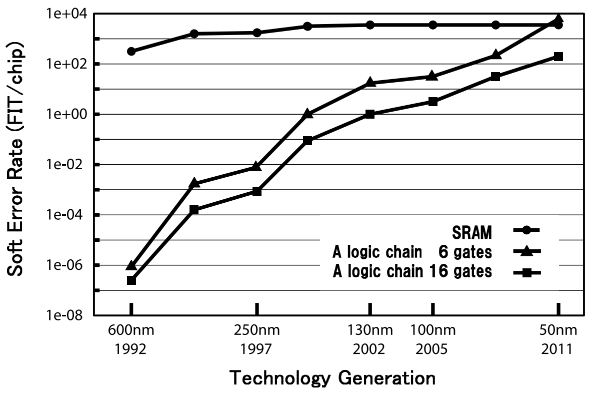


Fig. 1 Technology node and soft error rate [9].

ory circuits as the technology node advances, as we can see in **Fig. 1**. Soft errors on logical circuits that occur close to the clock edge can result in a *fault*. LSI manufacturing technology cannot completely prevent the faults caused by soft errors. That is why fault-secure design methodologies are strongly required in recent LSI design. High-level synthesis is also focused on to develop complicated LSIs in a short turn-around time. We have to focus on fault-secure high-level LSI synthesis.

There are several types of existing works on fault-secure high-level synthesis:
(1) Reliability-centric high-level synthesis
(2) Triple modular redundancy (TMR)
(3) Concurrent error detection (CED)

In (1), the main approach is to increase the reliability of the synthesized circuit as much as possible using more reliable functional units (FUs, in short). This approach can increase reliability but cannot detect or correct errors. In (2), a simple majority voting system is used where three modules are connected in parallel. TMR is a hardware-redundancy-based error correction technique and has at least 200% hardware overhead [3),4]. In (3), we duplicate a normal control/dataflow graph (CDFG) and generate a *recomputation CDFG*. We schedule and bind both the original CDFG and the recomputation CDFG by making use of the idle computation cycles and the idle data transfer cycles, and compare their re-

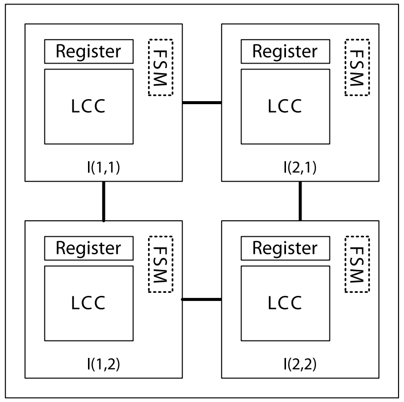


Fig. 2 RDR architecture.

sults[1),8)]. CED can reduce overheads significantly compared with TMR. Antola, et al.[1)] proposed an Force-Directed Scheduling (FDS) based fault-secure scheduling algorithm to minimize area overheads. Since this approach does not *break* a recomputation CDFG, the resultant latency or area overhead may increase. Wu, et al.[8)] proposed a CED technique that selectively breaks data dependencies in a recomputation CDFG to minimize the resultant latency. Edges in a recomputation CDFG are broken to improve the FU sharing between normal and recomputation CDFGs. However, since this technique does not consider new comparator insertion, it may increase the resultant latency if too many new comparison nodes are inserted. In this paper, we focus on (3) CED as fault-secure high-level synthesis.

Nanometer process technologies allow us to integrate billions of transistors on a single die, running at multiple-gigahertz frequencies. On the other hand, average interconnection delays exceed gate delays which leads to a timing closure problem. By using a regular-distributed-register architecture (RDR architecture), we can estimate interconnection delays very accurately[5)–7)]. The RDR architecture divides the entire chip into an array of islands. Each island has a local computational cluster (LCC), local registers, and a finite state machine (FSM). LCC includes several FUs such as multipliers and ALUs. **Figure 2** illustrates a 2×2 island-based RDR architecture. The registers are distributed to each island so that wire delay between FUs and registers can be reduced. Inter-island communication can use multicycle communication. Then clock cycle time can be occupied by almost the intra-island delay.

Since the RDR architecture divides the chip into an array of islands regularly, all the islands are not always *full*. Some of the islands have *vacant* spaces. By using these vacant islands for recomputation in fault-secure high-level synthesis, we expect that we can reduce latency with small area overheads. But there are no existing works in RDR architecture from the viewpoint of fault-secure high-level synthesis.

In this paper, we propose a fault-secure high-level synthesis algorithm for an RDR architecture. In fault-secure high-level synthesis, a recomputation CDFG as well as a normal CDFG must be scheduled to control steps and bound to FUs. Firstly, our algorithm re-uses vacant spaces on RDR islands to allocate new FUs additionally for the recomputation CDFG. Secondly, we propose an edge-break algorithm which considers comparison nodes' scheduling/binding. We can have small-latency scheduling/binding for both the normal CDFG and recomputation CDFG. Our algorithm reduces the required control steps by up to 53% compared with the conventional approach.

This paper is organized as follows: Section 2 defines our fault-secure high-level synthesis problem for RDR architecture; Section 3 proposes a fault-secure high-level synthesis algorithm; Section 4 demonstrates experimental results; and Section 5 gives several concluding remarks.

2. Problem Formulation

A *normal CDFG* $G = (V, E)$ is represented by a directed graph, where V is a set of operation nodes, constant nodes and branch nodes and E is a set of edges which denote the data/control dependencies. A *recomputation CDFG* $G_R = (V_R, E_R)$ is generated by just duplicating the normal CDFG $G = (V, E)$.

Given scheduling and binding of a normal CDFG $G = (V, E)$, we schedule and bind its recomputation CDFG $G_R = (V_R, E_R)$ so that it satisfies the *fault-secure conditions* (1) to (6) below[1),8)]:

(1) Scheduling and binding of $G = (V, E)$ is not changed.

(2) Let $n' \in V_R$ be one of the operation nodes in the recomputation CDFG,

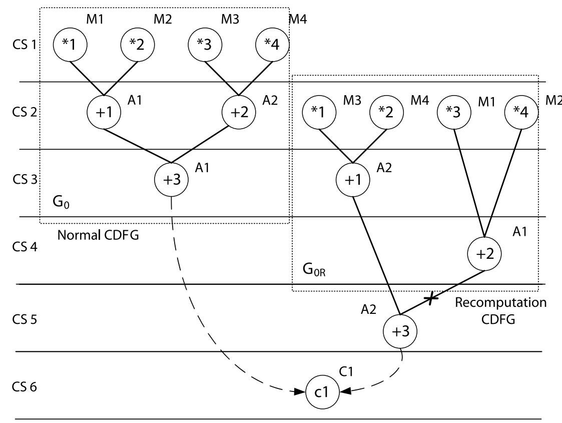


Fig. 3 Fault-secure scheduling and binding.

and let $n \in V$ be the corresponding normal CDFG's node. If there are two or more FUs which can execute n and n', they do not share the same FU.

(3) The outputs of normal and recomputation CDFG must be compared.

(4) A data-flow edge $e' \in E_R$ in the recomputation CDFG can be broken.

(5) Let $n' \in V_R$ and $e' = (m', n') \in E_R$ be the one of the operation nodes and its input data-flow edge in the recomputation CDFG, where m' is one of the parent nodes of n'. Let m be the normal CDFG's node corresponding to m'. If e' is broken, n' uses the output of the normal CDFG's node m instead of m'.

(6) In case of (5), the outputs of m and m' must be compared.

Example 1. **Figure 3** shows an example of fault-secure scheduling and binding. This example uses four multipliers “M1”,“M2”,“M3”, and “M4”, two adders “A1” and “A2”, and two comparators “C1” and “C2”. According to the fault-secure condition (1), scheduling/binding of the normal CDFG G_0 is not changed. According to the fault-secure condition (2), the normal CDFG's operation “*1” uses the multiplier “M1”, but the recomputation CDFG's operation “*1” uses the multiplier “M3”. According to the fault-secure condition (3), the outputs of the normal CDFG and the recomputation CDFG are compared by using the comparator “C1”.

Figure 4 shows the scheduling/binding of the recomputation CDFG G_{0R} when breaking the edge marked as × in Fig. 3. By breaking this edge, the recomputation CDFG's operation “+3” can be scheduled to the control step “CS4”. According to the fault-secure conditions (4) and (5), the input of recomputation CDFG's operation “+3” uses the output of the normal computation CDFG's operation “+2”. According to the fault-secure condition (6), the output of the normal CDFG's operation “+2” and recomputation CDFG's operation “+2” are compared by using the comparator “C2”. □

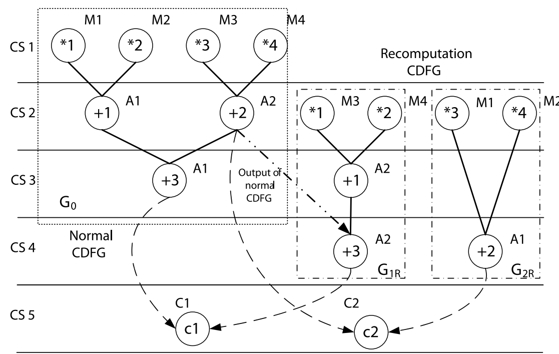


Fig. 4 Fault-secure scheduling and binding after the breaking edge.

The RDR architecture divides the entire chip into $N \times M$ array of islands. Let $I(x, y)$ be the island on the position (x, y) of the RDR array, where $1 \leq x \leq N$ and $1 \leq y \leq M$. Every island assumes to be square. An 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 file $R(I(x, y))$. Let i_1 and i_2 be two islands $I(x_1, y_1)$ and $I(x_2, y_2)$ in the RDR architecture. Interconnection delay $D_c(i_i, i_2)$ between the two islands i_1 and i_2 is proportional to the square of their distance and it is given by

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

where C_d shows the constant interconnection delay coefficient. 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 that $d(fu_1) < T_{clk}$. When

$$T_{clk} \geq D_c(i_1, i_2) + d(fu_1), \quad (2)$$

executing fu_1 and storing its output into the register file $R(i_2)$ are done in a single control step. On the other hand, when

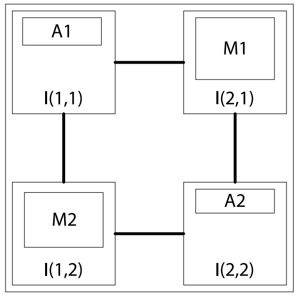


Fig. 5 Allocation of FUs on RDR architecture.

$$T_{clk} < D_c(i_1, i_2) + d(fu_1), \tag{3}$$

we only execute fu_1 and store its output into the register file $R(i_1)$ at the first control step. After that, we transfer the fu_1's output from $R(i_1)$ to $R(i_2)$ using $\lceil \frac{D_c(i_i,i_2)}{T_{clk}} \rceil$ control steps. If $d(fu_1) \geq T_{clk}$, we also use control steps only for data transfer.

Let $c(fu)$ be the *cost* of FU fu. Every island has the capacity C. Let $F(i)$ be a set of FUs allocated to the island $i = I(x, y)$. Any island i satisfies

$$C \geq \sum_{fu \in F(i)} c(fu). \tag{4}$$

This is called a *capacity constraint*. In other words, an additional FU fu_j can be newly allocated to the island i when

$$c(fu_j) \leq C - \sum_{fu \in F(i)} c(fu) \tag{5}$$

holds.

Example 2. **Figure 5** shows an example of RDR architecture. Let

$$d(\text{A1}) = d(\text{A2}) = 1\text{ns} \tag{6}$$

and

$$d(\text{M1}) = d(\text{M2}) = 2\text{ns} \tag{7}$$

be the delay of FUs "A1", "A2", "M1", and "M2". Let $T_{clk} = 2$ns be the clock period, and let $C_d = 1$ns be the interconnection delay coefficient. Let $i_1 = I(1, 1), i_2 = I(2, 1), i_3 = I(1, 2), i_4 = I(2, 2)$ be the four islands on the RDR architecture. Consider the DFG as shown in **Fig. 6** (a) and its data flow "D1". The operation node "+1" is bound to the adder "A1" and "*1" is bound to the multiplier "M1". "A1" is allocated to the island i_1 and "M1" is allocated to the island i_2. Then we have

$$\begin{aligned} D_c(i_1, i_2) + d(\text{A1}) &= D_c(I(1, 1), I(2, 1)) + d(\text{A1}) \\ &= 1 \times (|2-1| + |1-1|)^2 + 1 \\ &= 2 \leq T_{clk}. \end{aligned} \tag{8}$$

This means that executing the operation "+1" and storing its output into the island i_2 can be done in a single clock cycle "CS1" and the scheduling "D1" as shown in Fig. 6 (a) is feasible. However, consider the data flow "D2" in Fig. 6 (a). In the same way, we have

$$\begin{aligned} D_c(i_3, i_4) + d(\text{M2}) &= D_c(I(1, 2), I(2, 2)) + d(\text{M2}) \\ &= 1 \times (|1-2| + |2-2|)^2 + 2 \\ &= 3 > T_{clk}. \end{aligned} \tag{9}$$

This means that executing the operation "*2" and storing its output into the island i_4 cannot be done in a single clock cycle "CS1". We cannot start the operation "+2" at "CS2" in the data flow "D2".

The modified scheduling is shown in Fig. 6 (b), where the data flow "D2" uses the control step "CS2" only for data transfer. □

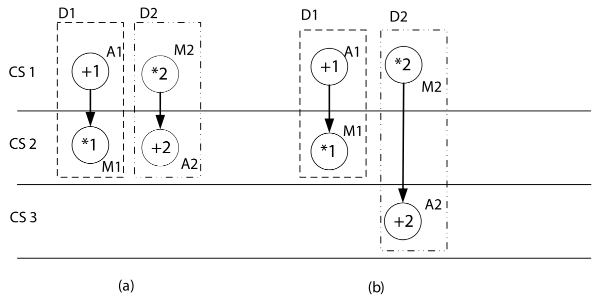


Fig. 6 (a) Scheduling/binding which ignores interconnection delays between islands. (b) Scheduling/binding which considers interconnect delays between islands.

Now, a fault-secure high-level synthesis problem for an RDR architecture is defined as follows:

Definition 1. For given the number of RDR architecture islands $N \times M$, their allocation of FUs, island capacity C, clock period T_{clk}, and normal CDFG's scheduling/binding, the fault-secure high-level synthesis problem for this RDR architecture is, to schedule and bind its recomputation CDFG satisfying the fault-secure conditions (1)–(6) and the capacity constraint. The objective is to minimize the total control steps. The FUs already allocated to the islands cannot be changed, but new FUs can be allocated when they satisfy the capacity constraint. □

3. Fault-secure High-level Synthesis Algorithm for RDR Architecture

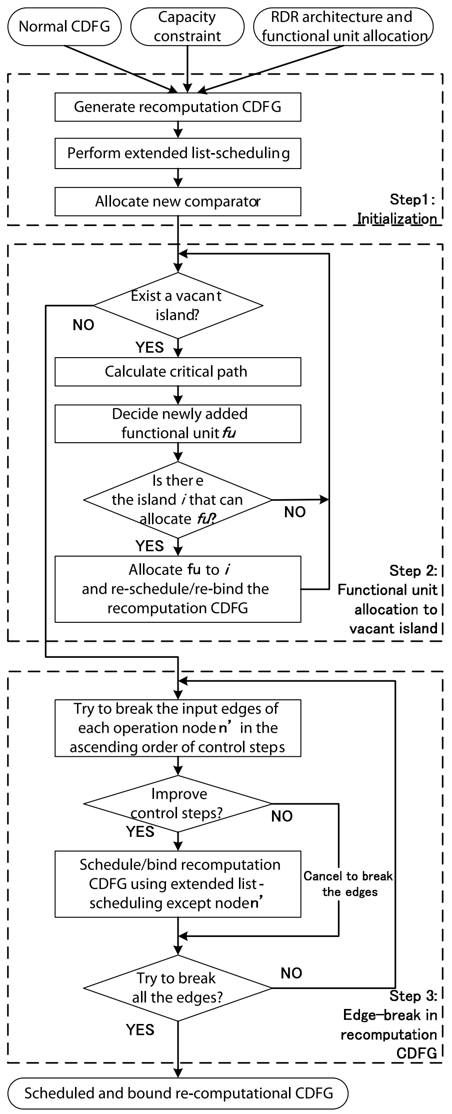
To solve the fault-secure high-level synthesis problem defined in Section 2, we need to consider (a) edge-break in the recomputation CDFG and (b) new FU allocation to vacant islands. We can have the following three options to tackle this problem:

(1) (a) is processed first and then (b) is processed secondly.
(2) (a) and (b) are processed in a step-by-step manner.
(3) (b) is processed first and then (a) is processed secondly.

(1) is hard to optimize fault-secure high-level synthesis since we need to *forecast* new FU allocation. (2) must improve fault-secure high-level synthesis locally in a step-by-step manner, but it must be very hard to have a globally optimal solution. In (3), we can completely see the FU arrangement including new FUs since (b) is processed first. Based on it, we can break recomputation CDFG's edges and schedule and bind the recomputation CDFG. By performing the edge-break step based on the complete FU arrangement, we can insert as small number of comparison nodes as possible and then we will obtain a recomputation CDFG's scheduling/binding with minimized control steps.

Based on the above discussion, we employ the strategy (3) as our algorithm. Our proposed fault-secure high-level synthesis algorithm for an RDR architecture is shown in **Fig. 7**. Our algorithm has the three steps: initialization (Step 1), FU allocation to vacant islands (Step 2), and edge-break in the recomputation CDFG (Step 3). For simplicity, we assume DFG as an input CDFG in this section but we can apply the proposed algorithm here to CDFG as shown in "PARKER" in the experimental results.

Note that, Step 1 and Step 2 in our algorithm are completely new ones, in which we consider recomputation CDFG scheduliIng/binding taking into account



Normal CDFG
Capacity constraint
RDR architecture and functional unit allocation
Generate recomputation CDFG
Perform extended list-scheduling
Allocate new comparator
Step1 : Initialization
Exist a vacant island?
NO
YES
Calculate critical path
Decide newly added functional unit fu
Is there the island i that can allocate fu?
NO
YES
Allocate fu to i and re-schedule/re-bind the recomputation CDFG
Step 2: Functional unit allocation to vacant island
Try to break the input edges of each operation node n' in the ascending order of control steps
Improve control steps?
NO
YES
Schedule/bind recomputation CDFG using extended list-scheduling except node n'
Cancel to break the edges
Try to break all the edges?
NO
YES
Step 3: Edge-break in recomputation CDFG
Scheduled and bound re-computational CDFG

Fig. 7 Synthesis flow.

RDR architecture. Step 3 introduces an idea of scheduling/binding a node in a recomputation CDFG to an earlier control step by "edge-break," which is similar to that of Wu's approach[8]. However, how to realize this edge-break is quite different. Wu's approach is just based on breaking as many edges as possible and assigning recomputation CDFG nodes to as early control steps as possible. It can introduce too many comparison nodes and thus the latency of recomputation CDFG may be increased. On the other hand, our algorithm is based on breaking edges only when a recomputation CDFG node n' and its associated comparison nodes can be assigned to the control steps earlier than or equal to the control step which n' is originally assigned to. We do not perform unnecessary edge-breaks here and then we can reduce the overall latency.

3.1 Initialization (Step 1)

This step solves our fault-secure problem temporarily to handle Step 2 and Step 3 easily. In Step 1, we employ the following approach:

(1.1) Generate a recomputation CDFG by duplicating the input normal CDFG.
(1.2) Schedule/bind the recomputation CDFG temporarily.
(1.3) Allocate a new comparator if none of the islands has one.

Step (1.1) just generates a recomputation CDFG $G_R = (V_R, E_R)$ by duplicating the normal CDFG $G = (V, E)$.

Step (1.2) schedules/binds the recomputation CDFG so that it meets the fault-secure conditions. Here, we propose an extended list-scheduling algorithm for scheduling/binding. This algorithm performs list-scheduling-based scheduling and binding as well as FU allocation to an RDR architecture. **Figure 8** shows our extended list-scheduling algorithm.

In our extended list-scheduling algorithm, the priority of operation nodes uses a difference between their ALAP control step and ASAP control step as in a normal list-scheduling algorithm. The priority is higher if the difference is smaller. The important key of this algorithm is Step (1.2)-**3.2.1** and we employ the following approach:

Now we focus on scheduling/binding at a control step cs and an operation type t such as "+" and "*" in Step (1.2)-**3.2.1**. Let $PQ(t)$ be a priority queue which includes operation nodes with operation type t in the recomputation CDFG and can be assigned to the control step cs. For each FU fu in the RDR architecture,

```
Step (1.2) Extended list-scheduling
1. Set a priority to each operation node $n' \in V_R$ in the recomputation CDFG.
2. For each operation type $t$, insert an operation node which has no preceding nodes
   into the priority queue $PQ(t)$. Let $cs \leftarrow 0$.
3. Repeat the following loop until $PQ(t) = \emptyset$ for all operation types:
   3.1 $cs \leftarrow cs + 1$.
   3.2 For each operation type $t$:
       3.2.1 Based on the FU-binding in the normal CDFG, assign the operation
             nodes in $PQ(t)$ to $cs$ as much as possible as described in Section 3.1.
       3.2.2 Delete the scheduled and bound operation nodes in 3.2.1 from $PQ(t)$.
   3.3 Insert the ready operation nodes into $PQ(t)$ and go to 3.
```

Fig. 8 Extended list-scheduling algorithm.

let $N(t, cs, fu) \subseteq PQ(t)$ be a set of operation nodes with operation type t which can use fu at the control step cs considering interconnection delay from its preceding nodes and the fault-secure condition (2). In Step (1.2)-**3.2.1**, we delete the highest-priority operation node n' from $PQ(t)$ and bind it to fu which n' can use at the control step cs. If there are more than one such FUs, we select the one which gives the minimum $|N(t, cs, fu)|$[⋆1]. We also schedule n' to cs. If there are no available FUs for n', we insert n' again into $PQ(t)$ and we try the next highest-priority operation node. By repeating this process, we can expect that it binds as many operation nodes as possible at each control step cs and thus reduces the overall latency.

Example 3. **Figure 9** illustrates how Step (1.2)-**3.2.1** works. For simplicity, we demonstrate here how our extended list scheduling algorithm schedules/binds CDFG nodes to control steps and FUs, just ignoring the fault-secure condition (2), and show the effectiveness of using the minimum $|N(t, cs, fu)|$ as a criterion. In fault-secure high-level synthesis, however, we have to consider the fault-secure condition (2) as well when we assign recomputation CDFG nodes to FUs.

Now we assume that FU allocation as shown in Fig. 5 is given and scheduling/binding CDFG nodes to CS1, CS2, and CS3 is done as in Fig. 9 (a). In this example, we assume

⋆1 As the example just below shows, using the minimum $|N(t, cs, fu)|$ can lead to improve FU usage in each control step. However, there may be better algorithms or optimal algorithms for initial recomputation CDFG scheduling/binding. This is one of our future works.

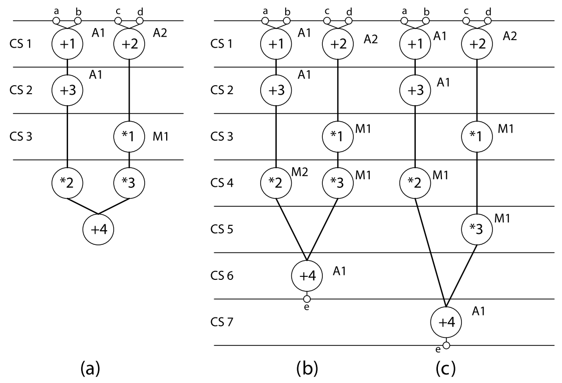


Fig. 9 Example of extended list-scheduling.

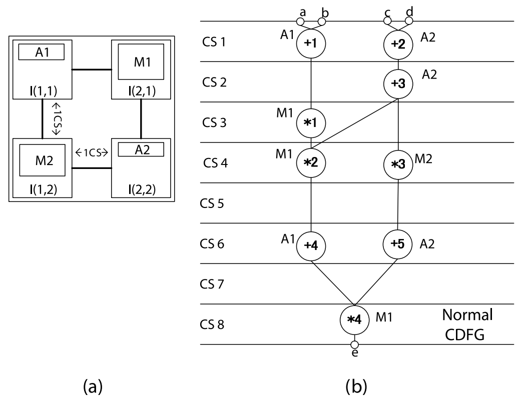


Fig. 10 (a) RDR architecture and (b) scheduling and binding for normal CDFG.

that the interconnection delay between adjacent islands requires one control step (1CS). We consider the next control step CS4. In this case, the nodes "*2" and "*3" can be scheduled to CS4 and we have $PQ(*) = \{*2, *3\}$. Since "+3" is executed by the adder "A1" at CS2 and the control step CS3 is used only for data transfer between "+3" and "*2", then "*2" can be executed at CS4 by either the multiplier "M1" or the multiplier "M2", which is one-island away from "A1". On the other hand, since "*1" is executed by "M1" at CS3, "*3" can be executed at CS4 only by "M1", not by "M2". Then we have

$$N(*, 4, M1) = \{*2, *3\} \quad (10)$$

$$N(*, 4, M2) = \{*2\} \quad (11)$$

and

$$|N(*, 4, M1)| = 2 \quad (12)$$

$$|N(*, 4, M2)| = 1. \quad (13)$$

In this case, the minimum $|N(t, cs, fu)|$ is given by $|N(*, 4, \mathrm{M2})|$, and then the operation node "*2" is scheduled to CS4 and bound to "M2" first. After that, the operation node "*3" is scheduled to CS4 and bound to "M1" as shown in Fig. 9 (b). Note that, if the operation node "*2" is bound to "M1" first, then we will have a scheduling/binding result as shown in Fig. 9 (c), which has a longer latency than Fig. 9 (b). By using the minimum $|N(t, cs, fu)|$ as a criterion, we can improve FU usage as illustrated in this example. □

In Step (1.2), we schedule/bind the recomputation CDFG by using this extended list-scheduling algorithm. Note that, in Step (1.2), we do not schedule/bind a new comparator to compare the results between normal CDFG and recomputation CDFG.

Then **Step (1.3)** allocates a new comparator if there are no comparators in the RDR architecture. Let $n' \in V_R$ be an operation node in the recomputation CDFG. Let $cs(n')$ be the control step to which n' is scheduled. Let $CS'_{\max}$ be the maximum control step in the recomputation CDFG. Let $n'_{\max}$ be one of the operation nodes scheduled to $CS'_{\max}$. Let fu' be the FU that is bound to $n'_{\max}$ and $i_r = I(x_r, y_r)$ be the island to which fu' is allocated. A new comparator is allocated to the island which is the nearest to the island i_r and has an enough capacity to accommodate a new comparator. In case the RDR architecture does not have such an island, a new comparator is allocated to the island i_r ignoring the capacity constraint. This case violates the capacity constraint and may occur significant area overhead.

Example 4. **Figure 10** (b) shows a scheduled and bound normal CDFG when RDR architecture in Fig. 10 (a) is given. In this example, we assume that the interconnection delay between adjacent islands requires one control step (1CS). **Table 1** shows the cost of each FU. The capacity constraint C for each island is set to be two. Step (1.1) duplicates the normal CDFG and generates a recomputation CDFG. According to the

Table 1 FU costs.

	Cost
Adder (A1, A2, A3)	1
Multiplier (M1, M2)	2
Comparator (C1)	1

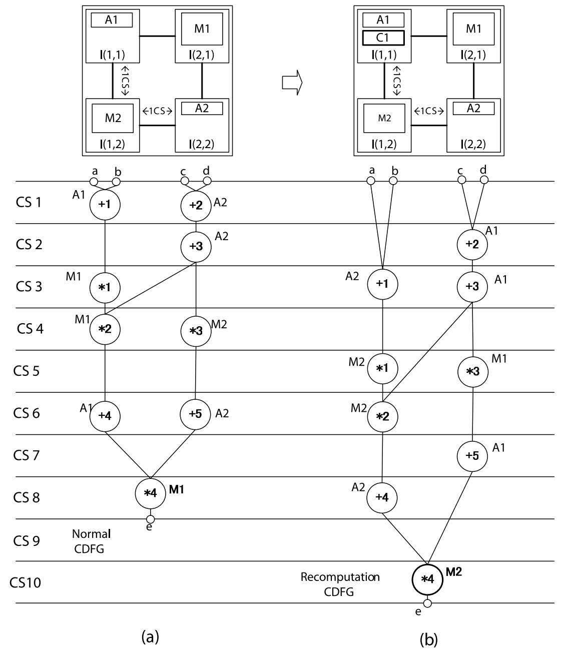


Fig. 11 Initialization.

Step (1.2), we have the scheduling/binding result of recomputation CDFG as shown in **Fig. 11** (b). In this scheduling/binding, the recomputation CDFG's operation node "*4" gives the maximum control step $\mathrm{CS}'_{\max} = \mathrm{CS10}$. This operation node "*4" is bound to the multiplier "M2" and the island which accommodates the multiplier "M2" is $i_r = I(1,2)$. According to Step (1.3), a new comparator is allocated to the island $I(1,1)$ which is the nearest to i_r and has a enough capacity to accommodate a new comparator. □

3.2 FU Allocation to Vacant Islands (Step 2)

This step allocates new FUs if the RDR architecture has vacant islands. We employ the following approach:

(2.1) Determine critical paths in the recomputation CDFG.
(2.2) Determine new FUs to be allocated to vacant islands.
(2.3) Allocate the new FUs to vacant islands.

In this approach, we first find an operation node which is scheduled to a "late" control step due to FU shortage. Then we allocate a new FU so that the operation node can be re-scheduled to an "earlier" control step. We expect that this approach also reduces the overall latency.

Step (2.1) calculates a critical path based on the scheduling/binding of the recomputation CDFG. We focus on the recomputation CDFG's operation node n' which is scheduled to $\mathrm{CS}'_{\max}$. Let $P(n')$ be a set of parent nodes of n' and $m' \in P(n')$ be one of them. Let $s(n', m')$ be the control step of operation node n' in case we assume that n' has only m' as its parent node, i.e, we assume that all the parents nodes other than m' are deleted. Let us consider

$$S_{\max}(n') = \max_{m' \in P(n')} s(n', m'). \tag{14}$$

and let $m'_{\max}(n')$ be one of the parent node of n' which gives $S_{\max}(n')$. We focus on $m'_{\max}(n')$ instead of n' and repeat this process until we finally reach the primary input node. We define a critical path by a sequence of operation nodes focused on at the above process and denoted as $CP_R = \{n'_1, n'_2, \ldots, n'_p\}$ from the beginning node to the ending node.

Based on this critical path CP_R, we determine new FUs which will be allocated to vacant islands. Let $CP = \{n_1, n_2, \ldots, n_p\}$ be the normal CDFG's path corresponding to the recomputation CDFG's critical path CP_R. We compute for CP

$$dist_i = \begin{cases} cs(n_i) - cs(n_{i-1}) & (2 \le i \le p) \\ cs(n_1) & (i = 1) \end{cases}. \tag{15}$$

Also we compute for CP_R

$$dist'_i = \begin{cases} cs(n'_i) - cs(n'_{i-1}) & (2 \leq i \leq p) \\ cs(n'_1) & (i = 1) \end{cases} . \qquad (16)$$

$dist_i$ shows the difference between the control step of n_i and the control step of its parent node n_{i-1}. Similarly, $dist'_i$ shows the difference between the control step of n'_i and the control step of its parent node n'_{i-1}. Then we can compute the difference between $dist_i$ and $dist'_i$:

$$dist(n_i) = dist'_i - dist_i. \qquad (17)$$

$dist(n_i)$ shows how many control steps the recomputation CDFG's operation node n'_i is delayed, compared with the corresponding normal CDFG's operation node, due to FU shortage. In other words, this delay directly increases the overall latency.

In **Step (2.2)**, we determine new FU candidates which will be added to vacant islands. We first sort the $dist(n_i)$ values in the descending order and try to allocate a new FU fu_i which can execute the operation node n_i in this order.

In **Step (2.3)**, we find out vacant islands and allocate new FUs to them. Let $I(fu_i)$ be a set of islands that has an enough capacity to accommodate a new FU fu_i. When we try to allocate each new FU candidate fu_i, we will have the following three cases:

(i) There exists only one island i_v in $I(fu_i)$.
(ii) There exist two or more islands in $I(fu_i)$.
(iii) No islands exists in $I(fu_i)$.

In case of (i), the new FU fu_i is allocated to the island i_v. In case of (ii), we determine the island i_v to which the new FU fu_i is allocated as follows: Firstly we try to allocate fu_i to each island in $I(fu_i)$. Secondly we re-schedule and re-bind the recomputation CDFG and calculate the required control steps. Lastly, we actually allocate fu_i to the island i_v which gives the minimum required control steps. In case of (iii), we just skip new FU allocation for fu_i and try to allocate the next FU candidate. Note that, every time the new FU is allocated, we re-schedule and re-bind the recomputation CDFG.

After trying to allocate new FU candidates for all the operation nodes in the critical path CP_R, we determine the next critical path if we further have vacant islands. The next critical path is determined in the same way as in Step (2.1) by just removing the first critical path[⋆1].

```
Step 2 FU allocation to vacant islands
  (2.1) Find the critical path $CP_R$ and $CP$ and calculate $dist(n_i)$.
  (2.2) Repeat Steps (2.2.1)–(2.2.2) in the descending order of $dist(n_i)$ until there are
     no vacant islands:
      (2.2.1) In case there exists only one vacant island $i_v$ for a new FU $fu_i$, allocate
          $fu_i$ to $i_v$.
      (2.2.2) In case there exist two or more vacant islands for a new FU $fu_i$:
        a:  Try to allocate $fu_i$ to every vacant island and re-schedule/re-bind the
            recomputation CDFG using the extended list-scheduling algorithm.
        b:  Allocates $fu_i$ to $i_v$ which gives minimum required control steps and
            re-schedule/re-bind the recomputation CDFG using the extended list-
            scheduling algorithm.
  (2.3) If there still exist vacant islands, remove all the nodes in $CP_R$ and go to
     Step (2.1). Otherwise, finish.
```

Fig. 12 New FU allocate algorithm.

After that, we repeat the same process as in Step (2.2) and Step (2.3).

Figure 12 summarizes the algorithm of Step 2.

Example 5. Now we will find a critical path in the recomputation CDFG as shown in **Fig. 13** (b). The operation node "*4" gives the maximum control step CS10. A set of parent nodes of "*4" is defined by

$$P(*4) = \{+4, +5\}. \qquad (18)$$

If "+4" is the only parent node of "*4", then "*4" can be scheduled to CS10. Then $s(*4, +4)$ becomes 10. Similarly, we have $s(*4, +5) = 9$. Thus the parent node "+4" gives $S_{\max}(*4)$. This means that the parent node "+4" determines the scheduling of "*4". Then we focus on the operation node "+4" and repeat the same process. We can finally have the critical path CP_R in the recomputation CDFG as follows:

$$CP_R = \{+1, *1, *2, +4, *4\}. \qquad (19)$$

Corresponding to each node in CP_R, we also have the critical path in the normal CDFG

[⋆1] If two or more critical paths in a recomputation CDFG share several operation nodes, "just removing the first critical path" may be insufficient for determining a second or third critical path. However, RDR architecture does not have too much vacant islands and then we do not always focus on too many critical path nodes. This means that focusing on operation nodes on a first critical path will be enough for new FU allocation in most cases. In fact, we just focused on the first critical path in all the cases in our experiments in Section 4.

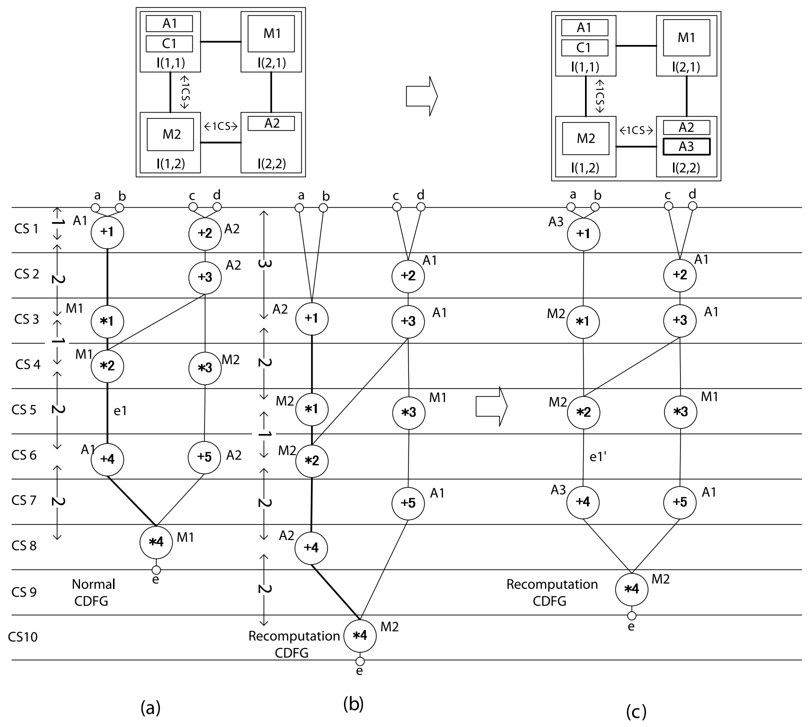


Fig. 13 FU allocation to vacant islands.

as follows:

$$CP = \{+1, {}^*1, {}^*2, +4, {}^*4\}. \quad (20)$$

Secondly, we determine new FU candidates to allocate. We calculate $dist_i$ and $dist'_i$ based on Eqs. (19) and (20):

$$\{dist_1, dist_2, \ldots, dist_5\} = \{1, 2, 1, 2, 2\} \quad (21)$$

$$\{dist'_1, dist'_2, \ldots, dist'_5\} = \{3, 2, 1, 2, 2\}. \quad (22)$$

We calculate the difference between $dist_i$ and $dist'_i$ and obtain $dist(n_i)$

$$\{dist(n_1), dist(n_2), \ldots, dist(n_5)\} = \{2, 0, 0, 0, 0\}. \quad (23)$$

Since the maximum value of $dist(n_i)$ is given by "+1", we try to allocate a new adder to a vacant island. As in Fig. 13, the island $I(2,2)$ has an enough capacity to accommodate a new adder, and then we actually allocate a new adder to the island $I(2,2)$. Since there are no vacant islands, we finish Step 2. □

3.3 Edge-Break in the Recomputation CDFG (Step 3)

This step breaks several edges in the recomputation CDFG so that some operation nodes can be scheduled/bound to earlier control steps. Breaking edges does not always lead to decreasing latency but it may increase the resultant latency when too many new comparison nodes are required. We propose here an efficient edge-break algorithm which takes into account comparison nodes' scheduling/binding. We employ the following two-step approach:

(3.1) Find an operation node n' in the recomputation CDFG which can be scheduled to an earlier control step if its input edges are broken.

(3.2) Insert a comparison node for each input edge of n' and re-schedule/re-bind the recomputation CDFG. Break the edges actually if the operation node n' as well as all its associated comparison nodes are scheduled to an earlier control steps and/or to the same control step.

In this two-step approach, we do not just break all possible edges but we first find out an operation node which can be scheduled to an earlier control step by breaking its input edges. After that, we try to insert comparison nodes and check whether they can be scheduled to an earlier control steps and/or to the same control step. We actually perform edge-break only if all the associated comparison nodes can be scheduled to an earlier control steps and/or to the same control step. This means that we actually perform edge-break only when we have small or equal latency for the targeted operation node. Since edge-break may increase the operation node's "mobility", we can expect that we will reduce the overall latency. In Steps (3.1) and (3.2), we focus on each recomputation CDFG's operation node n' in the ascending order of their scheduled control step $cs(n')$.

Step (3.1) is performed as follows: Let n' be an operation node in the recomputation CDFG and n be its corresponding operation node in the normal CDFG. Now the operation nodes n and n' are scheduled to the control steps $cs(n)$ and $cs(n')$, respectively. Let $P(n)$ be a set of parents nodes of n. If the operation node n' uses the outputs of $P(n)$ instead of the outputs of $P(n')$, it may be scheduled to the control step earlier than $cs(n')$. Let $s(n')$ be the earliest control step to which n' can be scheduled using the outputs of $P(n)$ instead of

```
Step 3 Edge-break in the recomputation CDFG
  (3.0) Visit each recomputation CDFG's operation nodes $n'$ in the ascending order of
    its control step $cs(n')$.
  (3.1) Calculate the earliest control step $s(n')$ to which $n'$ can be scheduled if the input
    edges in $E_b(n') \subseteq E_R$ of $n'$ are broken.
  (3.2) While $s(n') < cs(n')$, perform the Steps (3.2.1)–(3.2.4):
      (3.2.1) For each recomputation CDFG's edge $e' \in E_b(n')$, insert its comparison
        node $c_i$ and calculate its scheduled control step $cs(c_i)$.
      (3.2.2) Calculate the maximum scheduled control step
            $C_{\max} = \max_{c_i \in C_n} cs(c_i)$,
        where $C_n$ shows all the inserted comparison nodes.
      (3.2.3) If $C_{\max} \leq cs(n')$, break the edges in $E_b(n')$ and reschedule/rebind
        $n'$ to $s(n')$. Reschedule and rebind the recomputation CDFG's nodes
        except for $n'$ using the extended list-scheduling algorithm. Go to (3.3).
      (3.2.4) If $C_{\max} > cs(n')$, cancel the edge-break and insertion of the com-
        parison nodes. $s(n') \leftarrow s(n') + 1$ and go to (3.2).
  (3.3) Repeat Steps (3.1) and (3.2) until all the operation nodes in the recomputation
    CDFG are visited.
```

Fig. 14 The edge-break algorithm.

the outputs of $P(n')$. If $s(n') < cs(n')$, we determine a minimum set $E_b(n')$ of input edges to be broken and go to the next step (3.2). Otherwise, we try the next operation node in the recomputation CDFG.

We determine a minimum set $E_b(n')$ of input edges to be broken as follows: We first schedule the operation node n' to $s(n')$. Let e' be one of the input edges of n' and e be the corresponding edge in the normal CDFG. In case the operation node n' uses e' as its input, it violates the data transfer and n' cannot be scheduled to $s(n')$. In this case, $E_b(n')$ includes e'. Consider the case where, if the operation node n' uses e' as its input, n' can be still scheduled to $s(n')$. In this case, $E_b(n')$ does not include e'. Note that all the edges in $E_b(n')$ must be broken.

Step (3.2) is performed as follows: Let $e' = (m', n')$ be one of the edges in

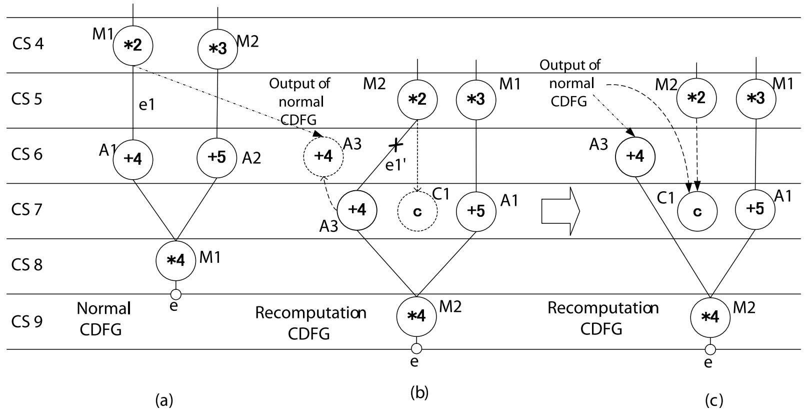


Fig. 15 Edge-break in the recomputation CDFG.

the minimum set $E_b(n')$ of edges to be broken. The operation node $m' \in P(n')$ is one of the parents node of n'. Let m be the normal CDFG's operation node corresponding to m'. We now insert a comparison node c_i to compare the outputs of m' and m and schedule/bind it. Let $cs(c_i)$ be the control step to which c_i is scheduled. In the similar way, we require several comparison nodes when all the edges in $E_b(n')$ are broken. Let C_n be the set of these comparison nodes. Then we calculate $C_{\max}$ as follows:

$$C_{\max} = \max_{c_i \in C_n} cs(c_i) \tag{24}$$

In case $C_{\max} \leq cs(n')$, we actually break each edge $e' \in E_b(n')$ and schedule/bind n' to the control step $s(n')^{\star 1}$. In this case, n' uses the outputs of the normal computation CDFG's nodes corresponding to the edge-breaks in $E_b(n')$. We also insert the comparison nodes for them. We re-schedule and re-bind the recomputation CDFG after breaking these edges.

⋆1 In case $C_{\max} < cs(n')$, the operation node n' as well as its associated comparison nodes can be scheduled to control steps earlier than $cs(n')$. We expect that we can finally reduce overall latency of our recomputation CDFG. Even when $C_{\max} = cs(n')$, the operation node n' can be scheduled to a control step earlier than $cs(n')$, although one of its associated comparison nodes is scheduled to $cs(n')$. We also expect that we can finally reduce overall latency of our recomputation CDFG in this case.

If $C_{\max} > cs(n')$, we cancel the edge-break for n' as well as the insertion of comparison nodes. If $(s(n') + 1) < cs(n')$ still holds, we increase the value of $s(n')$ by one, update a minimum set $E_b(n')$ of input edges to be broken, and repeat Step (3.2).

Figure 14 shows the edge-break algorithm.

Example 6. We explain our edge-break algorithm using the example as shown in Fig. 13. We visit each operation node in the recomputation CDFG in the ascending order of its control step, i.e., we visit "+1","+2","*1","+3", "*2", "*3", "+4", "+5", and "*4" in this order. From "+1" to "*3", the operation nodes cannot be scheduled to an earlier control step even if their input edges are broken. Then we focus on the operation node "+4" in the recomputation CDFG (Step (3.1)). "+4" is now scheduled to CS7 (**Fig. 15** (b)). Now we consider that we break the edge e'_1 and "+4" uses the output of "*2" in the normal computation CDFG. In this case, "+4" in the recomputation CDFG can be scheduled to CS6 (Fig. 15 (c)). Since CS6 is earlier than CS7, we go to the next step.

Now we insert the comparison node c to compare the output of "*2" in the normal CDFG with the output of "*2" in the recomputation CDFG (Step (3.2)). c can be scheduled to CS7. Since CS7 is equal to the original control step to which "+4" is scheduled, then we actually break the edge e'_1 and insert the comparison node c.

In the similar way, we visit the rest of the operation nodes in recomputation CDFG and break its input edges. Since the operation node "+5" does not satisfy $s(n') < cs(n')$, we do not break its input edges. Since the operation node "*4" satisfies $s(n') < cs(n')$ and $cs(c) \leq cs(n')$, we break its input edge. **Figure 16** shows the output of our edge-breaking algorithm applied to the recomputation CDFG shown in Fig. 13. As we can see in Fig. 16, the overhead to schedule/bind the recomputation CDFG is just two control steps, compared with scheduling/binding the normal CDFG only. □

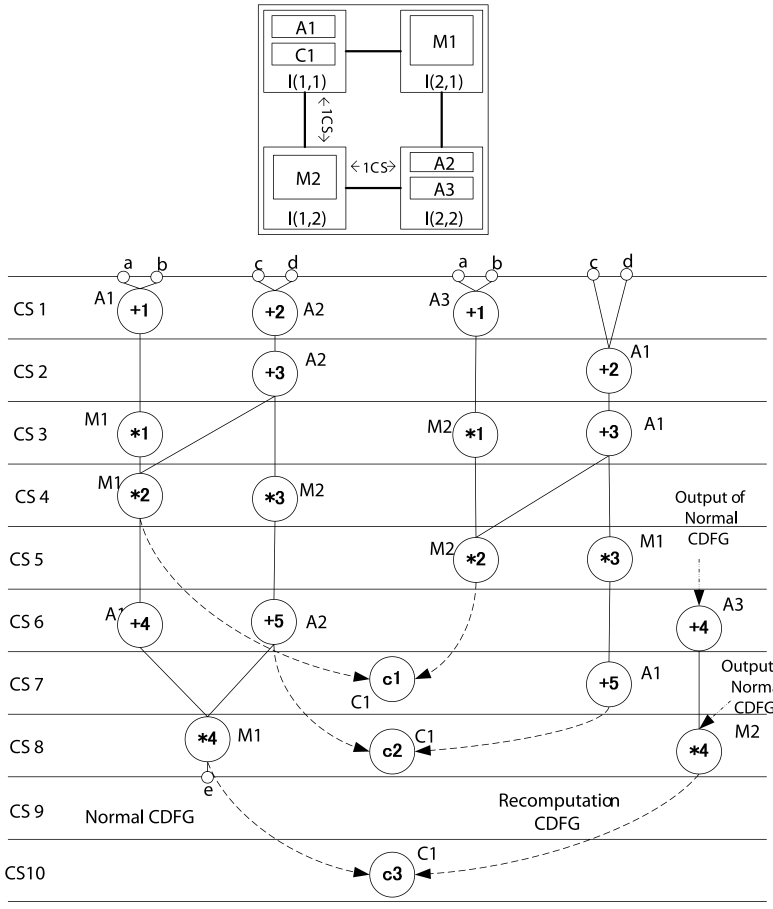


Fig. 16 Output of our approach.

4. Experimental Results

We have implemented our algorithm in C++. We have used the Intel Xeon 3.0 GHz and 4 GB memory PC. We have applied our algorithm to FIR (75 nodes), DCT (48 nodes), EWF (34 nodes), EWF3 (102 nodes), and PARKER (22 nodes

Table 2 Cost and delay of functional units.

	Cost	Delay [ns]
Adder (ADD)	1	1.32
Substractor (SUB)	1	1.33
Multiplier(MUL)	2	2.7
Comparator (COMP)	1	0.6
AND (AND)	1	0.03
Shifter (SHIFT)	1	0.55
Memory unit (MEM)	1	2.7

Table 3 Experimental results (control Steps).

			Normal CDFG only	Wu's approach [8]				Our algorithm			
Input CDFG	#Islands $N \times M$	Capacity constraint	Control steps	Control steps	#Inserted comparison nodes	Time overhead	CPU time [sec]	Control steps	#Inserted comparison nodes	Time overhead	CPU time [sec]
DCT	2×2	2	14	29	12	107%	111	27	8	92%	110
DCT	2×3	2	13	47	36	261%	167	22	14	69%	164
EWF	1×2	2	19	35	9	84%	56	32	3	68%	56
PARKER	1×1	4	7	13	5	85%	28	13	4	85%	28
PARKER	1×2	2	9	14	5	55%	55	14	4	55%	55
EWF3	2×2	2	53	62	25	16%	122	61	14	15%	121
EWF3	2×3	2	53	60	11	13%	176	60	10	13%	176
FIR	2×3	2	24	64	51	166%	176	31	18	29%	165
Average	—	—	24	41	19	69%	111	33	9	35%	109

Table 4 Experimental results (area).

Input CDFG	Algorithm	Maximum area island $I(x,y)$	Area [μm^2]	Controller Area [μm^2]	#Regs	#MUXs	Allocated functional units
DCT	Normal CDFG only	(1,1)	10836	661	10	21	MUL
2×2	Wu's approach [8]	(2,2)	14110	1418	18	62	ADD $\times 2$
	Our algorithm	(2,2)	13573	1409	15	65	ADD $\times 2$
DCT	Normal CDFG only	(2,2)	8737	813	10	40	ADD $\times 2$
2×3	Wu's approach [8]	(2,3)	13121	1528	10	73	ADD, COMP
	Our algorithm	(2,2)	9655	1059	10	46	ADD$\times 2$
EWF	Normal CDFG only	(1,1)	5750	289	2	3	MUL
1×2	Wu's approach [8]	(1,2)	13560	1319	14	46	ADD $\times 2$
	Our algorithm	(1,2)	10511	1227	12	47	ADD $\times 2$
PARKER	Normal CDFG only	(1,1)	4681	254	8	6	ADD $\times 2$, SUB, COMP
1×1	Wu's approach [8]	(1,1)	10451	489	17	30	ADD $\times 2$, SUB, COMP
	Our algorithm	(1,1)	8092	466	12	22	ADD $\times 2$, SUB, COMP
PARKER	Normal CDFG only	(1,1)	2619	244	4	4	SUB, COMP
1×2	Wu's approach [8]	(1,2)	6577	446	13	14	ADD $\times 2$
	Our algorithm	(1,1)	5771	340	8	20	SUB, COMP
EWF3	Normal CDFG only	(1,2)	10297	1541	9	50	ADD $\times 2$
2×2	Wu's approach [8]	(1,2)	18923	2279	15	105	ADD $\times 2$
	Our algorithm	(1,2)	16854	2130	13	93	ADD $\times 2$
EWF3	Normal CDFG only	(1,2)	10297	1541	9	50	ADD $\times 2$
2×3	Wu's approach [8]	(1,2)	11837	1668	7	68	ADD $\times 2$
	Our algorithm	(1,2)	11167	1627	9	57	ADD $\times 2$
FIR	Normal CDFG only	(1,1)	7089	428	5	6	MUL
2×3	Wu's approach [8]	(2,3)	16553	1792	11	109	ADD, COMP
	Our algorithm	(1,3)	8476	1105	7	35	ADD, MEM

including a branch node and a join node). **Table 2** shows the cost and delay time of FUs used. All of them are assumed to have 16-bit width under the 90 nm technology node. We set the clock period to be 3 ns. We also set the wire delay coefficient $C_d = 1\,ns$. Scheduling/binding of normal CDFGs and their functional unit allocation are given by using MCAS [5].

We have compared required control steps between our algorithm and Wu's approach [8]. Since Wu's approach does not have a functional unit allocation step, the functional unit allocation given by Step 2 of our algorithm is used.

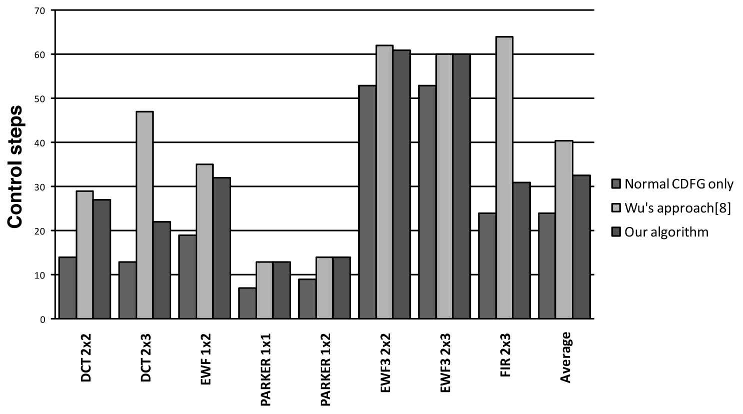


Fig. 17 Required control steps.

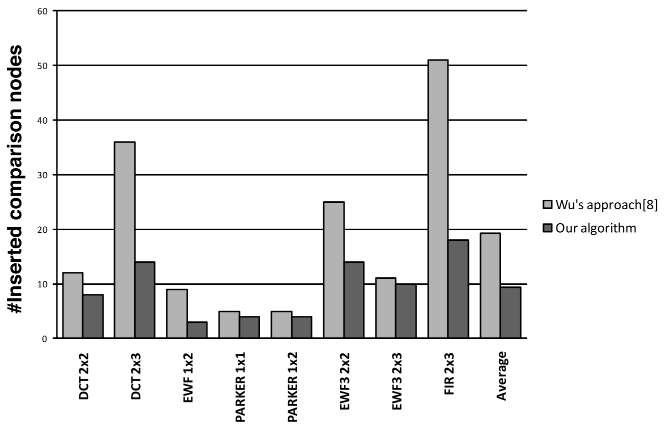


Fig. 18 Number of comparison nodes.

After that, recomputation CDFG is scheduled/bound by using Wu's algorithm.

Table 3 and **Table 4** show experimental results. Required control steps of normal CDFG, Wu's approach [8], and our algorithm are also shown in **Fig. 17**. The number of inserted comparison nodes are shown in **Fig. 18**. The first to the third columns of Table 3 show the input CDFG, the number of given islands, and capacity constraint. Time overhead shows the increasing rate of required control steps between the scheduling of only normal CDFG and scheduling of both normal and recomputation CDFGs. The first to the third columns of Table 4 show input CDFG, an algorithm used, the island $I(x, y)$ giving maximum area. The fourth to the eighth columns of Table 4 show the area of $I(x, y)$, controller area in $I(x, y)$, the number of registers, the number of multiplexers, and functional unit allocation in $I(x, y)$. These results demonstrate that our algorithm decreases up to 53% control steps compared with Wu's approach [8]. The number of comparison nodes inserted by edge-breaks is reduced by up to 66%. This is because Wu's approach [8] breaks all possible edges, but our approach considers the actual scheduling/binding of comparison nodes. Our algorithm can reduce redundant edge-breaks as well as insertion of comparison nodes. As Table 4 indicate, our algorithm reduces the maximum island area compared with Wu's approach. Our algorithm reduces the number of required control steps as well as comparison nodes, and thus reduces the required registers, MUXs and comparators.

5. Conclusion

In this paper, we proposed a fault-secure high-level synthesis algorithm for an RDR architecture. As device feature size is decreased, there arise so many its related problems. Our algorithm gives one of the solutions to these problems by considering fault caused by soft errors on logic circuits and interconnection delays. The experimental results show that our algorithm can reduce required control steps by up to 53% compared with the conventional approach.

The fault-secure design of this paper duplicates the entire CDFG. In the future, we need to determine subgraphs in the input CDFG necessary for fault-secure design and perform fault-secure high-level synthesis only for them.

Acknowledgments This research is supported in part by KAKENHI (22300019) and KDDI foundation.

References

1) Antola, A., Piuri, V. and Sami, M.: High-level synthesis of data paths with concurrent error detection, *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Systems*, pp.292–300 (1998).
2) Constantinescu, C.: Impact of deep submicron technology on dependability of VLSI circuits, *Proc. Dependable Systems and Networks*, pp.205–209 (2002).

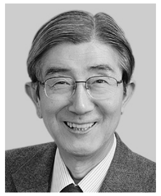
3) Kastensimidt, F.L., Sterpone, L., Carro, L. and Reorda, M.S.: On the optimal design of triple modular redundancy logic for SRAM-based FPGAs, *Proc. Conference on Design, Automation and Test in Europe*, Vol.2, pp.1290–1295 (2005).
4) Mathur, F.P. and Avizienis, A.: Reliability analysis and architecture of a hybrid-redundant digital system: generalized triple modular redundancy with self-repair, *Proc. AFIPS Joint Computer Conferences*, pp.375–383 (May 1970).
5) Cong, J., Fan, Y., Yang, X. and Zhang, Z.: Architecture and synthesis for multi-cycle communication, *Proc. 2003 International Symposium on Physical Design*, pp.190–196 (Apr. 2003).
6) Cong, J., Fan, Y., Han, G., Yang, X. and Zhang, Z.: Architectural synthesis integrated with global placement for multi-cycle communication, *Proc. International Conference on Computer-Aided Design*, pp.536–543 (Nov. 2003).
7) Cong, J., Fan, Y. and Zhang, Z.: Architecture-level synthesis for automatic interconnect pipelining, *Proc. Design Automation Conference*, pp.602–607 (Jun. 2004).
8) Wu, K. and Karri, R.: Fault secure datapath synthesis using hybrid time and hardware redundancy, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.23, No.10, pp.1476–1484 (2004).
9) Shicakumar, P., Kistler, M., Keckler, S.W., Burger, D. and Alvisi, L.: Modeling the effect of technology trends on the soft error rate of combinational logic, *Proc. International Conference on Dependable System and Networks*, pp.389–398 (2002).
10) Buchner, S., Baze, M., Brown, D., McMorrow, D. and Melinger, J.: Comparison of error rates in combinational and sequential logic, *IEEE Trans. Nuclear Science*, Vol.44, pp.2209–2216 (Dec. 1997).
11) Mitra, S., Seifert, N., Zhang, M., Shi, Q. and Kim, K.S.: Robust system design with built-In soft-error resilience, *Computer*, Vol.38, No.2, pp.43–52 (2005).
12) Tosun, S., Mansouri, N., Arvas, E., Kandemir, M. and Xie, Y.: Reliability-centric high-level synthesis, *Proc. Conference on Design, Automation and Test in Europe*, Vol.2, pp.1258–1263 (2005).

(Received November 26, 2010)
(Revised March 2, 2011)
(Accepted April 24, 2011)
(Released August 10, 2011)

(Recommended by Associate Editor: *Hiroshi Saito*)

Sho Tanaka received his B. Eng. degree from Waseda University in 2010 in Computer Sience. He is presently working toward M. Eng. degree there. His research interests are high-level synthesis and reliable LSI design.

Masao Yanagisawa received his B. Eng., M. Eng., and Dr. Eng. 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 Computer Science and Engineering. His research interests are combinatorics and graph theory, computational geometry, VLSI design and verification, and network analysis and design. He is a member of IEEE, ACM, and IEICE.



Tatsuo Ohtsuki received his B. Eng., M. Eng., and Dr. Eng. degrees from Waseda University in 1963, 1965, and 1970, respectively, all in Electrical Engineering. In 1965, he joined the NEC Corporation Ltd., Tokyo, Japan. From 1978 to 1980, he served as Research Manager, Application System Research Laboratory, at Central Research Laboratories. In 1980, he left NEC and joined Waseda University, where he is presently a Professor in the Department of Computer Science and Engineering. His research interests are algorithm and hardware engines for VLSI design and verification, computer algorithms for combinatorial problems, and network analysis/design. He is a fellow of IEEE and IEICE.

Nozomu Togawa received his B. Eng., M. Eng., and Dr. Eng. 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 VLSI design, graph theory, and computational geometry. He is a member of IEEE and IEICE.