Reliability-driven High-level Synthesis Algorithms for Distributed-register SoC Architectures

レジスタ分散型集積回路アーキテクチャを対象とした信頼性指向の高位合成に関する研究

February 2016

Kazushi KAWAMURA

川村 一志

Reliability-driven High-level Synthesis Algorithms for Distributed-register SoC Architectures

レジスタ分散型集積回路アーキテクチャを対象とした信頼性指向の高位合成に関する研究

February 2016

Waseda University

Graduate School of Fundamental Science and Engineering

Department of Computer Science and Engineering,

Research on Information System Design

Kazushi KAWAMURA

川村 一志

Contents

1 Introduction **1**

2 Related Works **5**
2.1 Introduction 5
2.2 Regular-distributed-register Architecture 5
2.3 Thermal-aware High-level Synthesis Techniques 7
2.4 Fault-secure High-level Synthesis Techniques 9
2.5 Conclusion 11

3 A Thermal-aware High-level Synthesis Algorithm **12**
3.1 Introduction 12
3.2 Problem Formulation 13
3.3 The Algorithm 15
3.3.1 Strategy 15
3.3.2 Overall synthesis flow 18
3.3.3 Energy-balanced FU binding (Step1) 18
3.3.4 Additional FU allocation (Step2) 25
3.4 Experimental Results 26
3.5 Conclusion 29

4 An Overhead Constraint-based Partially Redundant Fault-secure High-level Synthesis Algorithm **30**
4.1 Introduction 30
4.2 Problem Formulation 31
4.3 The Algorithm 36
4.3.1 Strategy 37
4.3.2 Overall synthesis flow 39
4.3.3 Scheduling/binding for duplicated operations (Step1) 39
4.3.4 Comparator binding (Step2) 44
4.3.5 Removing duplicated operations (Step3) 47

4.4 Experimental Results 48
4.5 Conclusion 50

5 A Low-overhead Fully Redundant Fault-secure High-level Synthesis Algorithm 52
5.1 Introduction 52
5.2 Problem Formulation 53
5.3 The Algorithm 56
5.3.1 Strategy 56
5.3.2 Overall synthesis flow 57
5.3.3 Scheduling/binding for duplicated operations with FU allocation and register synthesis 60
5.4 Experimental Results 70
5.5 Conclusion 71

6 Conclusion 72

Acknowledgment 74

List of Publications 78

List of Figures

2.1 RDR architecture [7]. 6
2.2 MCAS for RDR architecture [7]. 7
2.3 Technology node and power density [20]. 8
2.4 The thermal-aware HLS algorithm through resource allocation optimization [27]. 9
2.5 Technology node and soft error rate [22]. 10

3.1 An example of a DFG. 13
3.2 An example of interconnect delay aware operation scheduling. . . . 14
3.3 The proposed thermal-aware HLS algorithm. 17
3.4 An example of *Binding tree*. 19
3.5 An input example of thermal-aware HLS for RDR architecture. . . 22
3.6 Process of generating *Binding tree*. 23
3.7 Two FU binding solutions. 24
3.8 The partition algorithm. 25
3.9 Temperature distribution when applying each algorithm to DCT (the number of islands: 3×2). 28
3.10 The results of FU allocation when applying each algorithm to DCT (the number of islands: 3×2). 28

4.1 An example of fully redundant fault-secure scheduling/binding. . . . 32
4.2 An example of partially redundant fault-secure scheduling/binding. 33
4.3 An input example of partially redundant fault-secure HLS for RDR architecture. 36
4.4 The proposed partially redundant fault-secure HLS algorithm. . . . 38
4.5 Scheduling/binding of re-computational DFG by ignoring re-computational edges (Step(1.2)). 41
4.6 Removing operation nodes violating the step constraint (Step(1.3)). 42
4.7 Restoring re-computational edges (Step(1.4)). 42
4.8 Re-scheduling/re-binding of re-computational DFG (Step(1.5)). . . 44

4.9 Allocating comparators (Step(2.1)). 45
4.10 Inserting and scheduling/binding of comparison nodes (Step(2.2)). . 46
4.11 Adjusting comparison nodes (Step(2.3)). 46
4.12 Scheduled/bound re-computational DFG output from our partially redundant fault-secure HLS algorithm. 48
4.13 FU placement output from our partially redundant fault-secure HLS algorithm. 48

5.1 Impact of an area overhead on circuit performace in RDR architecture. The size of islands ($W \times H$) has a direct effect on a given clock period constraint (T_{clk}) and an interconnect delay between two islands ($D_c(i_p, i_q)$). 53
5.2 Comparison of conventional and our fault-secure HLS algorithms. . 56
5.3 The proposed fully redundant fault-secure HLS algorithm. 58
5.4 An input example of fully redundant fault-secure HLS for RDR architecture. 59
5.5 Register binding. 59
5.6 Scheduling/FU binding '+6'' to $\langle S5, A2\rangle$. 64
5.7 Area cost after scheduling/FU binding of '+6''. 65
5.8 Scheduling/binding '+5'' to $\langle S4, M4\rangle$. 66
5.9 Scheduling/binding '+5'' to $\langle S3, M1\rangle$. 67
5.10 Scheduling/binding '+5'' to $\langle S4, M1\rangle$. 68
5.11 An output example of Step(1.2). 69

List of Tables

3.1 Capacity cost, delay, energy, and leakage power of modules [3]. . . . 26
3.2 Experimental results. 27

4.1 Area and delay of modules. 49
4.2 The set of constraints. 49
4.3 Experimental results (reliability). 50
4.4 Experimental results (overhead). 51

5.1 Experimental results. 70
5.2 Comparison of area and performance. 70

Chapter 1

Introduction

Advanced semiconductor process technology expands the use of system-on-a-chip (SoC) which integrates a series of desired functions into one chip. While SoCs make electronic devices small size and high performance, recent market trend demands to integrate a wide variety of functions into them and hence complicates their design process. High-level synthesis (HLS) is an important technique to reduce the cost of designing complicated systems or ICs. Conventional IC design flow begins with a register-transfer-level (RTL) description which is written by designers with consideration for synchronization by a clock signal. In a design flow using HLS, its designing cost can be reduced since an RTL description is automatically generated from a behavioral description (written in such as C or C++) by applying an HLS algorithm. Various design factors, like performance, power, and reliability, are globally optimized by taking them into account in HLS algorithms. However, most HLS algorithms reported so far tend to separate low-level design phases from their algorithms and cannot achieve sufficient optimization of these factors.

With process technology scaling, the ratio of interconnect delays to circuit delays becomes increasingly dominant. This trend creates a serious problem faced by HLS and may result in performance degradation of ICs. In an HLS algorithm which separates low-level design phases and ignores the impact of interconnect delays, an excessive amount of timing margins might be required. While several HLS techniques have tried to avoid performance degradation by utilizing feed-backed interconnect-delay information, these approaches may complicate performance optimization in HLS phase. Utilizing distributed-register (DR) architectures is one of the effective and reasonable solutions to tackle this interconnect-delay problem, and several kinds of architectures have been studied so far [2,3,7,8,11,12,18,19]. In the DR architecture, an entire circuit is divided into small blocks and floorplanning is performed at the block level. In an HLS algorithm for DR architectures, interconnect delays can be appropriately handled based on the estimated inter-block

interconnect delays, which expects to improve circuit performance.

The reliability of SoCs is one of the most important design factors. With growing use of SoCs, highly-reliable systems are required in various situations such as vehicle systems, medical devices, and social infrastructure. On the other hand, advanced process technology creates many technological challenges related to reliability. Increasing heat problem in IC chips is one of these challenges. Hot-spots, where a chip is locally too much heated, are main cause of the problem. In terms of reliability, reducing hot-spot temperature is quite important since high temperature reduces the switching speed of transistors as well as accumulates severe damage in the long run [10, 23]. Several thermal-aware HLS algorithms have been proposed [9, 13, 15–17, 27], which can effectively reduce the temperature of hot-spots. However, their algorithms are not reasonable because their high- and low-level design phases are separated and hot-spot temperature is tried to reduce in HLS phase based on feed-backed information obtained through a thermal simulation for post-designed circuits. Moreover, all the conventional algorithms have not considered the impact of interconnect delays, which degrade circuit performance severely. By handling module floorplan and reducing the temperature of hot-spots in HLS phase, a reasonable HLS algorithm, which considers the impact of both hot-spot and interconnect delay, can expect to be realized.

As process technology advances, increasing soft error rate (SER) becomes another challenge in the aspect of reliability [4, 21]. A soft error is induced when a radiation particle strikes on a circuit node, which may cause a transient fault and partly upset the circuit function. Since current IC manufacturing technology cannot completely prevent the faults, fault-secure design methodologies are strongly required for highly-reliable systems. The overhead of performance and/or area is essentially a serious problem in fault-secure designs and thus circuits generated by a fault-secure design methodology should be low-overhead as much as possible. Incorporating a fault-secure design into HLS is one of the effective solutions to realize low-overhead fault-secure design, and several algorithms have been proposed [5, 24–26]. Conventional fault-secure HLS algorithms [5, 24, 26] are based on concurrent error detection (CED) schemes, which can detect a transient fault by duplicating operations and comparing their results. While they have tried to reduce the overhead with well-suited resource sharing, the area overhead is evaluated insufficiently because they have only considered the number of required functional units. To evaluate the overhead sufficiently and then reduce it, it is important to consider module floorplan more minutely in a fault-secure HLS algorithm.

In this dissertation, a thermal-aware HLS algorithm and fault-secure HLS algorithms for DR architectures are newly proposed. As discussed before, highly-reliable SoCs or ICs become increasingly important, and reducing additional cost

or overhead is so much required for their designing methodologies. In an HLS flow using DR architectures, additional design information such as interconnect delay, area, and hot-spot can be estimated more accurately at HLS phase based on the result of module floorplanning. The estimated information is utilized for optimizing various design factors in each step of HLS flow, and HLS algorithms achieving high-reliability at low-cost or low-overhead are constructed. All the our proposed algorithms have been implemented and evaluated computationally. Compared to a conventional approach with no thermal consideration, our proposed thermal-aware HLS algorithm reduces the peak temperature inside a chip by up to 15.5% with no performance/area overhead. Our firstly proposed fault-secure HLS algorithm improves reliability by up to 24% with no performance/area overhead compared to a conventional approach with no fault-security consideration, and our secondly proposed algorithm reduces area by up to 47% and improves performance by up to 41% compared to a conventional fault-secure algorithm.

This dissertation is organized as follows:

Chapter 2 [**Related Works**] presents the key techniques related to our research. First we introduce regular-distributed-register architecture (RDR architecture) which is one of the DR architectures and the HLS algorithm for this architecture. Next we introduce conventional leading algorithms of thermal-aware HLS which focuses on reducing hot-spot temperature and fault-secure HLS which is based on CED scheme. **Chapter 3** [**A Thermal-aware High-level Synthesis Algorithm**] proposes a novel thermal-aware HLS algorithm for DR architectures. Not only interconnect delays but also hot-spots can be estimated at HLS phase by utilizing DR architectures, which enables us to deal with hot-spots in HLS flow and hence expects to reduce the designing cost drastically. Our proposed algorithm balances the energy consumption among divided blocks by focusing on the number of operations executed in each block. Balancing the energy consumption can reduce the temperature of hot-spots with no performance degradation. Allocating new functional units further balances the energy consumption. Experimental results demonstrate that our proposed algorithm reduces the peak temperature inside a chip by up to 15.5% compared to a conventional HLS algorithm for DR architectures. **Chapter 4** [**An Overhead Constraint-based Partially Redundant Fault-secure High-level Synthesis Algorithm**] proposes a novel fault-secure HLS algorithm for DR architectures which partially duplicates operations based on constraints of performance/area overhead. Within a set of performance and area constraints, this algorithm attempts to maximize reliability which is evaluated by the output probability when a soft error occurs. Our proposed algorithm adopts a greedy search method by removing duplicated operations. Experimental results demonstrate that our proposed algorithm improves reliability by up to 24%

without any performance/area overhead compared to a conventional HLS algorithm for DR architectures. **Chapter 5 [A Low-overhead Fully Redundant Fault-secure High-level Synthesis Algorithm]** proposes a novel low-overhead fault-secure HLS algorithm for DR architectures which fully duplicates operations. By utilizing DR architectures, both the block area and the interconnect delay between each block can be evaluated at HLS phase. This evaluated information is used for optimizing circuit area/performance in HLS flow. In contrast to conventional approaches, our proposed algorithm adopts an integrative approach which concurrently performs scheduling, binding, allocation, and register synthesis. Since this approach monitors the cost of not only functional units but also registers and multiplexers during scheduling/binding, the area/performance can be estimated accurately and then reduced. Experimental results demonstrate that our proposed algorithm reduces area by up to 47% and improves performance by up to 41% compared to a conventional fault-secure HLS algorithm for DR architectures. **Chapter 6 [Conclusion]** summarizes this dissertation and gives future works.

Chapter 2

Related Works

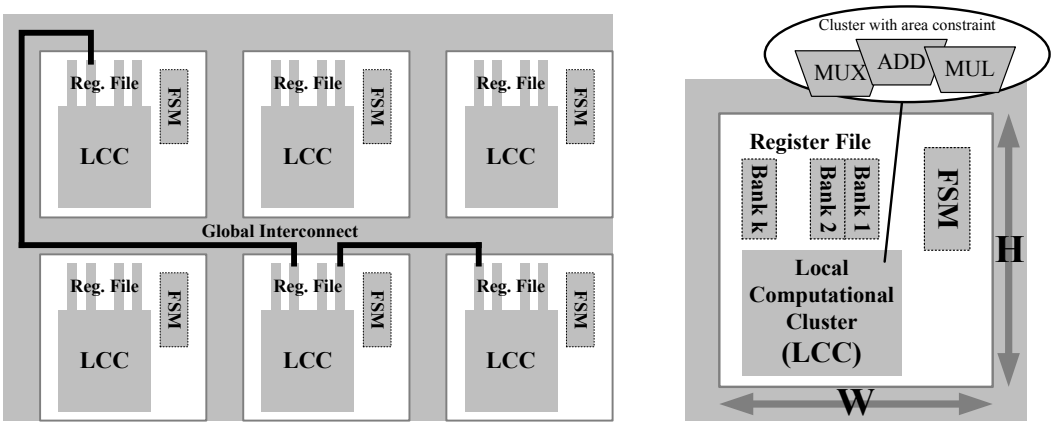
2.1 Introduction

In this chapter, we briefly discuss the key techniques related to our research. First we introduce regular-distributed-register (RDR) architecture which is one of the distributed-register (DR) architectures and the HLS algorithm for this architecture. Next we introduce conventional leading algorithms of thermal-aware HLS which focuses on reducing hot-spot temperature and fault-secure HLS which is based on concurrent error detection (CED) scheme.

2.2 Regular-distributed-register Architecture

Advanced semiconductor process technology increases the average interconnect delays. Moreover, it becomes impossible to transfer data across a chip in a single clock cycle as the operating frequency advances. For these reasons, considering interconnect delays in HLS flow becomes so important. Several HLS algorithms have been reported [2,3,7,8,11,12,18,19], which consider interconnect delay effects in their synthesis flows. They are based on DR architecture families which divide an entire circuit into small blocks such that the intra-island interconnect delay can be assumed to be zero. In HLS for DR architectures, floorplanning is performed at the block level and then the inter-block interconnect delays are estimated from the placement information.

As one of the DR architectures, regular-distributed-register (RDR) architecture has been proposed by Cong et al. in [7]. In the RDR architecture, a circuit is assigned to an array of small partitions called *islands*. In each island, a local computational cluster (LCC), local registers, and a finite state machine (FSM) are allocated, where LCC includes several functional units (FUs) such as adders



(a) 3×2 island-based RDR architecture. (b) Components of a single island.

Figure 2.1: RDR architecture [7].

and multipliers. Fig. 2.1 illustrates an example of RDR architecture. Considering the RDR architecture from an architectural point of view, its advantages and disadvantages are summarized as follows:

Advantages:

- Inter-island interconnect delays are estimated very accurately in HLS because of the regularity of architecture.
- Intra-island interconnect delays between FUs and registers can be reduced since registers are distributed to each island.
- Clock cycle time can be occupied by almost intra-island delays since inter-island communication can be realized by multicycle communication between registers.

Disadvantages:

- Some islands might have vacant spaces due to the regularity of architecture.

The HLS algorithm for RDR architecture called MCAS [7] is shown in Fig. 2.2. MCAS first determines the number of allocated FUs in the "Resource allocation" and then performs FU floorplanning in the "Scheduling-driven placement". After that, in the "Post-layout scheduling with rebinding", each operation is conclusively scheduled to an execution step and bound to an FU. We finally obtain an RTL

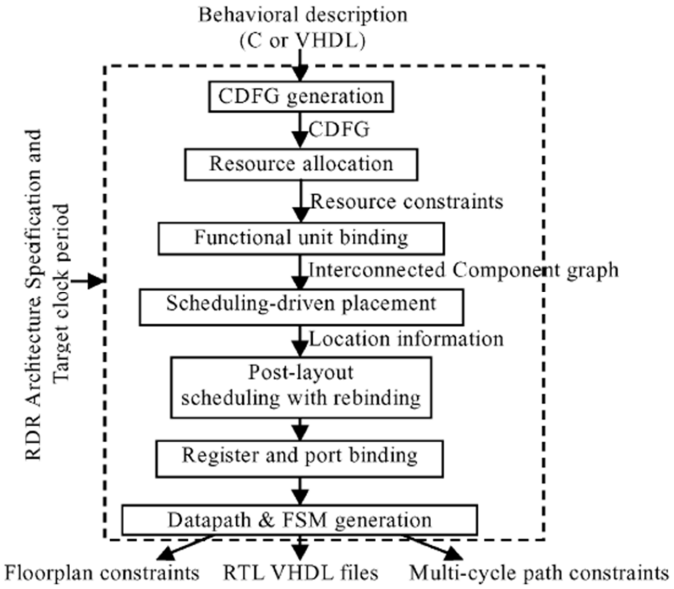


Figure 2.2: MCAS for RDR architecture [7].

circuit and floorplan constraints through the steps "Register and port binding" and "Datapath & FSM generation". The objective of MCAS is minimizing the number of required steps to execute the input behavior under a given specification of RDR architecture and clock period constraint.

2.3 Thermal-aware High-level Synthesis Techniques

With process technology scaling, increasing power density or energy density in IC chips is becoming a serious concern. The relationship between feature size and power density is shown in Fig. 2.3. Power density inside a chip is exponentially increasing as the technology node advances. Increased energy density generates a great deal of heat and thus increases chip temperature. High temperature is one of the factors having a significant negative impact on circuit reliability [10, 23]. Firstly, increasing chip temperature reduces the switching speed of transistors and leads to increase in the error rate of circuits. Secondly, high temperature can lead to permanent damage in the long run.

In general, energy density is spatially non-uniform across a chip. This results in uneven temperature distribution and hence we have so-called *hot-spots* where a chip is locally too much heated. Hot-spots are main cause of reliability degradation described above and it is quite necessary to reduce their temperature. Though low-energy design techniques have been reported by many researchers, they do not

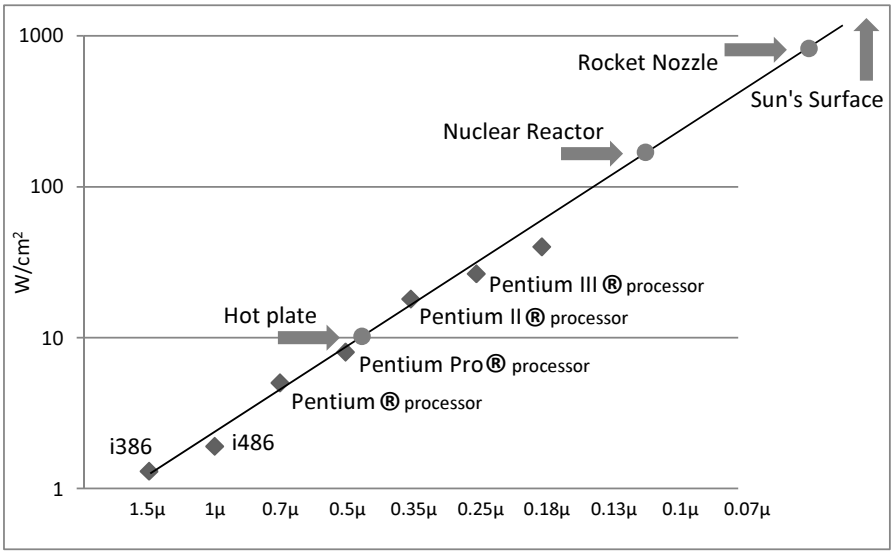


Figure 2.3: Technology node and power density [20].

specifically address energy density and thus do not reduce the temperature of hot-spots. In other words, thermal-aware design techniques which minimize the peak temperature inside a chip are quite required.

To effectively reduce hot-spot temperature, several thermal-aware HLS algorithms have been reported so far [9, 13, 15–17, 27]: A voltage island technique has been proposed in [9] for thermal management; A Dijkstra-like binding algorithm has been propsed in [17] to reduce the peak temperature; Lim et al. have proposed in [13] a binding algorithm based on the network flow method to reduce the peak temperature with the primary objective of minimizing the peak dynamic power and the secondary objective of minimizing the total dynamic power; Mukjerjee et al. have proposed techniques for controlling the peak temperature based on feed-backed temperature information [15, 16]. Their binding and scheduling algorithms move operations from the hottest to the coldest resources based on the resource temperatures obtained in floorplanning; Based on the HLS algorithm in [15], Yu et al. have proposed in [27] an allocation algorithm to reduce the difference in temperature among not only resources of the same types but also resources of the different types.

Fig. 2.4 shows the thermal-aware HLS algorithm proposed in [27]. In the algorithm, each of the HLS phases including operation scheduling, FU binding, and FU allocation is performed based on feed-backed temperature information obtained in the steps "Thermal-aware floorplanning" and "Thermal analysis with ISAC" of Fig. 2.4. It is therefore concerned that the designing cost can be high because the

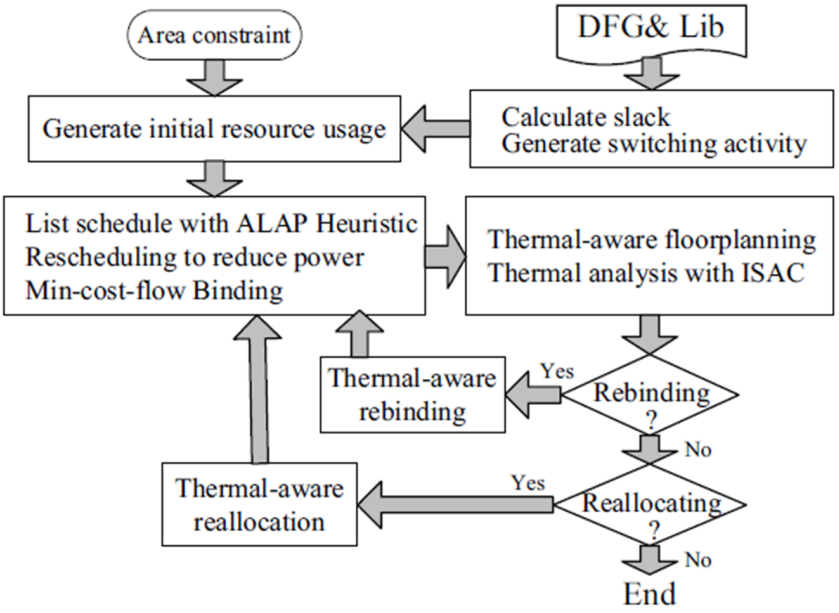


Figure 2.4: The thermal-aware HLS algorithm through resource allocation optimization [27].

thermal analysis (or thermal simulation) will need to be rum many times in the design flow. Moreover, conventional thermal-aware HLS algorithms such as [15] and [27] have used resource floorplanning only for obtaining temperature information and hence absolutely have not considered the impact of interconnect delays in HLS phases.

2.4 Fault-secure High-level Synthesis Techniques

With process technology scaling, increasing soft error rate (SER) becomes another serious concern [4, 21]. Fig. 2.5 shows the relationship between feature size and SER on memory and logic components. Memory components have been traditionally more susceptible to soft errors than logic components because soft errors on logic components can be masked by various effects such as logic masking, electric masking, and latch-window masking [22]. In order to deal with soft errors on memory components, many countermeasure techniques have been researched so far, and thus there have been effective and reasonable solutions such as error correction code (ECC) [6]. However, as shown in Fig. 2.5, logic components also susceptible to soft errors as the technology node advances. A transient fault may be caused when a soft error occurs in a logic component close to the clock edge.

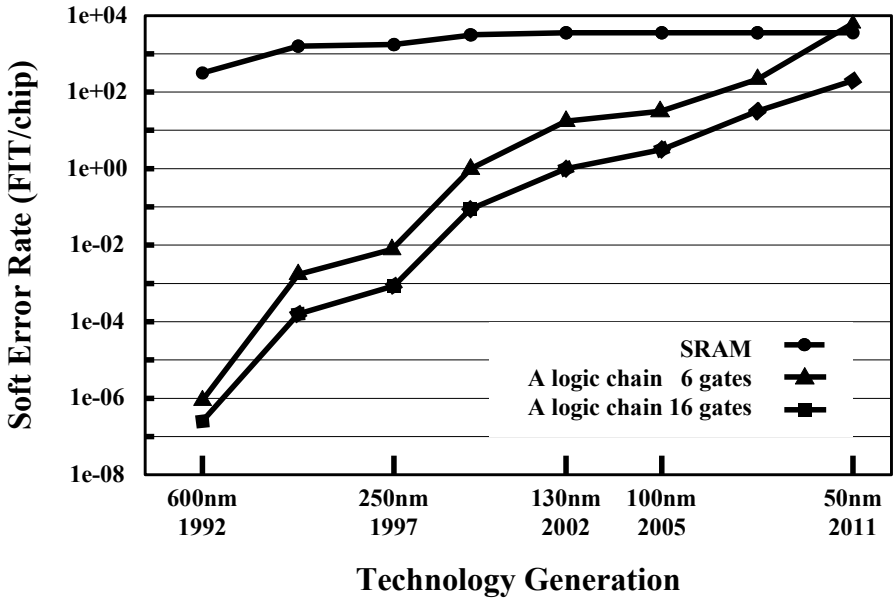


Figure 2.5: Technology node and soft error rate [22].

Since current IC manufacturing technology cannot completely prevent the faults, fault-secure design methodologies are strongly required.

In fault-secure designs, performance and/or area overhead is essentially a serious problem. Incorporating a fault-secure design into HLS is one of the effective solutions to realize low-overhead IC design, and several algorithms have been proposed [5, 24–26]. In [5, 24, 26], fault-secure HLS algorithms have been constructed by using CED schemes, which can detect a fault by duplicating operations and comparing their results. We call a set of original operations *normal-computation* and a set of duplicated operations *re-computation*.

Antola et al. have proposed in [5] an FDS-based fault-secure scheduling algorithm with the objective of minimizing the area overhead. Wu et al. have proposed in [26] a CED-based fault-secure HLS algorithm selectively breaks data dependencies between re-computational operations so that the performance overhead can be minimized. Breaking data dependencies enables us to improve the FU sharing between normal-computational and re-computational operations. Tanaka et al. have proposed in [24] a fault-secure HLS algorithm for RDR architecture by expanding the technique developed by [26]. Additionally, [14] has reported the relationship between reliability and the number of required FUs when a normal-computation is partially duplicated. We can expect to improve reliability with quite small overhead by introducing the idea of partial duplication. While the conventional CED-based fault-secure HLS algorithms try to reduce the performance/area overhead with

well-suited FU sharing, the area overhead is evaluated insufficiently because they have only considered the number of required FUs.

2.5 Conclusion

In this chapter, we have first introduced RDR architecture and the HLS algorithm for RDR architecture called MCAS. Next we have introduced conventional leading algorithms of thermal-aware HLS and CED-based fault-secure HLS.

Chapter 3

A Thermal-aware High-level Synthesis Algorithm[1]

3.1 Introduction

A thermal-aware HLS algorithm for DR architectures is proposed in this chapter. In our HLS algorithm, we use RDR architecture which is one of the DR architectures. In RDR architecture, a circuit is assigned to an array of small partitions called *islands* as illustrated in Fig. 2.1. Considering the amount of heat generated in an island strongly depends on its energy density, we expect that balancing the energy consumption among islands directly leads to reduce the temperature of hot-spots. Our proposed algorithm balances the energy consumption among islands by focusing on the number of operations executed in each island, which effectively reduces the temperature of hot-spots. Some of the islands in RDR architecture may have vacant spaces due to the regularity of architecture. In our proposed algorithm, by utilizing the vacant spaces and allocating additional new functional units into them, further balancing the energy consumption and thus reducing the hot-spot temperature are achieved. Experimental results demonstrate that our proposed algorithm reduces the peak temperature inside a chip by up to 15.5% with no performance degradation compared to a conventional HLS algorithm for RDR architecture.

[1]Technical contents in this chapter have been presented in the publications ⟨1⟩, ⟨6⟩, ⟨17⟩, and ⟨18⟩.

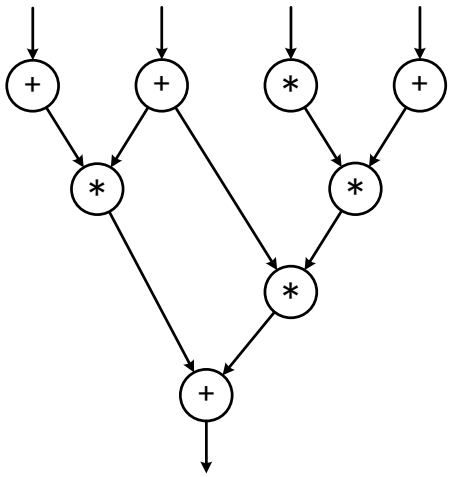


Figure 3.1: An example of a DFG.

3.2 Problem Formulation

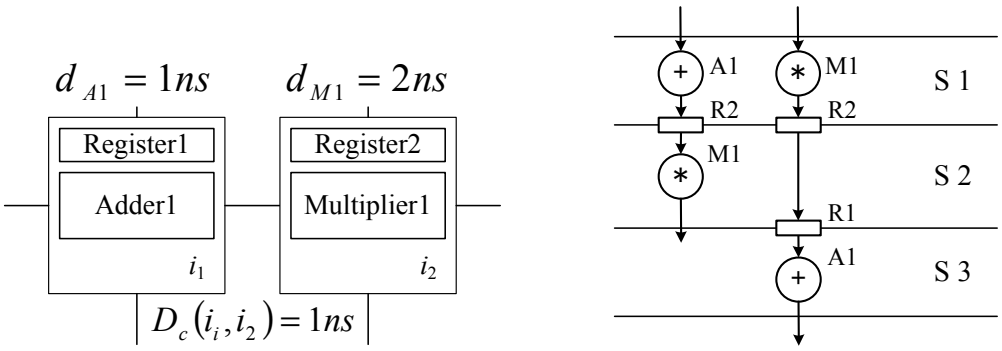
In HLS, behavioral description used as input is transformed into graphical representation like a control-data flow graph (CDFG) or a data-flow graph (DFG). Hereafter we use a DFG as an input for simplicity. Note that the discussion below can be easily extended to a CDFG. A DFG $G = (V, E)$ is represented by a directed graph, where V is a set of operation nodes and E is a set of edges which show the data dependencies between two operation nodes. Fig. 3.1 shows an example of a DFG.

In RDR architecture, an entire circuit is assigned to $N \times M$ array of islands. Let $I(x, y)$ be an island located at x-th column and y-th row of the array, where $1 \leq x \leq N$ and $1 \leq y \leq M$. All the islands assume to be square and have the same size. A functional unit (FU) fu is allocated to one of the islands and has a delay of d_{fu}. Each island $i = I(x, y)$ has a set of local registers $\{R(i, 1), R(i, 2), \dots\}$. In the following discussion, one of the registers allocated to the island i is shown as $R(i, *)$. Let i_1 and i_2 be two islands $I(x_1, y_1)$ and $I(x_2, y_2)$, respectively. An interconnect 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 it is given by

$$D_c(i_1, i_2) = C_d \times (|x_1 - x_2| + |y_1 - y_2|)^2 \tag{3.1}$$

where C_d shows the constant interconnect delay coefficient.

Let fu_1 be one of the FUs allocated to the island i_1. Assume that the output of fu_1 is continuously used by an FU allocated to the island i_2. Let T_{clk} be the given clock period constraint. For the clock period constraint T_{clk} and the FU fu_1



(a) FU placement. (b) Scheduling.

Figure 3.2: An example of interconnect delay aware operation scheduling.

which has a delay of d_{fu_1}, the computation by fu_1 can be done in $s_{fu_1} = \left\lceil \frac{d_{fu_1}}{T_{clk}} \right\rceil$ steps. When

$$D_c(i_1, i_2) + d_{fu_1} \leq s_{fu_1} \times T_{clk} \tag{3.2}$$

holds, the computation by fu_1 and its output data transfer to a register $R(i_2, *)$ are done in s_{fu_1} steps. On the other hand, when

$$D_c(i_1, i_2) + d_{fu_1} > s_{fu_1} \times T_{clk} \tag{3.3}$$

holds, the computation by fu_1 is done in s_{fu_1} steps and its output is stored into a register $R(i_1, *)$ at the s_{fu_1}-th step. After that the output of fu_1 is transferred from $R(i_1, *)$ to $R(i_2, *)$ using $\left\lceil \frac{D_c(i_i, i_2)}{T_{clk}} \right\rceil$ steps.

Example 3.1. *Fig. 3.2 shows an example of operation scheduling with interconnect delay consideration. We assume the FU placement shown in Fig. 3.2a as a part of RDR architecture. In this example, the delays of the adder 'A1' and the multiplier 'M1' are assumed to be* $d_{A1} = 1$ ns *and* $d_{M1} = 2$ ns*, respectively, and the data transfer delay between the two adjacent islands is assumed to be* $D_c(i_1, i_2) = 1$ ns*. The given clock period constraint is assumed to be* $T_{clk} = 2$ ns*.*

When we try to schedule two continuous operations, the 1st is addition and the 2nd is multiplication, the operation by 'A1' and the data transfer to the register 'R2' are completed in a single step ('S1' shown in Fig. 3.2b) because Eq. (3.2) is satisfied. Then the 2nd operation can be scheduled to the step 'S2' as shown in the left side of Fig. 3.2b. On the other hand, when we try to schedule two continuous operations, the 1st is multiplication and the 2nd is addition, only the operation by 'M1' is completed and its output is stored into the register 'R2' in the step 'S1' because Eq. (3.3) is satisfied. In this

case, the output of 'M1' is transferred from 'R2' to 'R1' using $\left\lceil \frac{D_c(i_i, i_2)}{T_{clk}} \right\rceil (= 1)$ *step. Therefore, the step 'S2' is used only for data transfer, and then the 2nd operation must be scheduled to later step than the step 'S2' as shown in the right side of Fig. 3.2b.* □

Every island in RDR architecture has a capacity C and every FU fu has a capacity cost c_{fu}. Let $Fu(i)$ be the set of FUs allocated to an island i. Any island i satisfies

$$\sum_{fu \in Fu(i)} c_{fu} \leq C. \tag{3.4}$$

In other words, an FU fu_j can be newly allocated to an island i when

$$c_{fu_j} + \sum_{fu \in Fu(i)} c_{fu} \leq C \tag{3.5}$$

holds.

Now we define a thermal-aware HLS problem for RDR architecture as follows:

Definition 3.1. *For a given DFG* $G = (V, E)$*, specification of RDR architecture (the number of islands* $N \times M$ *and the island capacity* C*), library of FUs, and clock period constraint* T_{clk}*, our thermal-aware HLS problem for this RDR architecture is to schedule and bind its DFG* G *and to allocate FUs to each island with the objective of minimizing the peak temperature.* □

3.3 The Algorithm

In this section, we propose a novel thermal-aware HLS algorithm for RDR architecture. First we discuss the strategy and then propose our algorithm.

3.3.1 Strategy

In general, temperature in a module such as an FU or a register is affected by its energy consumption, area, and floorplanning. If two different modules with different sizes consume the same amount of energy, the module with the smaller size can have higher energy density and hence can be heated more. Even if a module has low energy density, it will be heated when its surroundings are heated. When we estimate hot-spots, it is required to know the accurate floorplanning of modules and to simulate their on-chip temperatures, which is difficult to be achieved in HLS phase or is computationally expensive.

In RDR architecture, since each module is allocated to one of the islands and every island has the same size, we can easily expect that an island having high

energy consumption has high energy density and hence will be heated much. By balancing the energy consumption among islands when executing the given DFG, the energy consumed in an island having the highest energy consumption can be reduced. Therefore, by focusing on the intra-island energy consumption and maximally balancing it among islands, we can expect to minimize the peak temperature inside a chip.

High-level synthesis for RDR architecture consists of operation scheduling, FU binding, FU allocation, and floorplanning. The next problem here is which steps should be focused on and what strategies should be applied to each of them. Each step is focused on as follows:

Operation scheduling:

Compared to the conventional scheduling algorithm for RDR architecture [7], changing scheduling results may cause alomst no effect on reducing temperature since clock period is too short to reduce it. Furthermore, since considering interconnect delays in scheduling is very complicated, we may result in performance degradation even if developing new scheduling algorithms.

FU binding:

The energy consumption in each island is directly affected by binding results. FU binding is the most important step when we balance the energy consumption among islands.

FU allocation:

Generally, decreasing the number of FUs results in performance degradation. On the other hand, energy consumption can be further balanced by increasing the number of FUs since per unit energy consumption can be decreased.

Floorplanning:

In interconnect-delay aware HLS, changing the original floorplanning may result in performance degradation.

Based on the strategy discussed above, in HLS for RDR architecture, we propose thermal-aware FU binding and FU allocation algorithms.

For a given DFG and its initial FU placement, we first apply a conventional HLS algorithm for RDR architecture called MCAS [7] and obtain a scheduling solution. Original FU allocation and floorplanning will not be changed in our

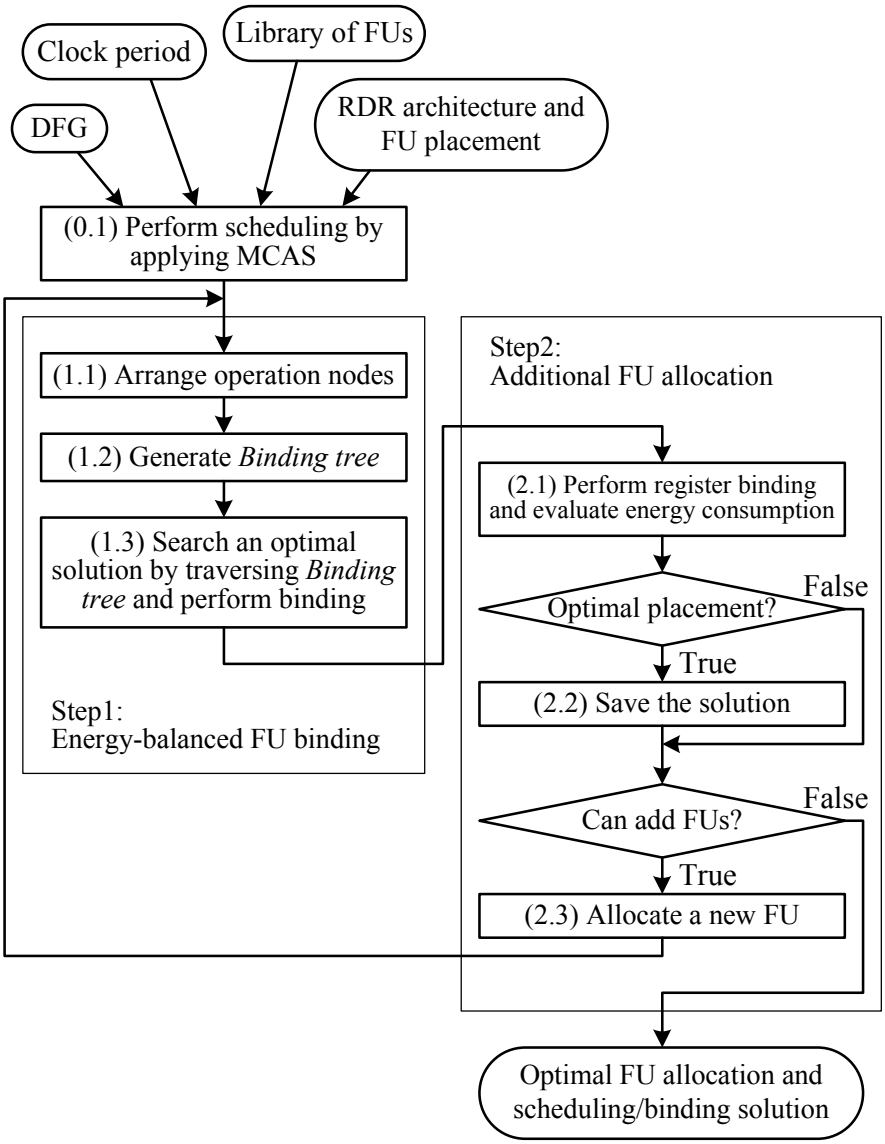


Figure 3.3: The proposed thermal-aware HLS algorithm.

synthesis flow in order not to cause any performance degradation, but we may allocate additional new FUs to islands with vacant spaces. After that we perform thermal-aware FU binding with the objective of balancing energy consumptions among islands (Step1). Moreover, new FUs will be allocated if they satisfy the capacity constraint and can reduce the maximum energy consumption in RDR architecture (Step2).

3.3.2 Overall synthesis flow

Fig. 3.3 shows our proposed algorithm. The main inputs to our algorithm are a DFG and its initial FU placement which is obtained in MCAS [7]. Note that MCAS first performs FU allocation and floorplanning and then we utilize its intermediate output as our input.

For a given DFG and its FU placement, we first generate a scheduling solution by applying MCAS. Based on this scheduling solution, we perform thermal-aware FU binding with the objective of balancing energy consumption among islands. If there are some vacant islands and we can add new FUs to them, we perform FU binding again. The iteration will be finished when all the pattern of adding FUs are evaluated through FU binding and register binding. We finally output an optimal FU allocation/floorplanning and scheduling/binding solution for its FU placement.

The algorithm shown in Fig. 3.3 mainly consists of two steps: energy-balanced FU binding (Step1) and additional FU allocation (Step2). As discussed in Section 3.3.1, we do not change the scheduling solution and the initial FU placement which are generated by MCAS in order not to cause any performance degradation in Step1 and Step2, but focus on FU binding and new FU allocation.

3.3.3 Energy-balanced FU binding (Step1)

For the FU placement in RDR architecture and the scheduling solution generated by MCAS, Step1 performs FU binding with the objective of balancing energy consumption among islands.

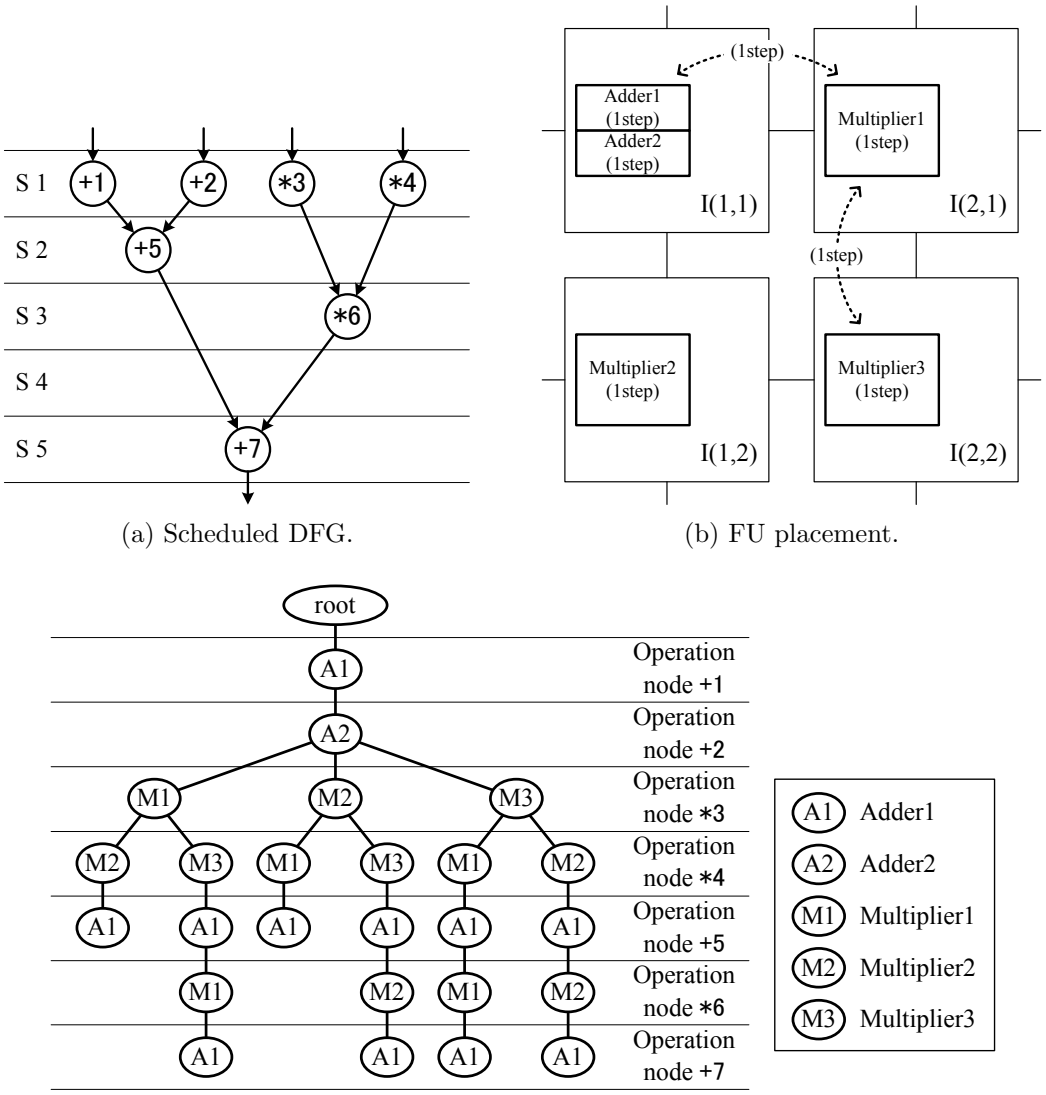
In Step(1.1), based on the scheduling solution, we arrange operation nodes included in the DFG $G = (V, E)$ in the order of their assigned steps. In the following discussions, $n_k \in V$ denotes a k-th operation node in G.

In Step(1.2), we generate *Binding tree*. By using *Binding tree*, we search all the binding solutions which can be realized for the current FU placement and scheduling. In *Binding tree*, each tree node v located at the level of k ($1 \leq k \leq |V|$) satisfies the conditions (1) to (5) below:

(1) Let Fu be a set of the FUs allocated in the current placement in RDR architecture. The tree node v has a *label* pointing to an FU $fu \in Fu$.

(2) FUs pointed to by v's sibling nodes are different from fu and allocated to different islands from fu.

(3) fu can execute the operation node n_k.



(a) Scheduled DFG.

(b) FU placement.

(c) *Binding tree.*

Figure 3.4: An example of *Binding tree.*

(4) When an operation node n_l $(1 \leq l \leq k-1)$ and n_k are scheduled in the same step, every ancestor tree node located at the level of l points to a different FU from fu.

(5) When we bind the operation node n_k to fu, no data transfer violation occurs for the current FU placement and scheduling. Note that the FUs bound to source operation nodes of n_k can be identified from v's ancestor tree nodes.

Algorithm 3.1 Generate *Binding tree* (Step(1.2)).

Require: arranged operation node list L

1: Generate *root* and insert *root* into the queue Q.
2: **while** $L \neq \emptyset$ **do**
3: Set $n \leftarrow$ the head of L and remove n from L.
4: **while** $Q \neq \emptyset$ **do**
5: Pick up a first tree node t from Q.
6: For each FU $fu \in Fu$, where Fu shows a set of the FUs allocated in RDR architecture, check whether n can be bound to fu.
7: **if** n can be bound to fu **then**
8: Generate a tree node t' having a label pointing to fu as a child node of t if all the FUs labeled by t's child nodes are allocated to islands different from fu, and insert t' into the queue Q'.
9: **end if**
10: **end while**
11: Move all the elements from Q' to Q.
12: **end while**

Ensure: *Binding tree*

Example 3.2. *When the scheduled DFG shown in Fig. 3.4a and the FU placement shown in Fig. 3.4b are given, we obtain Binding tree shown in Fig. 3.4c. As in Fig. 3.4b, we assume that one step is required to transfer data between the adjacent islands as well as to execute an operation by each FU.*

According to the condition (1), each tree node has a label which points to one of the FUs allocated in RDR architecture. According to the condition (2), sibling tree nodes at every level point to different FUs which are allocated to different islands. According to the condition (3), each addition node uses an adder as well as multiplication node uses a multiplier. According to the condition (4), the operation nodes '+1' and '+2' in the DFG are bound to different FUs since both of them are scheduled in the step 'S1'. According to the condition (5), we bind the operation node '∗6' to the FU not to cause any transfer violation since it has to be done after the operation nodes '∗3' and '∗4'.

By using Binding tree shown in Fig. 3.4c, we can find that the four different binding solutions can be realized for the FU placement and scheduling. □

We propose **Algorithm 3.1** to realize Step(1.2). In the 7th step of Algorithm 3.1, we can bind an operation node n to an FU fu when n and fu satisfy the following conditions:

- fu can execute n.
- fu is used by none of the operation nodes scheduled in the same step as n.

- When we bind n to fu, no transfer violation occurs.
- Let n_d be one of the destination operation nodes of n. For each of n_d, there is an FU to which n_d can be bound without causing any transfer violation when we bind n to fu.

In Step(1.3), we traverse *Binding tree* and search an optimal binding solution where the energy consumption is maximally balanced among islands. The main problem here is how to determine how much the energy consumption is balanced among islands. We propose $cost_1$ calculated by Eq. (3.6) to show how much the energy consumption is balanced among islands:

$$cost_1 = \sum_{i \in I} gap_i, \tag{3.6}$$

where I shows a set of islands which the target RDR architecture has. gap_i here shows the difference between ideal energy consumption per island and actual energy consumption per island. We design $cost_1$ by summing up the value of gap_i. In the following, we discuss how to design gap_i.

gap_i is based on the value *goal* which shows the ideal value of energy consumption per island. We find *goal* by using the number of operation nodes and the number of islands in RDR architecture, but *goal* must have the value of weighted average depending on operations since two FUs executing different operations can have different energy consumption. Let β_f be the weight of an operation f[2] such as an addition or a multiplication. We propose the equation to calculate *goal* as follows:

$$goal = \frac{sum}{|I|} \tag{3.7}$$

$$sum = \sum_{f \in F} |V_f| \times \beta_f \tag{3.8}$$

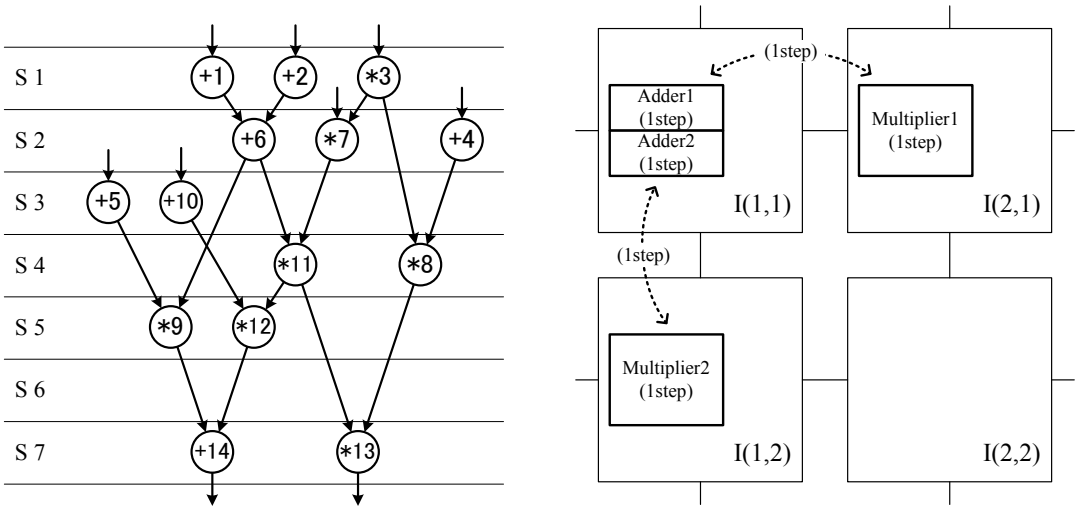
where F shows a set of operations appeared in the given DFG G, and V_f shows a subset of operation nodes in G which execute an operation f. Then gap_i is calculated by the following equation:

$$gap_i = |sum_i - goal| \tag{3.9}$$

$$sum_i = \sum_{f \in F} f(i) \times \beta_f \tag{3.10}$$

where $f(i)$ shows how many times an operation f is executed at an island i. sum_i roughly estimates the energy consumption in the island i and *cost* should be the

[2]In our experiments, we set $\beta_m = 2$ for a multiplication and $\beta_a = 1$ for an addition.



(a) Scheduled DFG. (b) Initial FU placement.

Figure 3.5: An input example of thermal-aware HLS for RDR architecture.

minimum when the estimated energy consumption in all the islands gets close to *goal*. Therefore, we expect that the energy consumption among islands can be maximally balanced when *cost* becomes the minimum value.

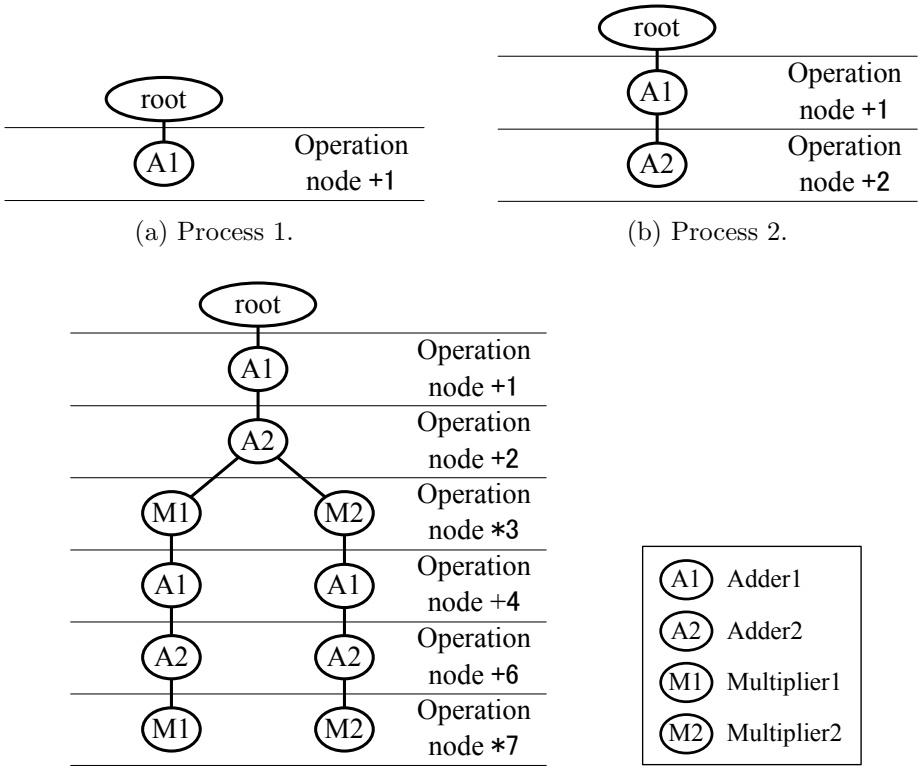
Example 3.3. *Assume that the scheduled DFG and the FU placement shown in Fig. 3.5 are given. We assume that one step is required to transfer data between the adjacent islands as well as to execute an operation by each FU.*

In Step(1.1), we arrange the operation nodes in their order of assigned step based on the scheduled DFG. In other words, we arrange the operation nodes in the order of

$$\{+1, +2, *3, +4, +6, *7, +5, +10, *8, *11, *9, *12, *13, +14\}$$

and store them into the list L.

*In Step(1.2), we generate Binding tree based on L. Fig. 3.6 shows the process of generating Binding tree. We can bind the operation node '+1' to 'A1' or 'A2' since it is an addition node, but their FUs are allocated to the same island $I(1,1)$ and hence we insert the tree node as shown in Fig. 3.6a. Though the operation node '+2' is an addition node, we cannot bind it to the same adder as the operation node '+1' since they are scheduled in the same step 'S1'. Then we insert the tree node as shown in Fig. 3.6b. In the same way, we can insert the tree nodes for the operation nodes '*3', '+4', and '+6'. The operation node '*7' is a multiplication node and we have to consider data transfer from the operation node '*3'. We cannot bind the operation node '*7' to 'M2' if we bind the operation node '*3' to 'M1'. Therefore, we insert the tree nodes as shown in Fig. 3.6c.*



(a) Process 1.

(b) Process 2.

(c) Process 3.

Figure 3.6: Process of generating *Binding tree*.

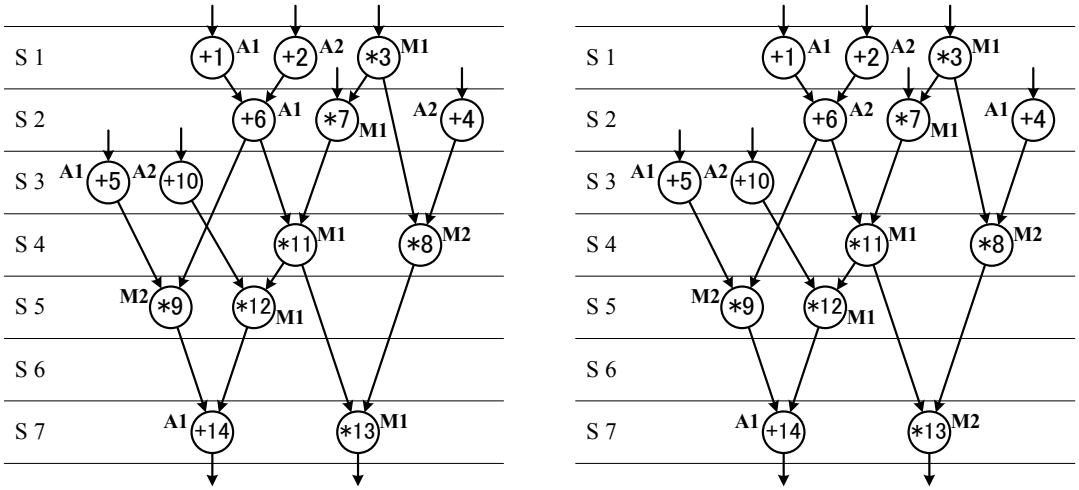
In Step(1.3), we traverse Binding tree and evaluate cost to search a binding solution where the energy consumption is maximally balanced among islands. In this example, we set $\beta_a = 1$ *and* $\beta_m = 2$ *for an addition and a multiplication, respectively. By using the value of* $|V_a| = 7$, $|V_m| = 7$, *and* $|I| = 4$ *obtained from Fig. 3.5, we calculate sum and goal as follows:*

$$sum = 7 \times 1 + 7 \times 2 = 21, \quad goal = 21/4 = 5.25$$

Let $i_1 = I(1,1)$, $i_2 = I(2,1)$, $i_3 = I(1,2)$, *and* $i_4 = I(2,2)$ *and assume that we have a binding solution of Fig. 3.7a. In this case, additions are executed seven times in the island* i_1 *and we have* $i_1(a) = 7$. *We also have* $i_2(m) = 5$ *and* $i_3(m) = 2$ *in the same way. Then we calculate* gap_i *as follows:*

	$i_1 = I(1,1)$	$i_2 = I(2,1)$	$i_3 = I(1,2)$	$i_4 = I(2,2)$
sum_i	7	10	4	0
gap_i	1.75	4.75	1.25	5.25

Based on the value calculated above, we obtain $cost_1 = 13$ *from Eq. (3.6).*



(a) Solution 1. (b) Solution 2.

Figure 3.7: Two FU binding solutions.

If we have a binding solution of Fig. 3.7b, we have $i_1(a) = 7$, $i_2(m) = 4$, $i_3(m) = 3$ *and then obtain cost* $= 10.5$ *from Eq. (3.6). This result shows that the binding solution shown in Fig. 3.7b balances the energy consumption more among islands compared with the solution shown in Fig. 3.7a.*

We have used HotSpot-5.0 [1] to simulate the peak temperature difference between the surface and inside of the chip, and obtained 7.81°C and 7.05°C for the FU binding solutions in Fig. 3.7a and Fig. 3.7b, respectively. This result demonstrates that a binding solution with small $cost_1$ *gives low peak temperature.* □

Step1 may have a serious problem that the computational time exponentially increases as the number of operation nodes increases. According to our experimental results, we cannot directly apply our thermal-aware FU binding algorithm to application benchmarks having more than 30 operation nodes.

To solve this problem, we propose a partition algorithm for *Binding tree*. When the application has a large number of operation nodes, Step(1.2) and Step(1.3) will be done several times for partitioned *Binding tree*.

Fig. 3.8 shows the flow of our partition algorithm when partitioning the operation node list L into λ sub-lists. A set of the Step(1.2) and Step(1.3) is called *stage* and we assign each operation node in L to one of the stages. In other words, we assign all the operation nodes stored in the list L to several sub-lists L_k and the stage k is applied to L_k where L_k has around $|V|/\lambda$ operation nodes. In our algorithm, we just store each node in L into L_k from the first node to the last node in L. Note that we partition L into L_k so that the operation nodes assigned to the

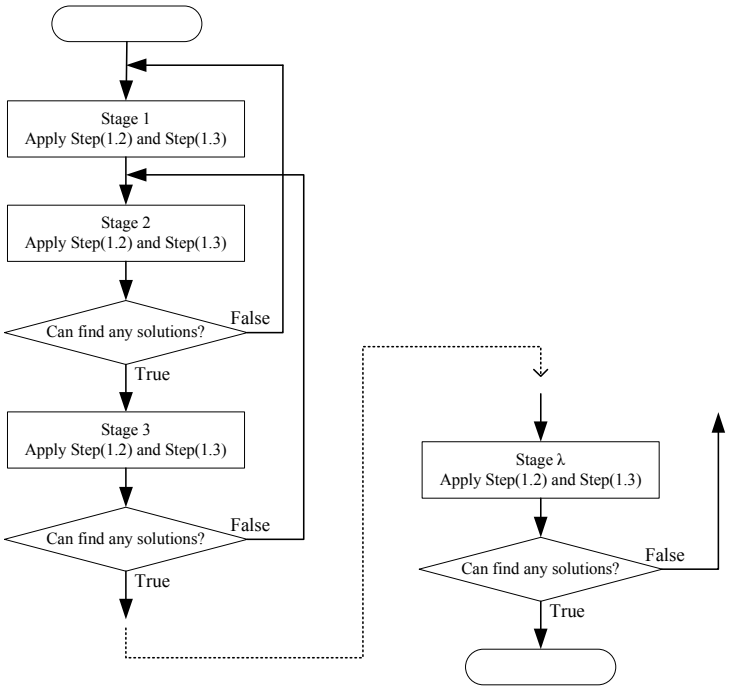


Figure 3.8: The partition algorithm.

same step is included into the same sub-list.

In each stage, we search a binding solution where the energy consumption is maximally balanced among islands. After that we move to the next stage. If we have no solutions, we go back to the previous stage.

3.3.4 Additional FU allocation (Step2)

Step2 tries all the pattern of adding FUs to vacant spaces in RDR architecture so that the energy consumption is maximally balanced among islands and hence the highest one is minimized. In Step2, we bind registers to islands based on the scheduling/FU binding solution obtained in Step1, which enables more accurate evaluation of energy consumption.

In Step(2.1), we first bind registers to each island based on the current FU placement and its FU binding solution obtained just before. The register binding algorithm is the same used in MCAS [7]. Binding registers enables us to evaluate the intra-island energy consumption more accurately compared to Eq. (3.10) in Step1 and then it is estimated for each island i by:

$$energy_i = \sum_{fu \in Fu(i)} Exe(fu) \times E_{fu} + CS_{req} \times \sum_{reg \in Reg(i)} E_{reg} \qquad (3.11)$$

Table 3.1: Capacity cost, delay, energy, and leakage power of modules [3].

	Capacity cost	Delay (ns)	Energy (pJ)	Leakage power (mW)
Adder	1	1.32	0.064	0.0032
Multiplier	2	2.70	0.788	0.0165
Register	–	0.11	0.743	0.0107
Multiplexer	–	0.04	0.144	0.0017

where $Fu(i)$ and $Reg(i)$ show a set of the FUs and the registers allocated to the island i, respectively. $Exe(fu)$ and CS_{req} are obtaind from the scheduling/FU binding solution, which show the number of operations bound to the FU fu and the number of required steps when executing the DFG G, respectively. E_{fu} and E_{reg} show the per unit energy consumption of the FU fu and the register reg, respectively. Note that they include the energy consumed by multiplexers connected to each of them.

By utilizing Eq. (3.11) to estimate the energy consumed in each island, we can improve the accuracy of the estimation and hence identify the island having the highest energy consumption and then heated much. Therefore, by minimizing the maximum energy consumption among islands when executing the DFG G, we can expect to achieve minimization of the peak temperature. Then we propose $cost_2$ calculated by Eq. (3.12) to show how much the maximum energy consumption is minimized among islands, and attempt to minimize its value:

$$cost_2 = \max_{i \in I} energy_i \tag{3.12}$$

In Step(2.2), based on $cost_2$ calculated by Eq. (3.12), we determine whether the current FU placement is the best so far. If $cost_2$ gives the minimum value at this time, we save this solution.

In Step(2.3), we try all the feasible pattern of allocating a new FU into a vacant space. After allocating a new FU, we go back to the beginning of Step1. This step is tried in a one-by-one manner.

3.4 Experimental Results

We have implemented our algorithm in C++. The algorithm has been run on AMD Opteron 2360 SE (2.5 GHz, Quad core) with 16 GB memory and applied to DCT (48 operation nodes), EWF3 (102 operation nodes), and FIR (75 operation nodes). We have used RDR architectures with island capacity of $C = 2$. Table 3.1

Table 3.2: Experimental results.

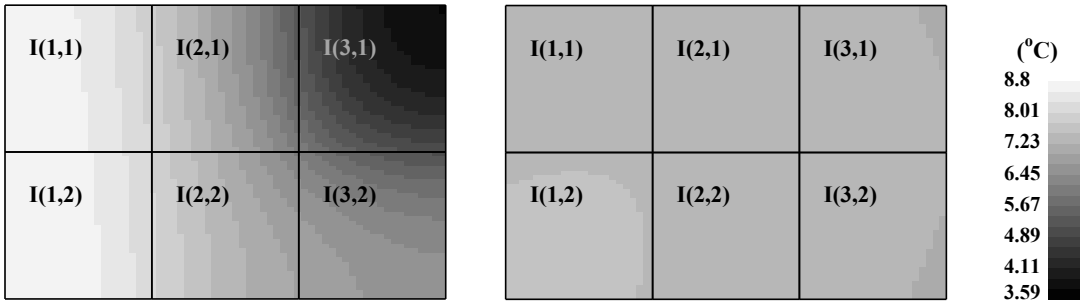
App. #Islands	T_{clk} (ns)	Algorithm	#Steps	#FUs (Add, Mul)	Max area (μm^2)	CPU time (sec)	Max temp. diff. (°C)	Reducing rate (%)
DCT	3.0	[7]	12	(3, 2)	7,278	99.48	11.16	–
2×2		Ours	12	(4, 2)	6,893	99.64	11.06	0.90
DCT	3.5	[7]	10	(4, 3)	6,639	124.25	8.80	–
3×2		Ours	10	(4, 4)	6,517	149.80	7.53	14.4
EWF3	3.0	[7]	55	(2, 2)	10,213	81.53	8.51	–
2×2		Ours	55	(4, 2)	7,988	107.67	7.19	15.5
FIR	3.0	[7]	24	(3, 2)	6,816	101.12	10.50	–
2×2		Ours	24	(3, 2)	6,816	100.59	10.49	0.10
FIR	3.5	[7]	20	(3, 3)	6,667	125.32	8.07	–
3×2		Ours	20	(3, 4)	6,653	151.22	7.47	7.43

shows the capacity cost, delay, energy, and leakage power of FUs, a register, and a multiplexer used in this experiment, which are all assumed to have 16 bit width under the CMOS 90 nm technology. A controller has been synthesized for each island by Synopsys Desgin Compiler after applying our algorithm. We have set in Eq. (3.1) the interconnect delay coefficient $C_d = 1$ ns as in [24].

We have compared our experimental results with those obtained by MCAS [7]. The partition algorithm described in Section 3.3.3 is utilized in our algorithm, and set the parameter $\lambda = 3$, 7 and 5 when applying it to DCT, EWF3 and FIR, respectively.

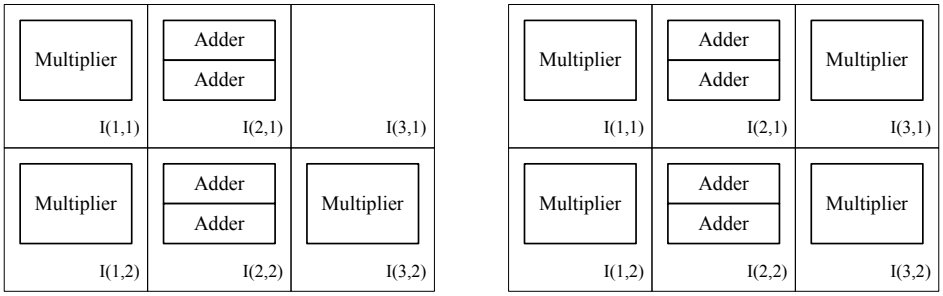
Table 3.2 shows the experimental results. The 1st column shows the given DFG and specification of RDR architecture (the number of islands), and the 2nd column shows the given clock period constraint. As shown in the 4th column, ours and [7] have the same number of required steps in every application. This means that no performance degradation occurs by applying our algorithm. The 5th and the 6th columns show the number of FUs and the maximum effective island area, respectively. The maximum effective island area means the module area of the island which gives the highest module occupancy of all the islands. As the 6th column indicates, our algorithm reduces the maximum effective island area and hence causes no area overhead. This is because ours reduces the number of registers and multiplexers per island by balancing the operations among islands. The 7th column shows CPU time to synthesize each circuit.

Then we have simulated the peak temperature difference between the surface and inside of the chip using HotSpot-5.0 [1]. Its inputs are the chip floorplan and the energy consumption trace and, after that, it outputs steady-state temperatures inside the chip. In our experiments, we have input island floorplans in RDR architecture and energy consumption traces in each island. The experimental re-



(a) MCAS [7]. (b) Ours.

Figure 3.9: Temperature distribution when applying each algorithm to DCT (the number of islands: 3×2).



(a) MCAS [7]. (b) Ours.

Figure 3.10: The results of FU allocation when applying each algorithm to DCT (the number of islands: 3×2).

sults in the 8th and the 9th columns of Table 3.2 demonstrate that our algorithm successfully reduces the peak temperature by up to 15.5%.

The temperature distribution inside a chip when applying MCAS [7] to the application DCT is shown in Fig. 3.9a. In this result, four adders and three multipliers have been allocated to six islands as in Fig. 3.10a. Since decreasing the number of FUs results in increasing the number of required steps and increasing it does not change the number of required steps, FUs have been allocated appropriately by applying MCAS with the objective of minimizing the number of required steps. On the other hand, as in Fig. 3.10b, one multiplier has been newly allocated to the island $I(3, 2)$ by applying our algorithm. As a result, we have obtained the temperature distribution shown in Fig. 3.9b. Note that temperature in Fig. 3.9a and Fig. 3.9b shows the temperature difference between the surface and inside of

the chip. By compared these figures, we find that our algorithm balances chip temperature and hence removes a hot-spot shown in the lower left-hand corner of Fig. 3.9a.

3.5 Conclusion

In this chapter, we have proposed a thermal-aware HLS algorithm for RDR architecture through energy-balanced FU binding and additional FU allocation. Our algorithm gives one of the solutions to tackle increasing heat problem in an IC chip by incorporating a thermal-aware synthesis flow into HLS for RDR architecture. The experimental results have shown that our algorithm can reduce the peak temperature by up to 15.5% compared with the conventional approach.

Chapter 4

An Overhead Constraint-based Partially Redundant Fault-secure High-level Synthesis Algorithm[1]

4.1 Introduction

In this chapter, an overhead constraint-based partially redundant fault-secure HLS algorithm for DR architectures is proposed. In our HLS algorithm, RDR architecture which is one of the DR architectures is utilized. As illustrated in Fig. 2.1, each island in RDR architecture keeps a fixed area even when few circuit modules are assigned, and hence some of the islands may have vacant spaces. By utilizing the vacant spaces for re-computation in fault-secure design, we expect that reliability can be improved without any area overhead. This algorithm attempts to maximize reliability within a set of latency and area constraints. In order to quantitatively evaluate reliability, we formulate the error output probability when a soft error occurs. Particularly in our HLS algorithm proposed in this chapter, the latency and area constraints are set not to cause any overhead. These constraints are set in the design flow based on the synthesized results for normal-computation only. Our proposed algorithm duplicates all the operations and then removes some of them within a set of constraints by a greedy search method. Experimental results demonstrate that our proposed algorithm improves reliability by up to 24% without any performance/area overhead compared to a conventional HLS algorithm for RDR architecture.

[1]Technical contents in this chapter have been presented in the publications ⟨7⟩, ⟨15⟩, and ⟨16⟩.

4.2 Problem Formulation

In HLS, behavioral description used as input is transformed into graphical representation like a control-data flow graph (CDFG) or a data-flow graph (DFG). Hereafter we use a DFG as an input for simplicity. Note that the discussion below can be easily extended to a CDFG.

A normal-computational 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 show the data dependencies between two operation nodes. In fully redundant fault-secure HLS, its re-computational DFG $G_R = (V_R, E_R)$ is generated by duplicating the normal-computational DFG $G = (V, E)$. Given a scheduling/binding result of the normal-computational DFG G, its re-computational DFG G_R is scheduled/bound so that the *fault-secure conditions* (1) to (6) below are satisfied:

(1) The scheduling/binding result of the normal-computational DFG G is not changed.

(2) Let $n' \in V_R$ be one of the re-computational operation nodes, and $n \in V$ be the normal-computational operation node corresponding to n'. They cannot share an identical functional unit (FU).

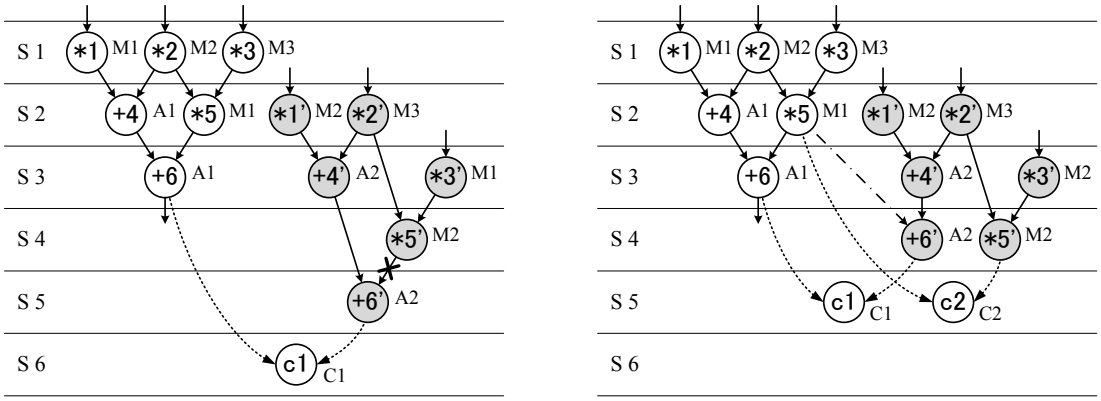
(3) All the outputs of the normal-computational DFG G and the re-computational DFG G_R must be compared.

(4) An edge $e' \in E_R$ can be broken.

(5) Let $e' = (m', n') \in E_R$ be the input edge of n' in the re-computational DFG G_R, where m' is one of the parent nodes of n'. Let m be the normal-computational operation node corresponding to m'. If e' is broken, n' uses the output of the normal-computational operation node m instead of m'.

(6) The outputs of m and m' must be compared if all the output edges of m' are broken.

Example 4.1. *Fig. 4.1a shows an example of fully redundant fault-secure scheduling/binding. We assume that the three multipliers 'M1', 'M2', 'M3', the two adders 'A1', 'A2', and the two comparators 'C1', 'C2' can be used in this example. The re-computational operation nodes are illustrated by the shaded nodes. According to the fault-secure condition (1), the scheduling/binding result of the normal-computational DFG is not changed. According to the fault-secure condition (2), the normal-computational operation '$*1$' is bound to the multiplier 'M1' but its re-computational operation '$*1'$' is*



(a) Before edge breaking.

(b) After edge breaking.

Figure 4.1: An example of fully redundant fault-secure scheduling/binding.

bound to 'M2'. According to the fault-secure condition (3), the outputs of the normal-computational DFG and the re-computational DFG are compared by using the comparator 'C1'.

Fig. 4.1b shows a scheduling/binding result of the re-computational DFG when the edge marked as '×' in Fig. 4.1a is broken. By breaking the edge, the re-computational operation '+6′' can be scheduled to the step 'S4'. According to the fault-secure conditions (4) and (5), the output of the normal-computational operation '∗5' is used for the input of the re-computational operation '+6′'. According to the fault-secure condition (6), the outputs of the normal-computational operation '∗5' and the re-computational operation '∗5′' are compared by using the comparator 'C2'. As shown in this example, breaking edges in a re-computational DFG can reduce the number of steps required. □

In partially redundant fault-secure HLS, partly removing operation nodes in the re-computational DFG G_R is acceptable at the risk of reliability degradation. Let $n'_1 \in V_R$ be a re-computational operation node, and let $n_1 \in V$ be the normal-computational operation node corresponding to n'_1. When n'_1 is removed, all the input/output edges connected to n'_1 are broken and the fault which occurs during executing n_1 becomes less able to be detected. In this case, the following operations are performed to prevent the secondary effect on fault detectability:

- Let $n'_2 \in V_R$ be the re-computational operation node whose input edge is broken when removing n'_1. n'_2 uses the output of n_1 instead of n'_1.
- Let $n'_3 \in V_R$ be the re-computational operation node whose output is not connected to any node when removing n'_1. The outputs of n'_3 and $n_3 \in V$ which is the normal-computational operation node corresponding to n'_3 are compared.

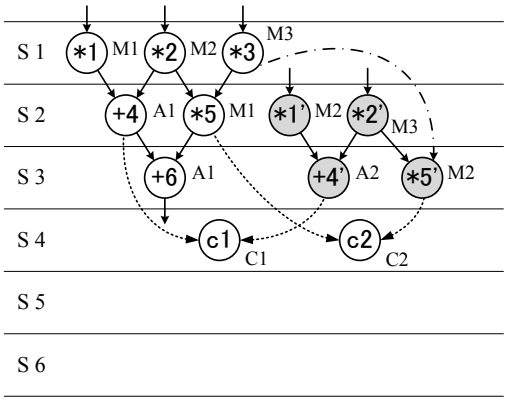


Figure 4.2: An example of partially redundant fault-secure scheduling/binding.

Example 4.2. *Fig. 4.2 shows an example of partially redundant fault-secure scheduling/binding which removes the re-computational operations '$*3'$' and '$+6'$' from the result of fully redundant fault-secure scheduling/binding shown in Fig. 4.1. As the operation '$*3'$' is removed, one of the input edges of '$*5'$' is broken and hence the output of '$*3$' is used for '$*5'$'. As the operation '$+6'$' is removed, both '$+4'$' and '$*5'$' have no output edges and hence their outputs are compared to '$+4$' and '$*5$', respectively. As shown in this example, removing some operation nodes in a re-computational DFG can reduce the number of steps required.* □

In RDR architecture, an entire circuit is assigned to $N \times M$ array of islands. Let $I(x, y)$ be an island located at x-th column and y-th row of the array, where $1 \leq x \leq N$ and $1 \leq y \leq M$. All the islands assume to be square and have the same size. Let W and H be the width and the height of islands, respectively. The equation $W = H$ is always true and the size of islands $W \times H$ is calculated as

$$W \times H = \frac{A_{max}}{N \times M}, \tag{4.1}$$

where A_{max} denotes the given total area constraint.

An RTL circuit is composed of FUs, registers, multiplexers, and controllers. Let $Fu(i)$, $Reg(i)$, and $Mux(i)$ be the set of FUs, registers, and multiplexers which are allocated to an island i, respectively. Any island i in RDR architecture satisfies

$$W \times H \geq \sum_{fu \in Fu(i)} A_{fu} + |Reg(i)| \times A_{reg} + |Mux(i)| \times A_{mux} + A_{cont}(i), \tag{4.2}$$

where A_{fu}, A_{reg}, and A_{mux} show the areas of an FU fu, a register, and a multiplexer, respectively. $A_{cont}(i)$ shows the area of a controller synthesized for controlling the datapath on an island i.

Let i_1 and i_2 be two islands $I(x_1, y_1)$ and $I(x_2, y_2)$, respectively. An interconnect 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 it is given by

$$D_c(i_1, i_2) = C_d \times (|x_1 - x_2| + |y_1 - y_2|)^2 \tag{4.3}$$

where C_d shows the constant interconnect delay coefficient. Let fu_1 be one of the FUs allocated to the island i_1, which has a delay of d_{fu_1}. Assume that the output of fu_1 is continuously used by an FU allocated to the island i_2. Let T_{clk} be the given clock period constraint. For the clock period constraint T_{clk} and the FU fu_1, the computation by fu_1 can be done in $s_{fu_1} = \left\lceil \frac{d_{fu_1}}{T_{clk}} \right\rceil$ steps. When

$$D_c(i_1, i_2) + d_{fu_1} \leq s_{fu_1} \times T_{clk} \tag{4.4}$$

holds, the computation by fu_1 and its output data transfer to a register allocated to the island i_2 are done in s_{fu_1} steps. On the other hand, when

$$D_c(i_1, i_2) + d_{fu_1} > s_{fu_1} \times T_{clk} \tag{4.5}$$

holds, the computation by fu_1 is done in s_{fu_1} steps and its output is stored into a register allocated to the island i_1 at the s_{fu_1}-th step. After that the stored data is transferred to a register allocated to the island i_2 using $\left\lceil \frac{D_c(i_i, i_2)}{T_{clk}} \right\rceil$ steps. Therefore, when two continuous operations are serially executed by fu_1 and fu_2, the step required only for data transfer between the two operations is summarized as follows:

$$dt(fu_1, fu_2) = \begin{cases} 0 & (D_c(i_1, i_2) + d_{fu_1} \leq s_{fu_1} \times T_{clk}) \\ \left\lceil \frac{D_c(i_i, i_2)}{T_{clk}} \right\rceil & (D_c(i_1, i_2) + d_{fu_1} > s_{fu_1} \times T_{clk}) \end{cases} \tag{4.6}$$

We focus on a fault-secure design for logic cells and assume that all the memory components have fault-tolerant capabilities such as ECC [6] based on the discussions in Section 2.4. In this chapter, the fault model is set as follows:

- In a component (an FU, a register, a multiplexer, or a controller), the probability a soft error occurs is proportional to its area.
- A soft error which occurs in a component has no effect on all other components.
- When a soft error occurs at a memory component such as a register or a part of a controller, no faults are caused.

- When a soft error occurs at a logic component such as an FU, a multiplexer, or a part of a controller during it operates, a fault is always caused.
- When a soft error occurs at a logic component during it does not operate, no faults are caused.
- A fault always upsets the circuit function during only one step.

When a fault is caused by a soft error and is not detected by operation duplication, an erroneous result is output from the circuit.

Let S_{max} be the given total step constraint in partially redundant fault-secure HLS. Assume that a single radiation-induced soft error occurs on RDR architecture in a step k, where $1 \leq k \leq S_{max}$. We define P_E as the error output probability when a soft error occurs. In partially redundant fault-secure HLS, minimizing the value of P_E is assumed to equal maximizing reliability. The following two paragraphs discuss how to calculate P_E:

Circuit behavior in RDR architecture is classified into (a) execution of an operation and (b) data transfer between two registers. (a) reads a data from a register, executes an operation by an FU, and writes its execution result to a register, while (b) reads a data from a register and writes it to another register. Since the input port of every FU or register may have one or more multiplexers, a soft error which occurs not only at each FU but also at each multiplexer must be considered. Let $v \in V$ be a normal-computational operation node not to be duplicated. With respect to v, an erroneous result is output under any of the situations below:

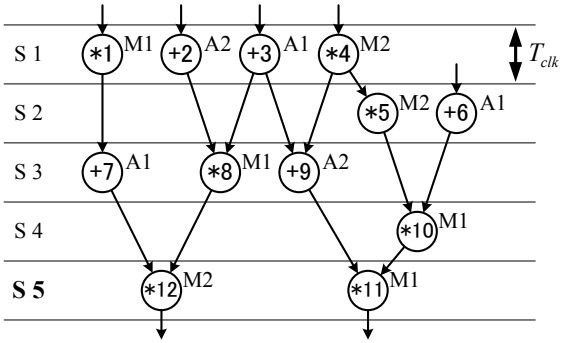
- During the execution of v, a soft error occurs at an FU or a multiplexer related to its behavior.
- During the inter-register transfer executed at one of the input edges of v, a soft error occurs at a multiplexer related to its behavior.

When calculating P_E, first we calculate in each step k the area of error sensitive reagions where a fault is caused by a soft error and cannot be detected. Next we sum up its area from 1-st to S_{max}-th steps, and obtain $A_{ES}(1 : S_{max})$ which shows the total area of error sensitive reagions when executing a whole DFG:

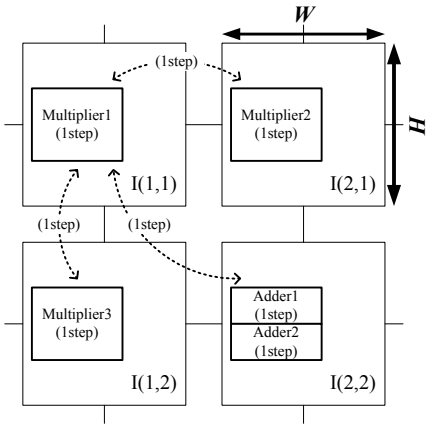
$$A_{ES}(1 : S_{max}) = \sum_{k=1}^{S_{max}} A_{ES}(k). \tag{4.7}$$

Based on Eq. (4.7), P_E is calculated as follows:

$$P_E = \frac{A_{ES}(1 : S_{max})}{A_{max} \times S_{max}}. \tag{4.8}$$



(a) Scheduled/bound normal-computational DFG.



(b) RDR architecture and its initial FU placement.

Figure 4.3: An input example of partially redundant fault-secure HLS for RDR architecture.

Now we define a partially redundant fault-secure HLS problem for RDR architecture as follows:

Definition 4.1. *For a given normal-computational DFG $G = (V, E)$ which has already been scheduled/bound, specification of RDR architecture (the number of islands $N \times M$), FU placement on this RDR architecture, library of FUs, clock period constraint T_{clk}, total area constraint A_{max}, and total step constraint S_{max}, our partially redundant fault-secure HLS problem for this RDR architecture is to construct and schedule/bind its re-computational DFG $G_R = (V_R, E_R)$ and to allocate FUs to each island with the objective of minimizing the error output probability P_E unless its initial FU placement is changed.* □

4.3 The Algorithm

In this section, an overhead constraint-based partially redundant fault-secure HLS algorithm for RDR architecture is proposed. In our HLS algorithm, the clock period constraint T_{clk} and the total area constraint A_{max} are given identically with those input to MCAS [7] which is a conventional HLS algorithm with no fault-secure consideration. The number of required steps, which is obtained in applying MCAS, is set as the total step constraint S_{max} in our HLS algorithm. The set of constraints means that the proposed algorithm realizes a fault-secure design not to cause any performance/area overhead.

Example 4.3. *Fig. 4.3 shows an input example of partially redundant fault-secure HLS for RDR architecture. The scheduled/bound normal-computational DFG and the FU placement in RDR architecture are obtained by applying MCAS, which are shown in Figs. 4.3a and 4.3b, respectively. The clock period constraint T_{clk} and the total area constraint A_{max} are given identically with those input to MCAS. In this example, the island size $W \times H = \frac{A_{max}}{2 \times 2}$ constrains the area of modules which can be allocated to each island. Since the normal-computational DFG is executed in 5 steps, the total step constraint S_{max} is set to be 5.* □

In the subsections below, we first discuss the strategy and then propose our algorithm under the above constraint set.

4.3.1 Strategy

To solve the problem defined in Section 4.2, we need to determine which operations to duplicate. Considering all combinations of duplicated operations is too computationally expensive because the number of combinations amounts $2^{|V|}$ where V shows the operation node set included in the given DFG $G = (V, E)$. When determining a set of operation nodes to be duplicated and scheduling/binding each of them, we can have two options as follows:

(a) We first determine which operation nodes to duplicate, and then schedule/bind their operation nodes under the given constraint set. In other words, the re-computational DFG is constructed by gradually *increasing* the number of operation nodes which are duplicated.

(b) We first schedule/bind the re-computational DFG generated by fully duplicating the normal-computational DFG, and then under the given constraint set, determine which operation nodes to duplicate. In other words, the re-computational DFG is constructed by gradually *reducing* the number of operation nodes which are duplicated.

When adopting the strategy (a), the space of solution in scheduling/binding a or some re-computational operation node(s) is too large to sufficiently increase the number of operation nodes which are duplicated. On the other hand, when adopting the strategy (b), the space of solution can be reduced because we only determine which operation nodes to remove from the re-computational DFG. For these reasons, we adopt the strategy (b) as our HLS algorithm.

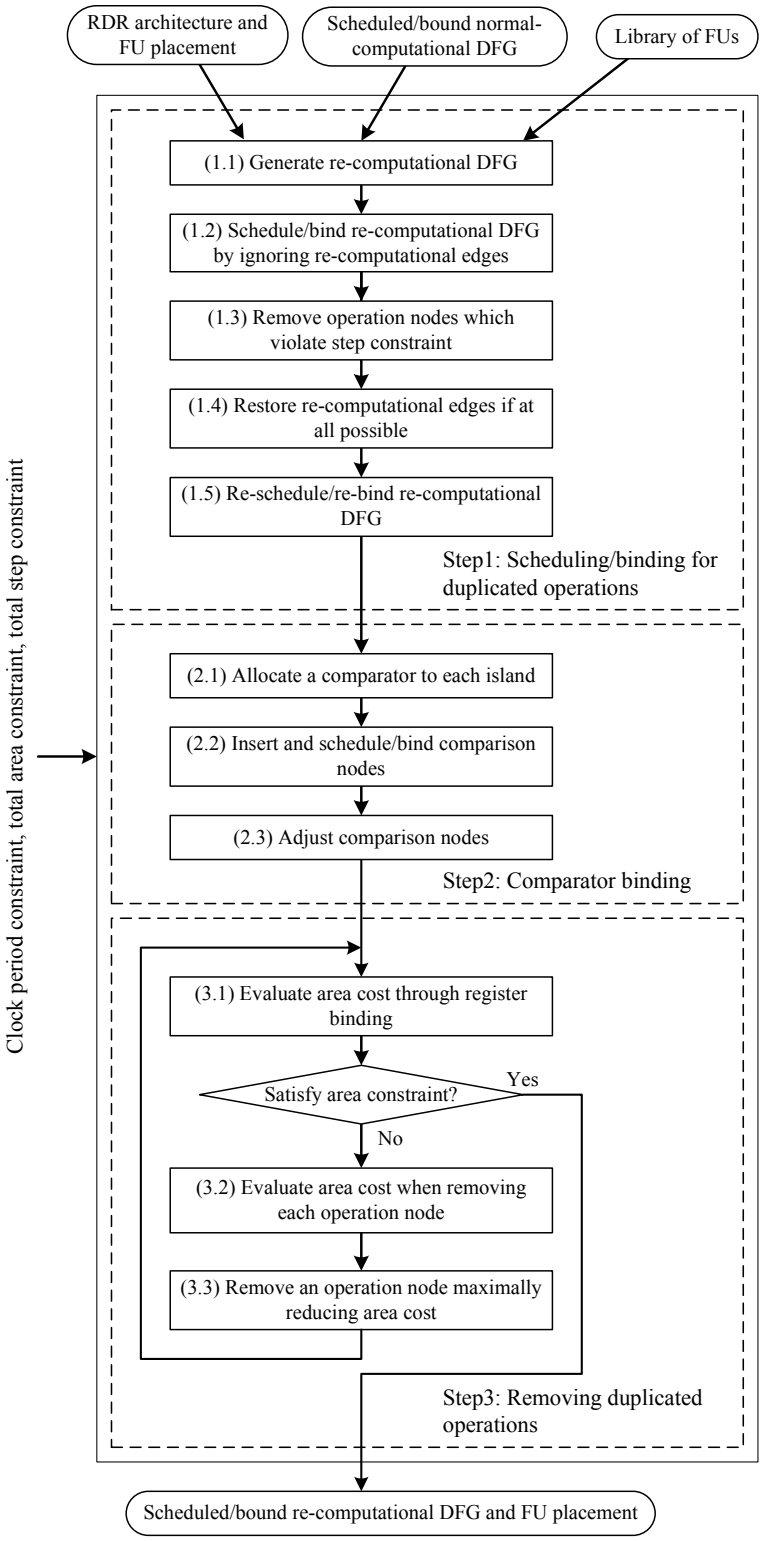


Figure 4.4: The proposed partially redundant fault-secure HLS algorithm.

4.3.2 Overall synthesis flow

Fig. 4.4 shows our proposed algorithm. The algorithm consists of three steps: scheduling/binding for duplicated operations (Step1), comparator binding (Step2), and removing duplicated operations (Step3).

In Step(1.1), a re-computational DFG is generated by fully duplicating the given normal-computational DFG. In Step(1.2), each re-computational operation node is scheduled/bound with all the input edges broken, which can schedule each node as soon as possible. In Step(1.3), the re-computational operation nodes which violate the given total step constraint are removed from the re-computational DFG. In Step(1.4) and Step(1.5), the re-computational DFG is re-scheduled/re-bound with the objective of minimizing the number of comparison nodes inserted in the later step.

In Step(2.1), a comparator for fault detection is allocated to each island in RDR architecture. In Step(2.2), according to the fault-secure conditions shown in Section 4.2, a comparison node is inserted to each of the re-computational nodes having no output edges and then is scheduled/bound. In Step(2.3), the comparison nodes which violate the given total step constraint are adjusted.

In Step(3.1), through register binding, the area of modules allocated to each island is evaluated. When there are any islands which violate the area constraint calculated by Eq. (4.1), a re-computational operation node is removed in Step(3.2) and Step(3.3), and then Step(3.1) is tried again. Step3 is iterated until all the islands in RDR architecture satisfy the area constraint.

4.3.3 Scheduling/binding for duplicated operations (Step1)

Under the clock period constraint T_{clk} and the total step constraint S_{max}, a re-computational DFG $G_R = (V_R, E_R)$ is generated and scheduled/bound. The first objective is to maximize the number of re-computational operation nodes, and the second objective is to minimize the number of comparison nodes inserted in the later step. In this step, the total area constraint A_{max} is not considered.

In Step(1.1), we generate a re-computational DFG $G_R = (V_R, E_R)$ by fully duplicating the normal-computational DFG $G = (V, E)$.

In Step(1.2), we schedule/bind the re-computational DFG G_R by completely ignoring re-computational edges. We propose **Algorithm 4.1** to realize Step(1.2). In the 1st step of Algorithm 4.1, a priority $pr(n')$ is set to each re-computational operation node $n' \in V_R$. The priority $pr(n')$ is set based on the critical path length from the end of G_R and is calculated by:

$$pr(n') = \min_{fu_s \in EFu(n')} \{cpl(n', fu_s)\} \tag{4.9}$$

Algorithm 4.1 Scheduling/binding re-computational DFG by ignoring re-computational edges (Step (1.2)).

1: Set a priority $pr(n')$ to each re-computational operation node $n' \in V_R$.
2: $cs \leftarrow 1$.
3: **while** scheduling/binding is not finished **do**
4: Let $PNode(n)$ be the set of parent operation nodes of an operation node n. Insert an operation node $n' \in V_R$, which all the normal-computational nodes corresponding to its parent nodes $m' \in PNode(n')$ have been scheduled before cs, into the ready list RL.
5: Arrange the operation nodes included in RL according to their priorities.
6: **while** $RL \neq \emptyset$ **do**
7: Set $n' \leftarrow$ the head of RL and remove n' from RL.
8: Insert an FU which can execute n' into the FU list FL.
9: **while** $FL \neq \emptyset$ **do**
10: Set $fu \leftarrow$ the head of FL and remove fu from FL.
11: **if** n' can be scheduled to the step cs and bound to the FU fu **then**
12: Perform scheduling/binding of n'.
13: **end if**
14: **end while**
15: **end while**
16: $cs \leftarrow cs + 1$.
17: **end while**

$$cpl(n', fu_s) = 1 + \max_{n'_c \in CNode(n')} \left\{ \min_{fu_d \in EFu(n'_c)} \{dt(fu_s, fu_d) + cpl(n'_c, fu_d)\} \right\} \tag{4.10}$$

where $EFu(n')$ and $CNode(n')$ show the set of FUs which can execute n' and the set of child operation nodes of n', respectively. $dt(fu_s, fu_d)$ is calculated by Eq. (4.6). In the 11th step of Algorithm 4.1, we need to ensure that all the re-computational operation nodes satisfy the fault-secure conditions (2) and (5). For this reason, the following requirements must be satisfied when a re-computational operation node n' is scheduled in a step cs and bound to an FU fu:

1. Let n and $PNode(n)$ be the normal-computational operation node corresponding to n' and the set of its parent operation nodes, respectively. When n' is scheduled to cs and bound to fu, the equation

$$\max_{m \in PNode(n)} \{cs(m) + dt(fu(m), fu)\} < cs$$

holds, where $cs(m)$ and $fu(m)$ show the step and the FU to which an operation node m has been scheduled/bound, respectively.

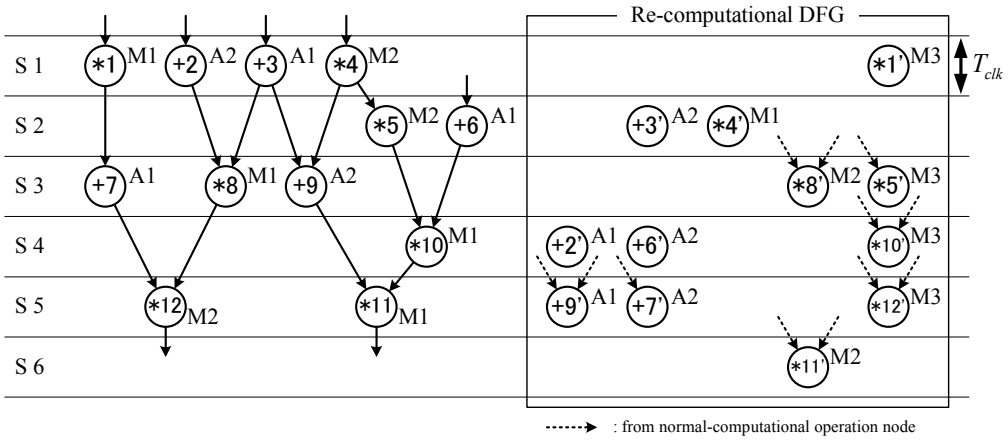


Figure 4.5: Scheduling/binding of re-computational DFG by ignoring re-computational edges (Step(1.2)).

2. Any other operation nodes have been bound to fu in the step cs.

3. $fu \neq fu(n)$, where $fu(n)$ shows the FU to which the normal-computational operation node n has been bound.

By applying Algorithm 4.1, the re-computational DFG G_R can be scheduled as soon as possible according to the fault-secure conditions.

Example 4.4. *For the input set as in Example 4.3, by applying Algorithm 4.1 after Step(1.1), we obtain the scheduled/bound re-computational DFG shown in Fig. 4.5. Every re-computational operation node satisfies the requirements (1)–(3) above, and is scheduled as soon as possible. By scheduling each re-computational operation node to earlier step, we expect to minimize the number of operation nodes removed in the later step.* □

In Step(1.3), considering the total step constraint S_{max}, we remove some operation nodes from the re-computational DFG G_R. A re-computational operation node $n' \in V_R$ is removed when

$$cs(n') > S_{max} - 1 \tag{4.11}$$

holds, where $cs(n')$ shows the step to which n' has been scheduled in Step(1.2). Note that the left side of Eq. (4.11) is not S_{max} but $S_{max} - 1$ because we need to schedule/bind some comparison nodes in the step S_{max} for fault detection.

Example 4.5. *In Step(1.3), for the result as in Example 4.4, we remove the re-computational operation nodes which have been scheduled to the steps later than* $S_{max} - 1 = 4$. *As shown in Fig. 4.6, four nodes are removed.* □

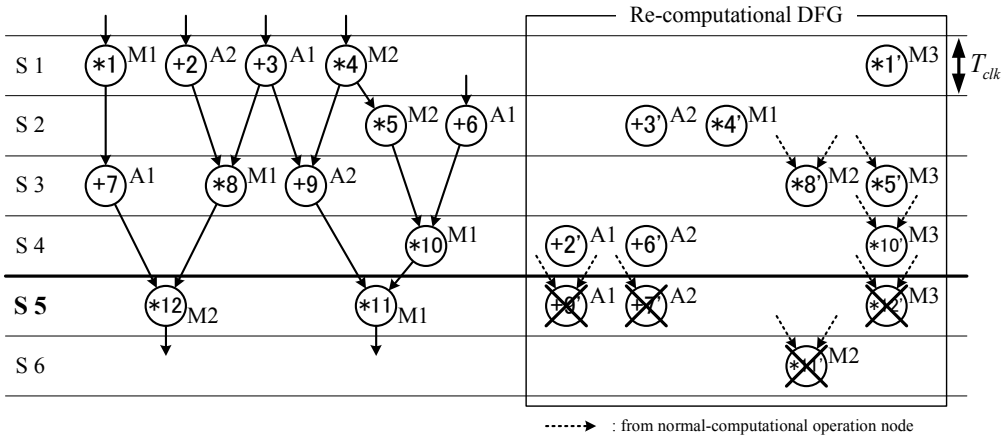


Figure 4.6: Removing operation nodes violating the step constraint (Step(1.3)).

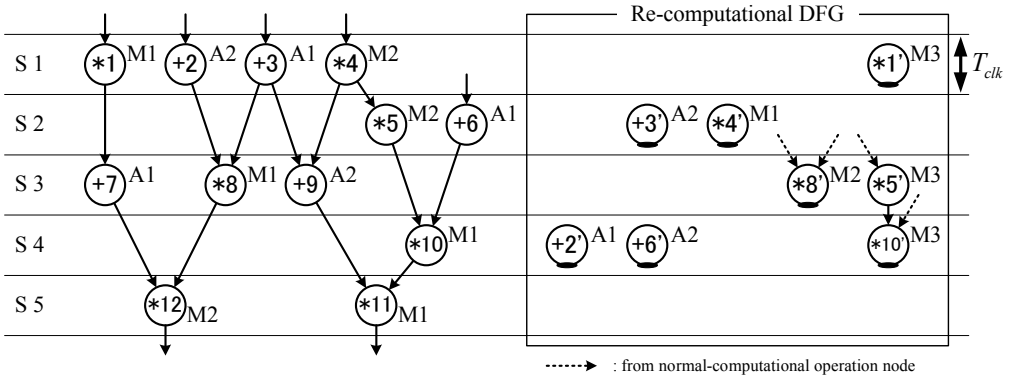


Figure 4.7: Restoring re-computational edges (Step(1.4)).

In Step(1.4), we restore some edges in the re-computational DFG G_R. Let $e' = (m', n') \in E_R$ be a re-computational edge, where m' and n' show a source and a destination operation nodes in the re-computational DFG G_R, respectively. For an operation node n, $cs(n)$ and $fu(n)$ show the step and the FU to which an operation node n has been scheduled/bound, respectively. We restore the edge $e' = (m', n')$ when the following equation holds:

$$cs(m') + dt(fu(m'), fu(n')) < cs(n'). \tag{4.12}$$

After applying Step(1.4), we can know the number of comparison nodes inserted for fault detection.

Example 4.6. *In Step(1.4), for the result as in Example 4.5, we restore the re-computational edges which satisfy Eq. (4.12). In this example, we can restore the edge*

Algorithm 4.2 Re-scheduling/re-binding re-computational DFG (Step (1.5)).

1: Insert an operation node $n' \in V_R$ having no output edges into the list L.
2: Count the number of required comparison nodes C_{req}.
3: **while** $L \neq \emptyset$ **do**
4: Set $n' \leftarrow$ the head of L and remove n' from L.
5: Let $cs(n)$ be the step to which an operation node n is scheduled. Let $EFu(n)$ be the set of FUs which can execute an operation node n. Insert a pair of a step and an FU $\langle cs, fu \rangle$, where $1 \leq cs \leq cs(n')$ and $fu \in EFu(n')$, into the list SFL.
6: **while** $SFL \neq \emptyset$ **do**
7: Set $\langle cs_i, fu_i \rangle \leftarrow$ the head of SFL and remove $\langle cs_i, fu_i \rangle$ from SFL.
8: **if** n' can be scheduled to the step cs_i and bound to the FU fu_i **then**
9: Let n'_o be the re-computational operation node which has been scheduled/bound to $\langle cs_i, fu_i \rangle$. Try to schedule/bind n' to $\langle cs_i, fu_i \rangle$ and re-schedule/re-bind n'_o as soon as possible.
10: Count the number of required comparison nodes C'_{req}.
11: **if** $C'_{req} < C_{req}$ **then**
12: Perform re-scheduling/re-binding of both n' and n'_o.
13: Set $L \leftarrow \emptyset$ and $SFL \leftarrow \emptyset$.
14: Go back to the beginning of this algorithm.
15: **end if**
16: **end if**
17: **end while**
18: **end while**

whose source and destination nodes are '$*5$' and '$*10$', respectively. As shown in Fig. 4.7, seven re-computational operation nodes have no output edges and hence we need to insert a comparison node into each of them. □

In Step(1.5), we re-schedule/re-bind the re-computational DFG G_R with the objective of minimizing the number of comparison nodes inserted for fault detection. We propose **Algorithm 4.2** to realize Step(1.5). In the steps 8–9 of Algorithm 4.2, we try to re-schedule/re-bind a re-computational operation node $n' \in V_R$ in earlier step instead of re-scheduling/re-binding another re-computational operation node $n'_o \in V_R$. In re-scheduling/re-binding any node in G_R, according to the fault-secure conditions, all the requirements described in Step(1.2) must be satisfied. In the steps 11–12 of Algorithm 4.2, we re-schedule/re-bind the nodes n' and n'_o when the number of inserted comparison nodes can be reduced.

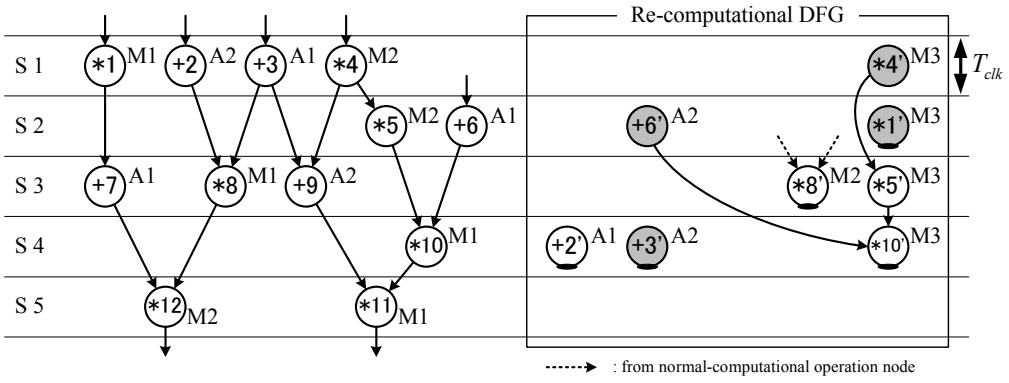


Figure 4.8: Re-scheduling/re-binding of re-computational DFG (Step(1.5)).

Example 4.7. *For the result as in Example 4.6, by applying Algorithm 4.2, we obtain the re-scheduled/re-bound re-computational DFG shown in Fig. 4.8. In this example, '$*4''$ is first re-scheduled to the step 'S1' and re-bound to the FU 'M3' instead of '$*1''$, and then '$+6''$ is re-scheduled to the step 'S2' and re-bound to the FU 'A2' instead of '$+3''$. Every re-computational operation node still satisfies the requirements (1)–(3) described in Step(1.2). As shown in Fig. 4.8, the number of comparison nodes inserted for fault detection can be reduced by* $7 - 5 = 2$. □

4.3.4 Comparator binding (Step2)

According to the fault-secure conditions, comparators are newly allocated to RDR architecture, and then comparison nodes are inserted to compare the outputs of some operation nodes included in the normal-computational DFG G and the re-computational DFG G_R. All the comparison nodes are scheduled/bound under the clock period constraint T_{clk} and the total step constraint S_{max}. The total area constraint A_{max} is not considered also in this step. The objective is to maximize the number of re-computational operation nodes.

In Step(2.1), we newly allocate a comparator for fault detection to each island in RDR architecture.

In Step(2.2), according to the fault-secure conditions (3) and (6), we insert and then schedule/bind comparison nodes to compare the outputs of some operation nodes included in the normal-computational DFG G and the re-computational DFG G_R. We propose **Algorithm 4.3** to realize Step(2.2). In the 2nd step of Algorithm 4.3, we set a priority $pr_c(c)$ to each comparison node c before its scheduling/binding. The priority $pr_c(c)$ is set by the earliest step to which a comparison

Algorithm 4.3 Inserting and scheduling/binding comparison nodes (Step (2.2)).

1: Let $n' \in V_R$ and $n \in V$ be a re-computational operation node which has no output edges and a normal-computational node corresponding to n', respectively. Insert a comparison node c for each pair of n' and n to compare their outputs.
2: Set a priority $pr_c(c)$ to each comparison node c.
3: Insert all the comparison nodes into the list L in the order of priority.
4: **while** $L \neq \emptyset$ **do**
5: Set $c \leftarrow$ the head of L and remove c from L.
6: Schedule/bind c as soon as possible.
7: **end while**

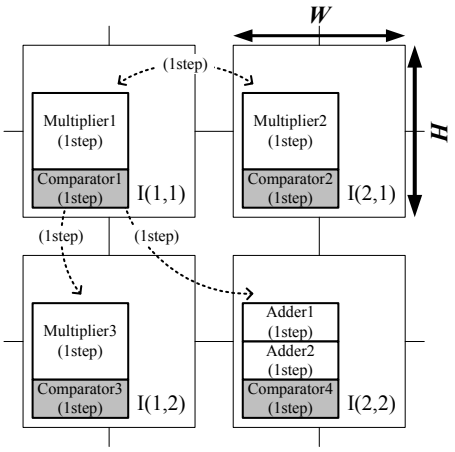


Figure 4.9: Allocating comparators (Step(2.1)).

node c can be scheduled, and is calculated by:

$$pr_n(c) = \min_{fu_c \in EFu(c)} \left\{ \max_{n_p \in PNode(c)} \{cs(n_p) + dt(fu(n_p), fu_c)\} \right\} \tag{4.13}$$

where $EFu(c)$ and $PNode(c)$ show the set of FUs which can execute c and the set of parent operation nodes of c, respectively. $cs(n_p)$ and $fu(n_p)$ show the step and the FU to which an operation node n_p has been scheduled/bound, respectively. By scheduling each comparison node as soon as possible, we expect to maximize the number of comparison nodes which satisfy the total step constraint S_{max}.

Example 4.8. *In Step(2.1), for the initial FU placement shown in Fig. 4.3b, we newly allocate a comparator to each island as shown in Fig. 4.9. For the result as in Example 4.7, by applying Algorithm 4.3, we obtain the scheduled/bound comparison nodes shown*

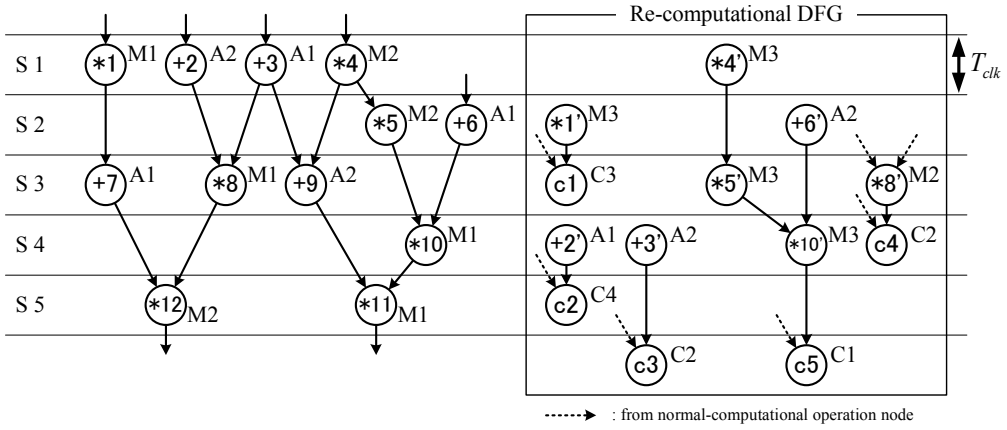


Figure 4.10: Inserting and scheduling/binding of comparison nodes (Step(2.2)).

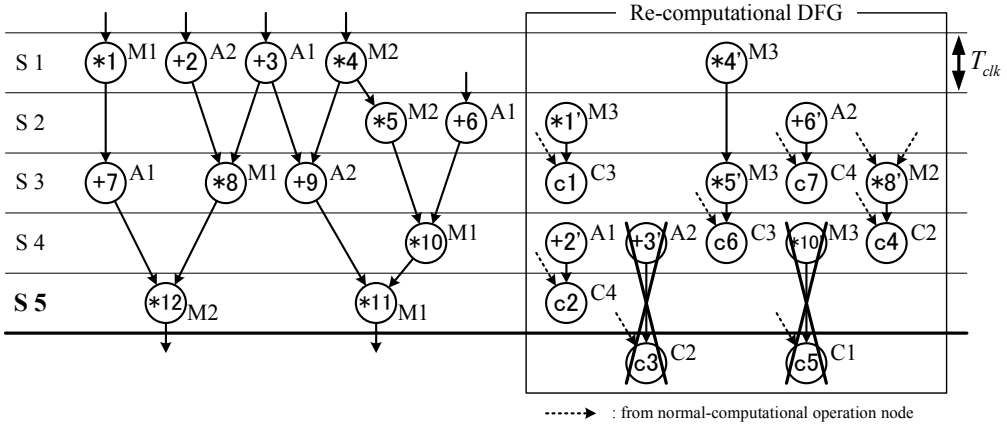


Figure 4.11: Adjusting comparison nodes (Step(2.3)).

in Fig. 4.10. Every comparison node is inserted for a re-computational operation node having no output edges in Fig. 4.8 and scheduled/bound as soon as possible. □

In Step(2.3), considering the total step constraint S_{max}, we remove some comparison nodes. A comparison node c is removed when $cs(c) > S_{max}$ holds, where $cs(c)$ shows the step to which c has been scheduled in Step(2.2). When a comparison node c is removed, removing the re-computational operation node directly connected to c is also required. Since it may result in some re-computational operation nodes having no output edges, according to the fault-secure conditions, we newly insert and schedule/bind comparison nodes. The algorithm of inserting and scheduling/binding comparison nodes is almost the same as Algorithm 4.3.

Example 4.9. *In Step(2.3), for the result as in Example 4.9, we remove the comparison*

nodes which have been scheduled to the steps later than $S_{max} = 5$. In this example, as shown in Fig. 4.11, two comparison nodes '$c3$', '$c5$' are removed and then two re-computational operation nodes '$+3'$', '$*10'$' are also removed. As a result, according to the fault-secure conditions, two comparison nodes '$c6$', '$c7$' are newly inserted and then scheduled/bound as soon as possible. □

4.3.5 Removing duplicated operations (Step3)

Based on the result obtained in Step1 and Step2, the re-computational DFG G_R is re-constructed under the total area constraint A_{max}. The objective is to maximize the number of re-computational operation nodes.

In Step(3.1), through temporary register binding, we evaluate the area of modules allocated to each island in RDR architecture. The register binding algorithm is the same as in [7]. Let $Fu'(i)$, $Reg'(i)$, and $Mux'(i)$ be the set of FUs, registers, and multiplexers which need to be allocated to an island i, respectively. After register binding, the maximum area cost C_A is evaluated by:

$$C_A = \max_{i \in I} \left\{ \sum_{fu \in Fu'(i)} A_{fu} + |Reg'(i)| \times A_{reg} + |Mux'(i)| \times A_{mux} + A'_{cont} \right\}, \quad (4.14)$$

where I shows the set of islands which the target RDR architecture has. A_{fu}, A_{reg}, and A_{mux} show the areas of an FU fu, a register, and a multiplexer, respectively. When evaluating Eq. (4.14), we secure a temporary area A'_{cont} for a controller in each island because controller synthesis is too computationally expensive to run it frequently.

For the total area constraint A_{max}, when the following equation is satisfied, we remove all the comparators unused and complete Step3:

$$W \times H = \frac{A_{max}}{N \times M} \geq C_A. \quad (4.15)$$

On the other hand, when Eq. (4.15) is not satisfied, we attempt to reduce the maximum area cost C_A by removing a re-computational operation node.

In Step(3.2), we evaluate the maximum area cost C_A when removing each re-computational operation node. For an operation node $n' \in V_R$, we try to remove n' and, if necessary, insert and schedule/bind comparison nodes. Based on the re-constructed DFG where n' is newly removed, the maximum area cost $C_A(-n')$ is evaluated by Eq. (4.14) through temporary register binding.

In Step(3.3), we remove a re-computational operation node n' giving the minimum value of $C_A(-n')$ in the re-computational DFG G_R. After that we go back to the beginning of Step3.

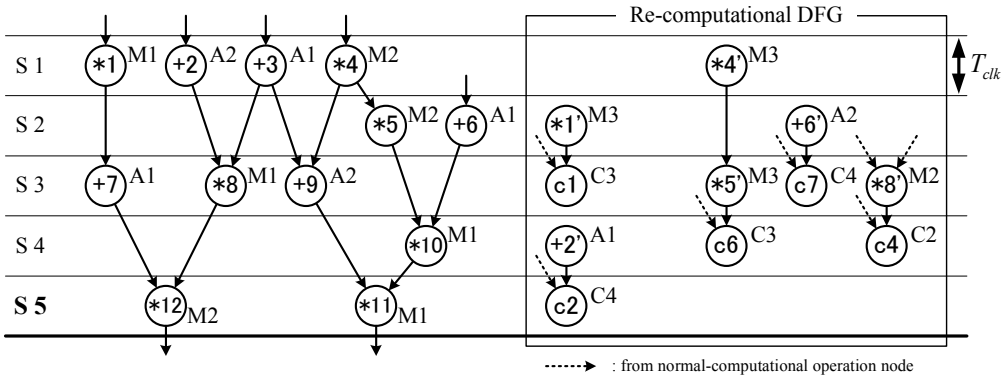


Figure 4.12: Scheduled/bound re-computational DFG output from our partially redundant fault-secure HLS algorithm.

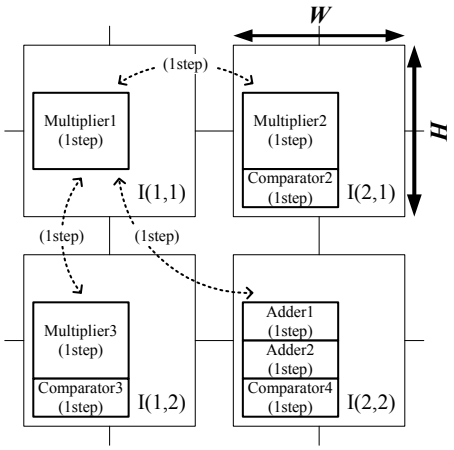


Figure 4.13: FU placement output from our partially redundant fault-secure HLS algorithm.

After completing Step3, we output the scheduled/bound re-computational DFG $G_R = (V_R, E_R)$ and the FU placement on RDR architecture.

Example 4.10. *For the input set as in Example 4.3, by applying our HLS algorithm, we obtain the scheduled/bound re-computational DFG shown in Fig. 4.12 and the FU placement on RDR architecture shown in Fig. 4.13.* □

4.4 Experimental Results

We have implemented our algorithm in C++. The algorithm has been run on AMD Opteron 2360 SE (2.5 GHz, Quad core) with 16 GB memory and applied

Table 4.1: Area and delay of modules.

	Area (μm^2)	Delay (ns)
Adder	282	1.32
Multiplier	4661	2.70
Comparator	255	0.60
Register	288	0.11
Multiplexer	112	0.04

Table 4.2: The set of constraints.

The inputs to MCAS [7].			T_{clk}	The set of constraints.	
App.	$N \times M$	$W \times H$ (μm $\times$ μm)		A_{max} (μm^2)	S_{max}
DCT	2×2	90×90	1.7	$32,400$	20
DCT	3×2	90×90	1.7	$48,600$	13
EWF	2×1	90×90	1.7	$16,200$	27
EWF3	2×2	90×90	1.7	$32,400$	70
FIR	2×2	90×90	1.7	$32,400$	42
FIR	3×2	90×90	1.7	$48,600$	30

to DCT (48 operation nodes), EWF (32 operation nodes), EWF3 (102 operation nodes), and FIR (75 operation nodes). Table 4.1 shows the area and delay of FUs, a register, and a multiplexer used in this experiment, which are all assumed to have 16 bit width and have been synthesized by Synopsys Desgin Compiler under the CMOS 90 nm technology. A controller has been synthesized for each island by Synopsys Desgin Compiler after applying our algorithm. We have set in Eq. (4.3) the interconnect delay coefficient $C_d = 1$ ns/250μm.

For a given description of application behavior, specification of RDR architecture (the number of islands $N \times M$ and the size of islands $W \times H$), and clock period constraint T_{clk}, we have first applied MCAS [7] which is the conventional HLS algorithm and obtained the scheduled/bound normal-computational DFG and its FU placement. The set of constraints input to our algorithm as well as the inputs to MCAS are summarized in Table. 4.2. We have obtained the total step constraints S_{max} in the 6th column of Table. 4.2 from the results of MCAS. The experimental results are shown in Table 4.3 and Table 4.4.

Table 4.3: Experimental results (reliability).

App. #Islands	Algorithm	$A_{max} \times S_{max}$ (μm^2)	Error sensitive reasion (μm^2)				P_{E}
			FU	MUX	Cont.	Total	
DCT	Normal DFG	648000	158176	31024	15640	204840 (1.0000)	31.61%
2×2	Ours		154228	26432	16940	197600 (0.9647)	30.49%
DCT	Normal DFG	631800	158176	26880	8099	193155 (1.0000)	30.57%
3×2	Ours		116940	19824	9243	146007 (0.7559)	23.11%
EWF	Normal DFG	437400	81908	19600	11745	113253 (1.0000)	25.89%
2×1	Ours		69766	14000	14904	98670 (0.8712)	22.56%
EWF3	Normal DFG	2268000	245724	89376	125720	460820 (1.0000)	20.32%
2×2	Ours		205898	83664	139370	428932 (0.9308)	18.91%
FIR	Normal DFG	1360800	306230	44352	48846	399428 (1.0000)	29.35%
2×2	Ours		270650	15344	56364	342358 (0.8571)	25.16%
FIR	Normal DFG	1458000	306230	40992	24720	371942 (1.0000)	25.51%
3×2	Ours		265840	33264	34050	333154 (0.8957)	22.85%

4.5 Conclusion

In this chapter, we have proposed a partially redundant fault-secure HLS algorithm for RDR architecture. The experimental results have shown that our algorithm can improve reliability by up to 24% without any performance/area overhead compared to the conventional approach.

Table 4.4: Experimental results (overhead).

App. #Islands	Algorithm	#Steps	Module area (μm^2)	Each module area (μm^2)				Vacant (μm^2)	CPU time (sec)
				FU	Reg.	MUX	Cont.		
DCT	Normal DFG	20	6775	4661	864	896	354	1325	92.11
2×2			**6948**	4661	1152	784	351	1152	
			4833	564	1728	1904	637	3267	
			6293	564	2304	2688	737	1807	
	Ours	20	6775	4661	864	896	354	1325	174.31
			6957	4661	1152	784	360	1143	
			7430	819	2304	3472	835	670	
			7837	819	2592	3584	842	263	
DCT	Normal DFG	13	6393	4661	1152	336	244	1707	135.11
3×2			6090	4661	864	336	229	2010	
			6084	4661	864	336	223	2016	
			6524	4661	1152	448	263	1576	
			5271	564	1728	2464	515	2829	
			5480	564	1728	2576	612	2620	
	Ours	13	6782	4916	1152	448	266	1318	149.97
			6760	4916	1152	448	244	1340	
			6445	4916	864	448	217	1655	
			6895	4916	1152	560	267	1205	
			7733	819	2880	3360	674	367	
			7478	819	2304	3584	771	622	
EWF	Normal DFG	27	**6003**	4661	576	448	318	2097	45.59
2×1			5598	564	1728	2576	730	2502	
	Ours	27	6410	4661	576	784	389	1690	82.04
			7889	819	2592	3472	1006	211	
EWF3	Normal DFG	70	6241	4661	576	560	444	1859	99.99
2×2			6321	4661	576	560	524	1779	
			5664	564	1440	2576	1084	2436	
			7593	564	1728	4032	1269	507	
	Ours	70	6709	4916	576	672	545	1391	2676.90
			6954	4916	576	896	566	1146	
			7366	819	1728	3584	1235	734	
			7583	564	1728	4032	1259	517	
FIR	Normal DFG	42	6712	4661	864	784	403	1388	93.09
2×2			**6890**	4661	864	896	469	1210	
			4359	564	1152	1904	739	3741	
			6326	282	2592	2576	875	1774	
	Ours	42	7133	4916	864	896	457	967	140.31
			7253	4916	864	1008	465	847	
			8039	819	2304	3808	1108	61	
			7669	537	3456	2688	987	431	
FIR	Normal DFG	30	6109	4661	864	336	248	1991	136.38
3×2			5812	4661	576	336	239	2288	
			5798	4661	576	336	225	2302	
			6148	4661	864	336	287	1952	
			3234	282	1152	1344	456	4866	
			6537	564	1440	3696	837	1563	
	Ours	30	6775	4916	864	672	323	1325	473.37
			6816	4916	864	672	364	1284	
			6957	4916	1152	560	329	1143	
			6401	4661	864	560	316	1699	
			6637	537	2016	3248	836	1463	
			7860	564	2016	4368	912	240	

Chapter 5

A Low-overhead Fully Redundant Fault-secure High-level Synthesis Algorithm[1]

5.1 Introduction

In this chapter, a low-overhead fully redundant fault-secure HLS algorithm for DR architectures is proposed. Also in this HLS algorithm, RDR architecture is utilized. As described in Section 2.4, a fully redundant fault-secure HLS algorithm for RDR architecture has already been proposed in [24]. The conventional algorithm has tried to maximize circuit performance under a given functional unit constraint. However, the area overhead is significantly large because it has only considered the number of required functional units and ignored the cost of registers, multiplexers, and controllers. Due to a large amount of area overhead, the size of islands in RDR architecture becomes large and hence performance degradation may be caused. As illustrated in Fig. 5.1, a given clock period constraint (T_{clk}) and an interconnect delay between two islands ($D_c(i_p, i_q)$) increase with increase in the size of islands ($W \times H$). Our proposed algorithm tries to maximize circuit performance under a given *area* constraint. In contrast to conventional approaches like [24] or [26], our proposed algorithm adopts an integrative approach which concurrently performs scheduling, binding, allocation, and register synthesis. Since this approach monitors the cost of not only functional units but also registers and multiplexers during scheduling/binding, the area/performance overhead can be reduced sufficiently. Experimental results demonstrate that our proposed algorithm reduces area by up to 47% and improves performance by up to 41% compared to the conventional

[1]Technical contents in this chapter have been presented in the publication ⟨11⟩.

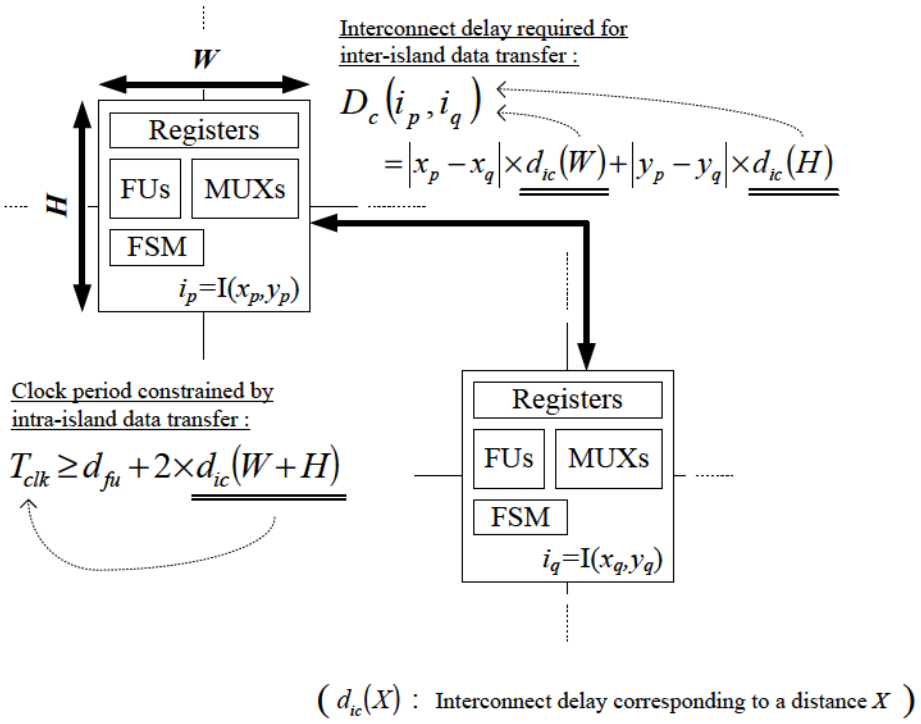


Figure 5.1: Impact of an area overhead on circuit performace in RDR architecture. The size of islands ($W \times H$) has a direct effect on a given clock period constraint (T_{clk}) and an interconnect delay between two islands ($D_c(i_p, i_q)$).

fault-secure HLS algorithm for RDR architecture.

5.2 Problem Formulation

In HLS, behavioral description used as input is transformed into graphical representation like a control-data flow graph (CDFG) or a data-flow graph (DFG). Hereafter we use a DFG as an input for simplicity. Note that the discussion below can be easily extended to a CDFG.

A normal-computational 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 show the data dependencies between two operation nodes. In fully redundant fault-secure HLS, its re-computational DFG $G_R = (V_R, E_R)$ is generated by duplicating the normal-computational DFG $G = (V, E)$. Given a scheduling/binding result of the normal-computational DFG G, its re-computational DFG G_R is scheduled/bound so that the *fault-secure conditions* (1) to (6) below are satisfied:

(1) The scheduling/binding result of the normal-computational DFG G is not

changed.

(2) Let $n' \in V_R$ be one of the re-computational operation nodes, and $n \in V$ be the normal-computational operation node corresponding to n'. They cannot share an identical functional unit (FU).

(3) All the outputs of the normal-computational DFG G and the re-computational DFG G_R must be compared.

(4) An edge $e' \in E_R$ can be broken.

(5) Let $e' = (m', n') \in E_R$ be the input edge of n' in the re-computational DFG G_R, where m' is one of the parent nodes of n'. Let m be the normal-computational operation node corresponding to m'. If e' is broken, n' uses the output of the normal-computational operation node m instead of m'.

(6) The outputs of m and m' must be compared if all the output edges of m' are broken.

In RDR architecture, an entire circuit is assigned to $N \times M$ array of islands. Let $I(x, y)$ be an island located at x-th column and y-th row of the array, where $1 \leq x \leq N$ and $1 \leq y \leq M$. All the islands assume to be square and have the same size. Let W and H be the width and the height of islands, respectively. The size of islands $W \times H$ is calculated as

$$W \times H = \frac{A_{max}}{N \times M}, \tag{5.1}$$

where A_{max} denotes the given total area constraint.

An RTL circuit is composed of FUs, registers, multiplexers, and controllers. Let $Fu(i)$, $Reg(i)$, and $Mux(i)$ be the set of FUs, registers, and multiplexers which are allocated to an island i, respectively. Any island i in RDR architecture satisfies

$$W \times H \geq \sum_{fu \in Fu(i)} A_{fu} + |Reg(i)| \times A_{reg} + |Mux(i)| \times A_{mux} + A_{cont}(i), \tag{5.2}$$

where A_{fu}, A_{reg}, and A_{mux} show the areas of an FU fu, a register, and a multiplexer, respectively. $A_{cont}(i)$ shows the area of a controller synthesized for controlling the datapath on an island i.

Let T_{clk} be the given clock period constraint. In order to ensure that execution of an operation and intra-island data transfer are completed in T_{clk}, any island i satisfies

$$\max_{fu \in Fu(i)} d_{fu} + 2 \times d_{ic}(W + H) \leq T_{clk}, \tag{5.3}$$

where d_{fu} shows the delay of an FU fu and $d_{ic}(X)$ shows the interconnect delay corresponding to a distance X. Eq. (5.3) states that the clock period is constrained by the sum of the largest FU delay and the worst-case interconnect delay within an island.

Let i_1 and i_2 be two islands $I(x_1, y_1)$ and $I(x_2, y_2)$, respectively. An interconnect delay $D_c(i_1, i_2)$ between the two islands i_1 and i_2 is proportional to their distance and it is given by

$$D_c(i_1, i_2) = d_{ic}(W) \times |x_1 - x_2| + d_{ic}(H) \times |y_1 - y_2|. \tag{5.4}$$

Let fu_1 be one of the FUs allocated to the island i_1, which has a delay of d_{fu_1}. Assume that the output of fu_1 is continuously used by an FU allocated to the island i_2. For the clock period constraint T_{clk} and the FU fu_1, the computation by fu_1 can be done in $s_{fu_1} = \left\lceil \frac{d_{fu_1}}{T_{clk}} \right\rceil$ steps. When

$$D_c(i_1, i_2) + d_{fu_1} \leq s_{fu_1} \times T_{clk} \tag{5.5}$$

holds, the computation by fu_1 and its output data transfer to a register allocated to the island i_2 are done in s_{fu_1} steps. On the other hand, when

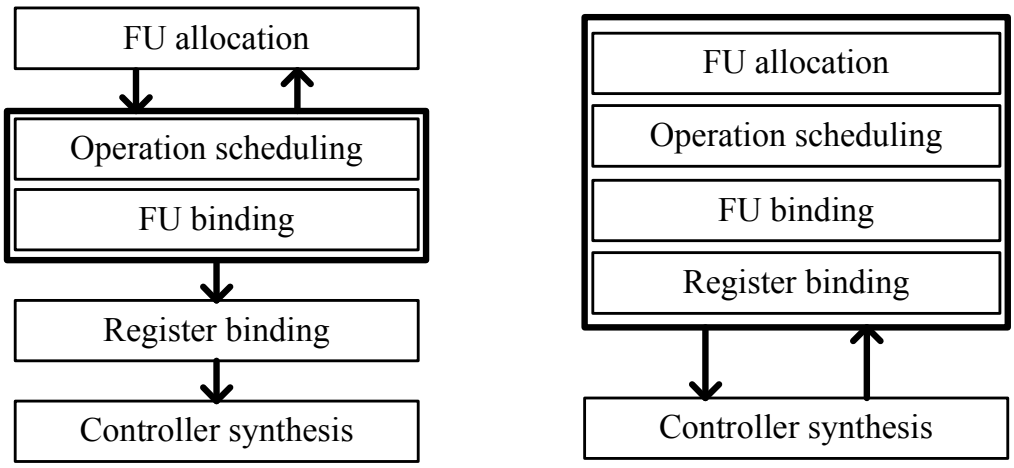
$$D_c(i_1, i_2) + d_{fu_1} > s_{fu_1} \times T_{clk} \tag{5.6}$$

holds, the computation by fu_1 is done in s_{fu_1} steps and its output is stored into a register allocated to the island i_1 at the s_{fu_1}-th step. After that the stored data is transferred to a register allocated to the island i_2 using $\left\lceil \frac{D_c(i_i, i_2)}{T_{clk}} \right\rceil$ steps. Therefore, when two continuous operations are serially executed by fu_1 and fu_2, the step required only for data transfer between the two operations is summarized as follows:

$$dt(fu_1, fu_2) = \begin{cases} 0 & (D_c(i_1, i_2) + d_{fu_1} \leq s_{fu_1} \times T_{clk}) \\ \left\lceil \frac{D_c(i_i, i_2)}{T_{clk}} \right\rceil & (D_c(i_1, i_2) + d_{fu_1} > s_{fu_1} \times T_{clk}) \end{cases} \tag{5.7}$$

Now we define a fully redundant fault-secure HLS problem for RDR architecture as follows:

Definition 5.1. *For a given normal-computational DFG $G = (V, E)$ which has already been scheduled/bound, specification of RDR architecture (the number of islands $N \times M$), FU placement on this RDR architecture, library of FUs, clock period constraint T_{clk}, and total area constraint A_{max}, our fully redundant fault-secure HLS problem for this RDR architecture is to schedule/bind its re-computational DFG $G_R = (V_R, E_R)$, to allocate additional FUs to each island unless its initial FU placement is changed, and to synthesize registers and controllers. The objective here is to minimize the number of required steps.* □



(a) Conventional (sequential) approach.

(b) Our (integrative) approach.

Figure 5.2: Comparison of conventional and our fault-secure HLS algorithms.

5.3 The Algorithm

In this section, we propose a low-overhead fully redundant fault-secure HLS algorithm for RDR architecture. First we discuss the strategy and then propose our algorithm.

5.3.1 Strategy

To solve the problem defined in Section 5.2, we need to consider *how a given total area constraint can be satisfied* in fault-secure HLS. In order to answer this question, it is important to (a) evaluate the area of modules allocated to each island and then (b) minimize the area cost as much as possible.

(a) Area evaluation in HLS flow

The conventional algorithm [24] has tried to minimize the number of required steps under a given FU constraint. In [24], scheduling/FU binding is first performed under an FU constraint and then registers/controllers are synthesized based on the result obtained in the precedent phases. Fig. 5.2a shows the brief synthesis flow of the fault-secure HLS algorithm. In such a sequential approach, the total module area is informed only after completing the flow and hence the area may become large due to registers, multiplexers, and controllers.

In order to tackle this problem, it is required to adopt an integrative approach which concurrently performs scheduling, FU binding, FU allocation, and register

binding. The brief synthesis flow of our fault-secure HLS algorithm is proposed in Fig. 5.2b. Since such an integrative approach can tentatively inform the module area even when DFGs are partly scheduled/FU bound, we can expect to evaluate the area of modules at any point in the flow.

(b) Area cost minimization in scheduling/FU binding of re-computational DFG

In fully redundant fault-secure HLS, its circuit area and circuit performance may have a trade-off relation in general. To relax the total area constraint enables us to allocate new additional modules for re-computation and we can expect to improve circuit performance. On the other hand, by relaxing the total step constraint and improving module reusability, we can expect to reduce circuit area. Especially in terms of reusability, it is preferable that re-computational operations should be delayed as much as possible to normal-computational operations.

When we focus on module reusability, the conventional algorithm [24] has two concerns; one is the scheduling algorithm and the other is the way to break re-computational edges, which are adopted in [24] to minimize the number of required steps. However, they may increase the number of modules such as registers and multiplexers which consist of the datapath circuit since the scheduling is performed in an ASAP manner and the additional cost of registers/multiplexers is not considered in operation scheduling and breaking edges.

For these reasons, we start our fault-secure HLS algorithm with the minimum total step constraint and then gradually relax the constraint in the synthesis flow. Our operation scheduling is performed in an ALAP manner in order to maximally delay the operation timing of re-computation to normal-computation and improve module reusability. By adopting an integrative approach as in Fig. 5.2b, the area cost of not only FUs but also registers and multiplexers are always monitored and then minimized. In our scheduling/FU binding, some re-computational edges will be broken only when the additional cost of registers/multiplexers with edge breaking becomes smaller than the additional cost of FUs without it.

5.3.2 Overall synthesis flow

Fig. 5.3 shows our proposed algorithm.

Example 5.1. *Fig. 5.4 shows an input example of fully redundant fault-secure HLS for RDR architecture. The scheduled/bound normal-computational DFG and the FU placement in RDR architecture are obtained by applying MCAS, which are shown in Figs. 5.4a and 5.4b, respectively. In our HLS algorithm, we give the total area constraint*

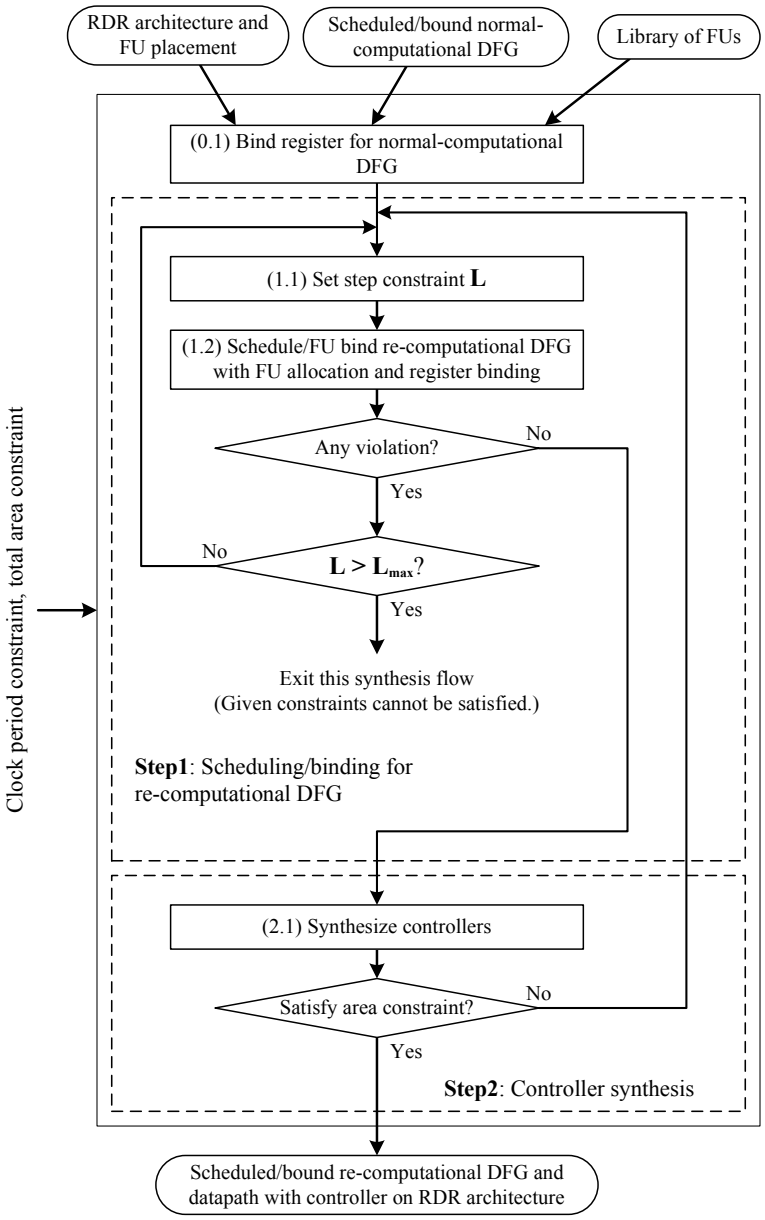
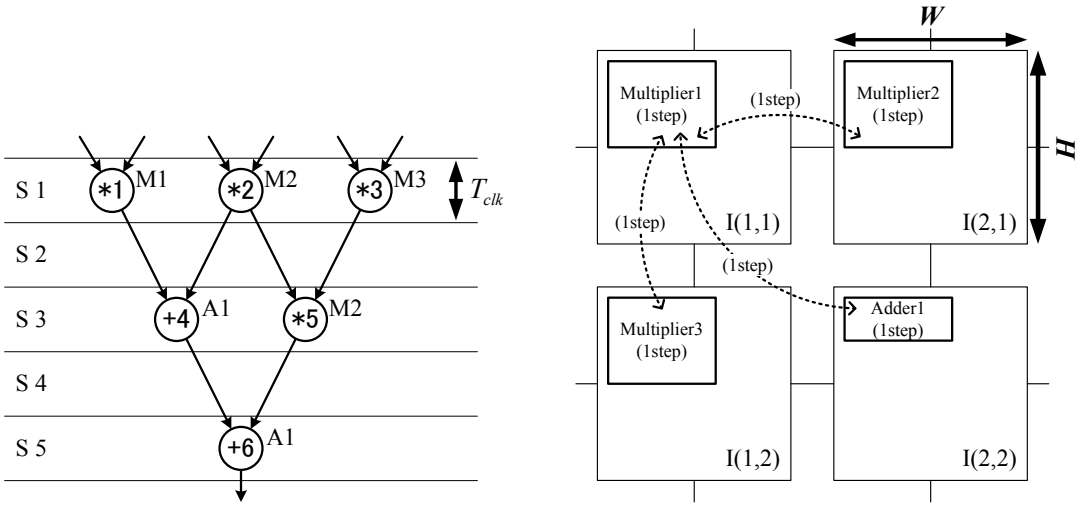


Figure 5.3: The proposed fully redundant fault-secure HLS algorithm.

A_{max} and constrain the area of modules which can be allocated to each island. In this example, we set $A_{max} = 25,600$ and hence the island size $W \times H$ is set to be $6,400$. As in Fig. 5.4b, we assume that one step is required to transfer data between each island as well as to execute an operation by each FU.

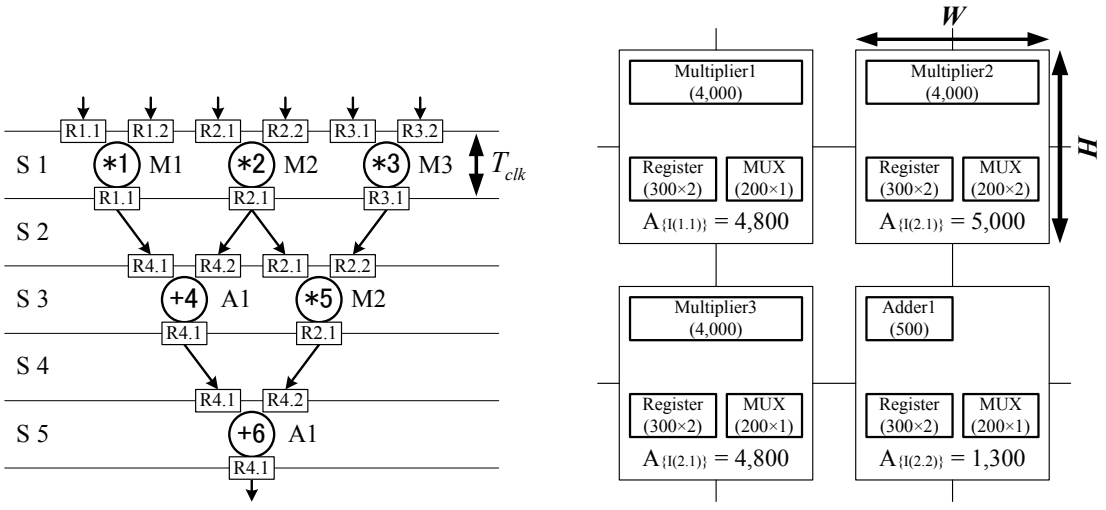
For the input set as in Fig. 5.4, we first bind registers for the scheduled/bound normal-computational DFG in Step(0,1). Fig. 5.5a shows the result of register binding. By binding registers, we can find out the area of modules for the datapath which realizes



(a) Scheduled/bound normal-computational DFG.

(b) RDR architecture and its initial FU placement.

Figure 5.4: An input example of fully redundant fault-secure HLS for RDR architecture.



(a) Register binding result.

(b) Area.

Figure 5.5: Register binding.

the behavior of normal-computational DFG. The total area of modules allocated to each island is shown in Fig. 5.5b. □

5.3.3 Scheduling/binding for duplicated operations with FU allocation and register synthesis

Under the clock period constraint T_{clk}, the total step constraint S_{max}, and the total area constraint A_{max}, we schedule/FU bind the re-computational DFG $G_R = (V_R, E_R)$. In contrast to conventional approaches like [24] or [7], our scheduling/FU binding algorithm is integrated with FU allocation and register binding so that we can accurately monitor the area of modules allocated to every island during scheduling/FU binding. By adopting an integrative approach which concurrently performs scheduling, FU binding, FU allocation, and register binding, we can evaluate the area even when DFGs have been partly scheduled/FU bound. Let $Fu'(i)$, $Reg'(i)$, and $Mux'(i)$ be the set of FUs, registers, and multiplexers which need to be tentatively-allocated to an island i, respectively. At any point in scheduling/FU binding, the area of modules allocated to an island i is evaluated by:

$$A_i = \sum_{fu \in Fu'(i)} A_{fu} + |Reg'(i)| \times A_{reg} + |Mux'(i)| \times A_{mux} + A'_{cont}(i), \tag{5.8}$$

where A_{fu}, A_{reg}, and A_{mux} show the areas of an FU fu, a register, and a multiplexer, respectively. When evaluating Eq. (5.8), we secure in each island a temporary area $A'_{cont}(i)$ for a controller because controller synthesis is too computationally expensive to run it frequently. Let $A_{cont}(i)_j$ be the controller's area synthesized for an island i in the j-th iteration of Step1 and Step2 in Fig. 5.3. In Step(1.2) of its k-th iteration, $A'_{cont}(i)$ is calculated for each island as:

$$A'_{cont}(i) = \begin{cases} 0 & (k = 1) \\ \max\limits_{1 \le j \le k-1} A_{cont}(i)_j & (k \ge 2) \end{cases} \tag{5.9}$$

We propose **Algorithm 5.1** and **Algorithm 5.2** to realize Step(1.2). Algorithm 5.1 shows the overall flow of Step(1.2). It should be noted that our scheduling/FU binding for a re-computational DFG is performed in an ALAP manner. As described in Section 5.3.1, it may enable us to maximally delay the operation timing of re-computation to normal-computation, and hence we can expect to improve module reusability such as FUs and registers.

In the 1st step of Algorithm 5.1, a priority $pr(n')$ is set to each re-computational operation node $n' \in V_R$ before scheduling/FU binding of the re-computational DFG $G_R = (V_R, E_R)$. The priority $pr(n')$ is set based on the critical path length from the start of G_R and is calculated by:

$$pr(n') = \min_{fu_d \in EFu(n')} \{cpl(n', fu_d)\} \tag{5.10}$$

Algorithm 5.1 Schedule/FU bind re-computational DFG with FU allocation and register binding.

1: Set a priority $pr(n')$ to each re-computational operation node $n' \in V_R$.
2: $cs \leftarrow S_{max} - 1$.
3: **while** scheduling/FU binding of the re-computational DFG $G = (V_R, E_R)$ has not been finished yet **do**
4: **if** $cs = 0$ **then**
5: Report violation and exit algorithm.
6: **end if**
7: Insert an operation node $n' \in V_R$, which all the child nodes of n' have already been scheduled, into the ready list RL.
8: Arrange the operation nodes included in RL according to their priorities.
9: **while** $RL \neq \emptyset$ **do**
10: Set $n' \leftarrow$ the head of RL and remove n' from RL.
11: Search an optimal scheduling/FU binding solution $\langle cs', fu' \rangle$ for n' (**Algorithm 5.2**).
12: **if** $cs' = cs$ **then**
13: Schedule/FU bind n' to $\langle cs', fu' \rangle$ with FU allocation and register binding. A comparison node is inserted and scheduled/bound, if necessary.
14: **end if**
15: **if** area violation occurs **then**
16: Report violation and exit algorithm.
17: **end if**
18: **end while**
19: $cs \leftarrow cs - 1$.
20: **end while**

$$cpl(n', fu_d) = \max_{n'_p \in PNode(n')} \left\{ \min_{fu_s \in EFu(n'_p)} \{dt(fu_s, fu_d) + cpl(n'_p, fu_s) + 1\} \right\} \quad (5.11)$$

where $EFu(n')$ and $PNode(n')$ show the set of FUs which can execute n' and the set of parent operation nodes of n', respectively. $dt(fu_s, fu_d)$ is calculated by Eq. (5.7).

In the 11th and 23th steps of Algorithm 5.2, the area cost, when we try to schedule/bind a re-computational operation node n' to a step cs and an FU fu, is evaluated. In evaluating the cost, if necessary, additional FUs can be allocated to islands and comparison nodes can be inserted and scheduled/bound in order to satisfy the fault-secure conditions. Moreover, registers are bound to the input and output edges of n'. Note that we bind comparison nodes and registers according

Algorithm 5.2 Search an optimal scheduling/FU binding solution for a re-computational operation node.

Require: operation node n', current step cs_c

1: $\langle cs_{opt}, fu_{opt} \rangle \leftarrow \langle null, null \rangle$.
2: Insert all the islands into the priority list PL in ascending order of module areas.
3: **while** $PL \neq \emptyset$ **do**
4: Set $i \leftarrow$ the head of PL and remove i from PL.
5: **repeat**
6: Select one of the FUs fu allocated to i.
7: **if** n' can be executed by fu **and** n, which is the normal-computational operation node corresponding to n', has not been bound to fu **then**
8: Search the maximum step cs_{max} ($\leq cs_c$) to which n' can be scheduled with no edge breaking when binding n' to fu.
9: $cs \leftarrow cs_{max}$.
10: **while** $cs \leq cs_c$ **do**
11: Calculate the cost *cost* when n' is scheduled/bound to $\langle cs, fu \rangle$.
12: **if** *cost* is minimum so far **then**
13: $\langle cs_{opt}, fu_{opt} \rangle \leftarrow \langle cs, fu \rangle$.
14: **end if**
15: $cs \leftarrow cs + 1$.
16: **end while**
17: **end if**
18: **until** all the FUs allocated to i are once selected.
19: An additional FU fu_a which can execute n' is virtually-allocated to i.
20: Search the maximum step cs_{max} ($\leq cs_c$) to which n' can be scheduled with no edge breaking when binding n' to fu_a.
21: $cs \leftarrow cs_{max}$.
22: **while** $cs \leq cs_c$ **do**
23: Calculate the cost *cost* when n' is scheduled/bound to $\langle cs, fu_a \rangle$.
24: **if** *cost* is minimum so far **then**
25: $\langle cs_{opt}, fu_{opt} \rangle \leftarrow \langle cs, fu_a \rangle$.
26: **end if**
27: $cs \leftarrow cs + 1$.
28: **end while**
29: **end while**

Ensure: a pair of scheduled step and bound FU $\langle cs_{opt}, fu_{opt} \rangle$

to the rules as follows:

- Let $n \in V$ be a normal-computational operation node corresponding to n'. When a comparison node c is inserted for a pair of n' and n to compare their outputs, c is bound to a comparator in the island to which fu is allocated and is scheduled as soon as possible.

- Registers are bound so that the area cost associated with registers and multiplexers may become minimum, which can be obtained by a nearly full-search algorithm.

Let I be the set of islands which the target RDR architecture has. Since all the islands in RDR architecture assume to have the same size, it is important to minimize the maximum area $\left\{\max_{i \in I} A_i\right\}$ when we schedule/FU bind operation nodes with consideration for the total area constraint A_{max}. Now we propose the following equation to evaluate the area cost *cost* after trying to schedule/FU bind n' with FU allocation and register binding:

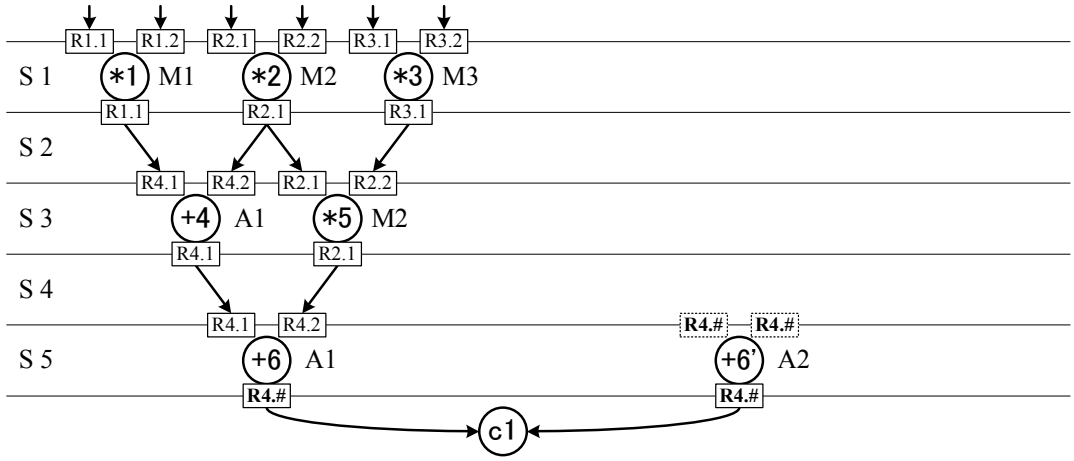
$$cost = \sum_{i \in I} A_i + N \times M \times \max_{i \in I} A_i \tag{5.12}$$

By utilizing Eq. (5.12), we achieve scheduling/FU binding of the re-computational DFG $G_R = (V_R, E_R)$ with the primary objective of minimizing the maximum area $\left\{\max_{i \in I} A_i\right\}$ and the secondly objective of minimizing the total area $\left\{\sum_{i \in I} A_i\right\}$.

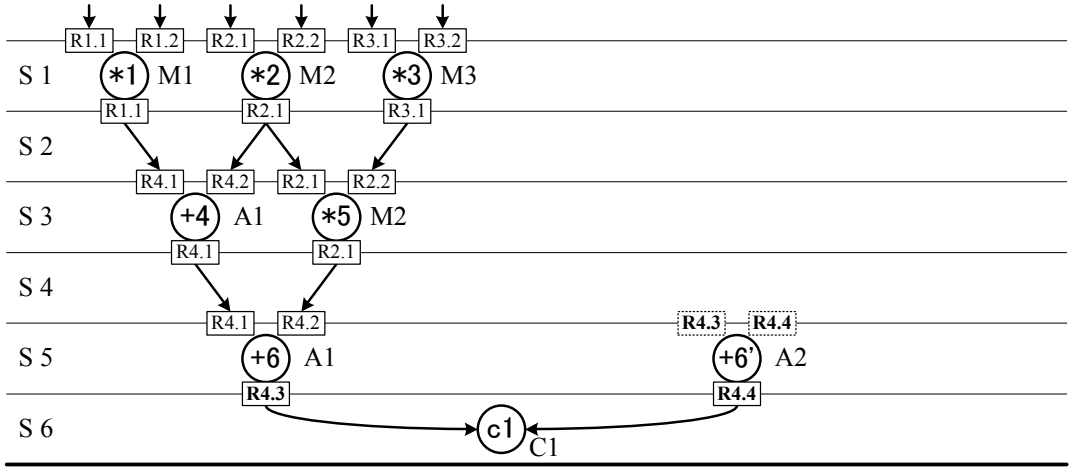
In the 13th step of Algorithm 5.1, each re-computational operation node is scheduled/FU bound according to the evaluation by Eq. (5.12), and then allocating new FUs, inserting and scheduling/binding a comparison node, and binding registers are concurrently performed as necessary.

Example 5.2. *Assume that the total step constraint S_{max} is set to be 6 in Step(1.1). In the following, the island size is set to be $6,400$ as in Example 5.1 and, for simplicity, the area of a controller allocated to each island is assumed to be zero. In Step(1.2), based on the normal-computational DFG shown in Fig. 5.5a, the re-computational DFG is scheduled/FU bound.*

According to Algorithm 5.1, we first seek to schedule/FU bind the re-computational operation node '+6′' which has no child operation nodes. According to Algorithm 5.2, we select the island $I(2,2)$ having the minimum area of modules which have been allocated. Binding '+6′' to the adder 'A1' cannot be realized because the normal-computational operation node '+6' has been bound to 'A1'. For this reason, an additional adder 'A2' is newly allocated to the island $I(2,2)$ and the re-computational operation node '+6′



(a) Scheduling/FU binding of '$+6'$' and inserting a comparison node '$c1$'.



(b) Scheduling/FU binding of '$c1$' and register binding.

Figure 5.6: Scheduling/FU binding '$+6'$' to $\langle S5, A2\rangle$.

is scheduled/FU bound to $\langle S5, A2\rangle$ *as shown in Fig. 5.6a. In addition, as shown in Fig. 5.6b, a comparison node '*$c1$*' is inserted and scheduled/bound since '*$+6'$*' has no child operation nodes. As a result, we obtain module allocation shown in Fig. 5.7.*

We next seek to schedule/FU bind the re-computational operation node '$+5'$*'. According to Algorithm 5.2, we select the island* $I(2, 2)$ *having the minimum area of modules which have been allocated. Since there have been no multipliers in this island, we virtually-allocate an additional multiplier '*$M4$*' and try to schedule/FU bind the re-computational operation node '*$+5'$*' to* $\langle S4, M4\rangle$*. Fig. 5.8a shows the result of this scheduling/FU binding and the cost is calculated as cost* $= 49,100$ *as in Fig. 5.8b.*

According to Algorithm 5.2, we then select the island $I(1, 1)$*. The re-computational operation node '*$+5'$*' can be bound to the multiplier '*$M1$*' and can be scheduled to the step*

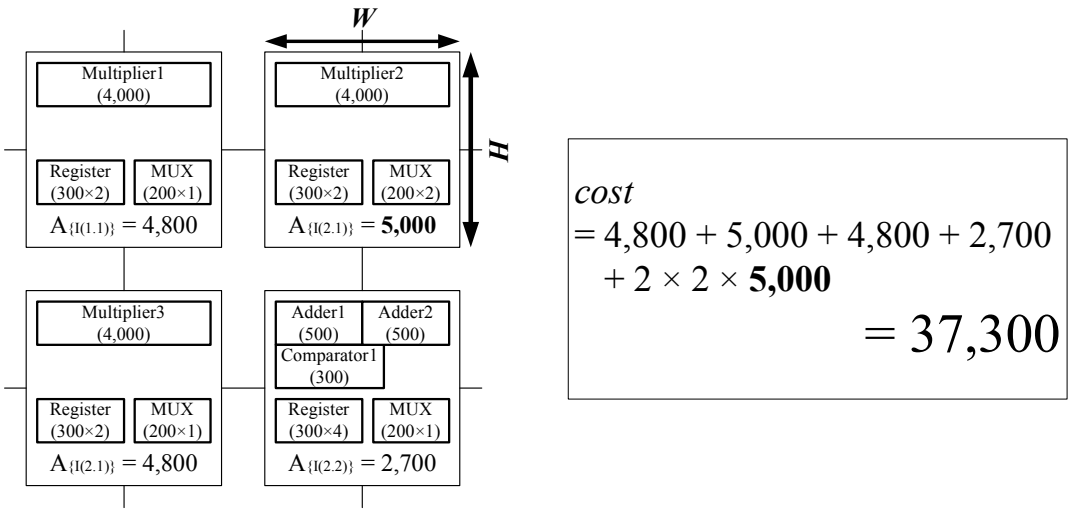
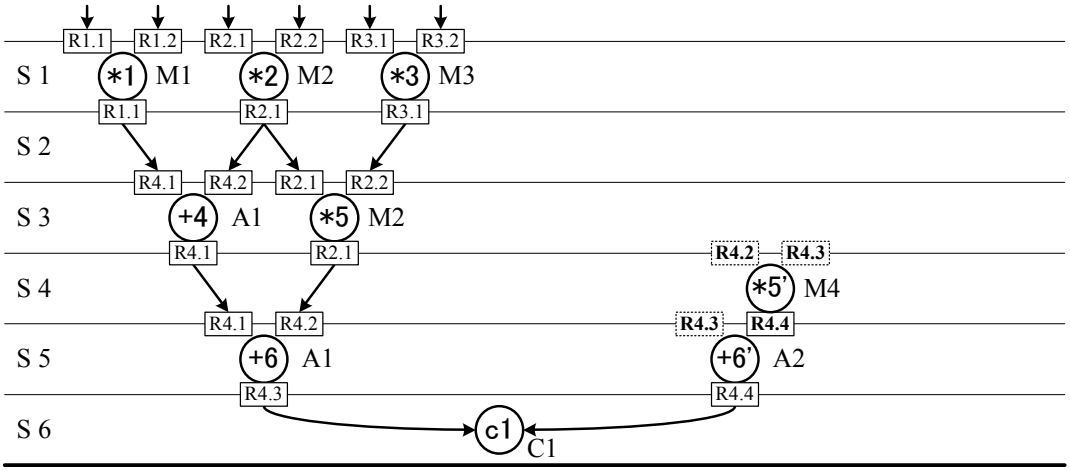


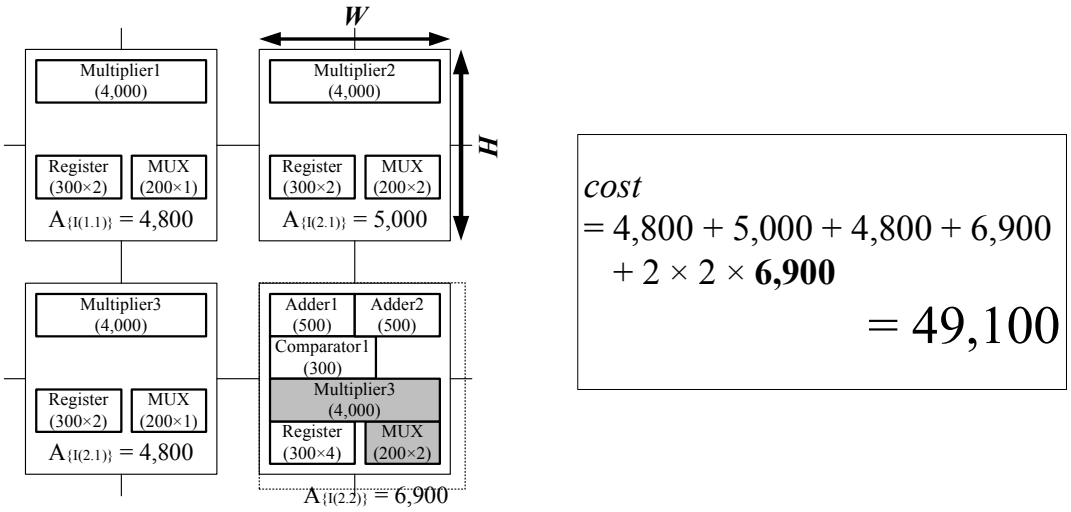
Figure 5.7: Area cost after scheduling/FU binding of '+6''.

'S3' with no edge breaking as shown in Fig. 5.9a. For this scheduling/FU binding result, the cost is calculated as cost $= 37,500$ *as in Fig. 5.9b. The re-computational operation node '+5'' also can be scheduled to the step 'S3' and bound to the multiplier 'M1' with edge breaking as shown in Fig. 5.10a. For this scheduling/FU binding result, the cost is calculated as cost* $= 39,000$ *as in Fig. 5.10b. The re-computational operation node '+5'' is finally scheduled/FU bound to* $\langle S3, M1 \rangle$ *by evaluating the value of cost.*

Fig. 5.11 shows the scheduling/FU binding result of the re-computational DFG which is obtained after Step(1.2). □

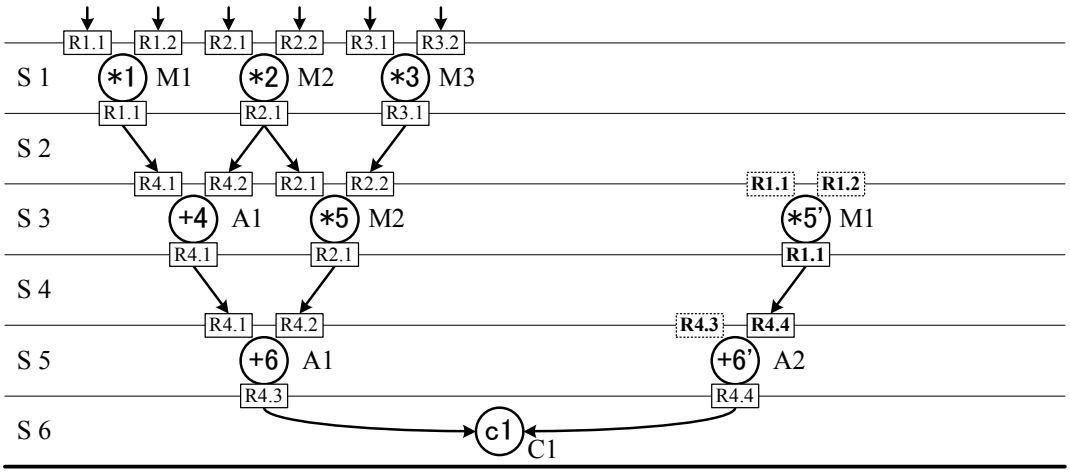


(a) Scheduling/FU binding and register binding.

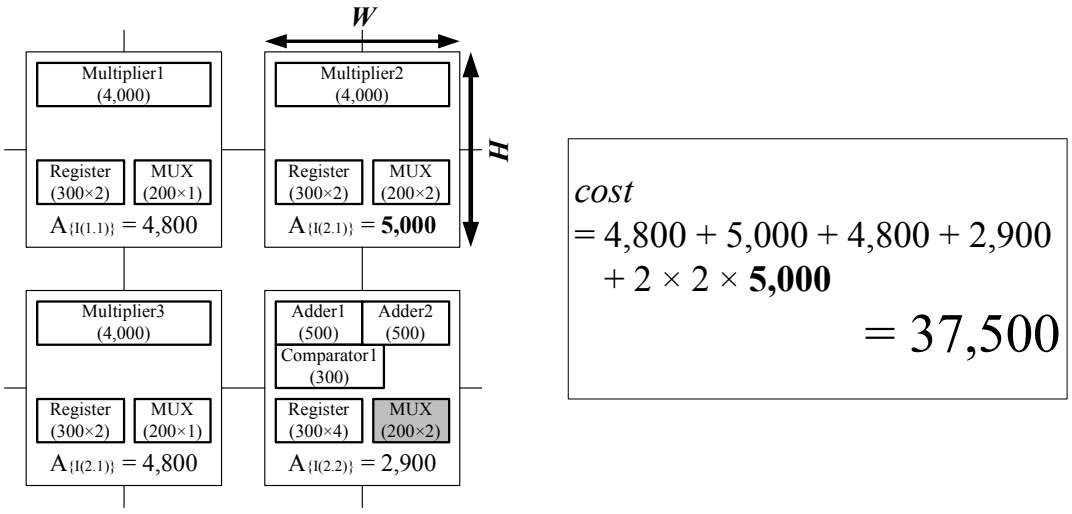


(b) Area cost.

Figure 5.8: Scheduling/binding '+5'' to $\langle S4, M4 \rangle$.

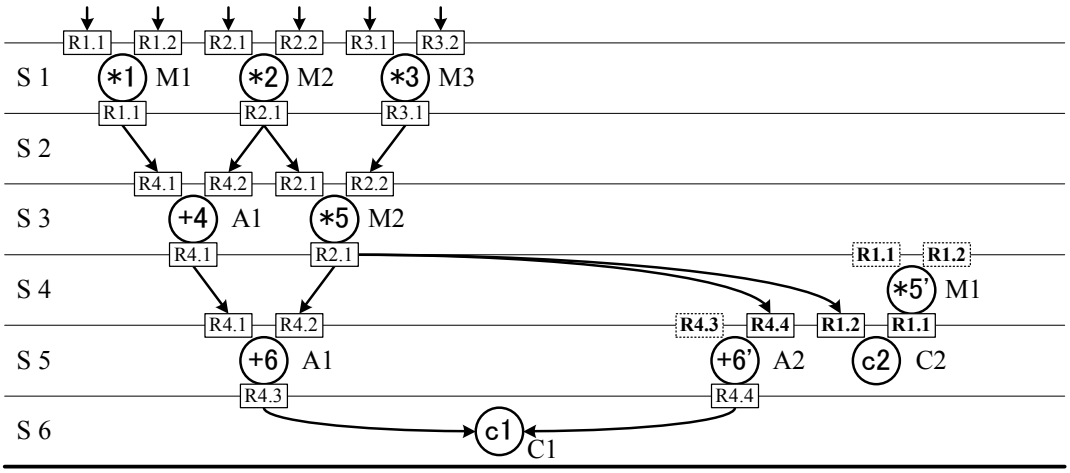


(a) Scheduling/FU binding and register binding.

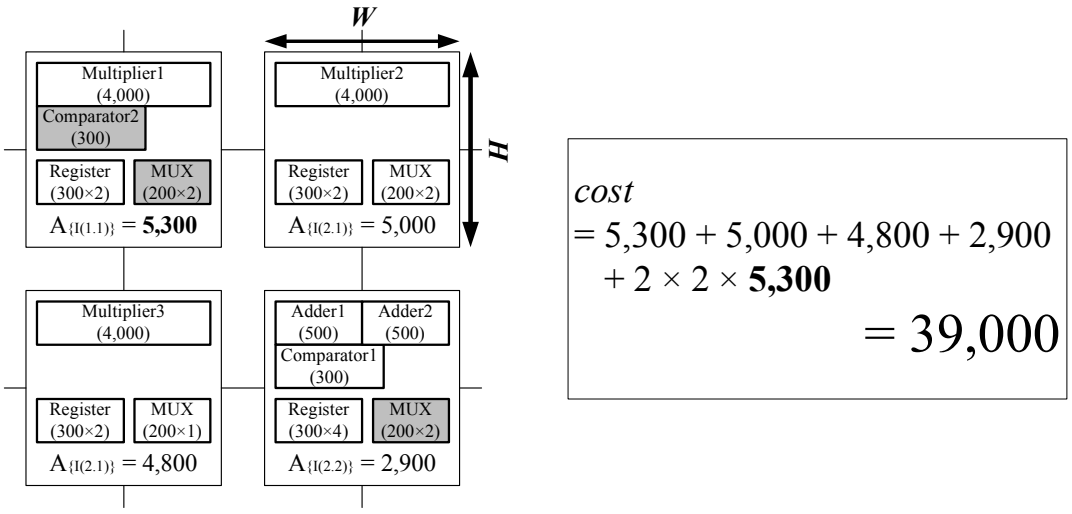


(b) Area cost.

Figure 5.9: Scheduling/binding '+5'' to $\langle S3, M1 \rangle$.

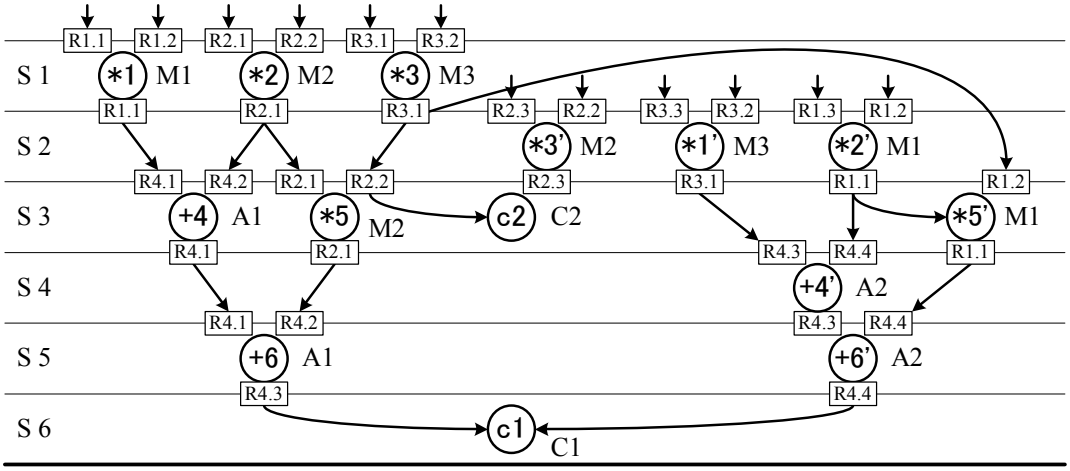


(a) Scheduling/FU binding and register binding.

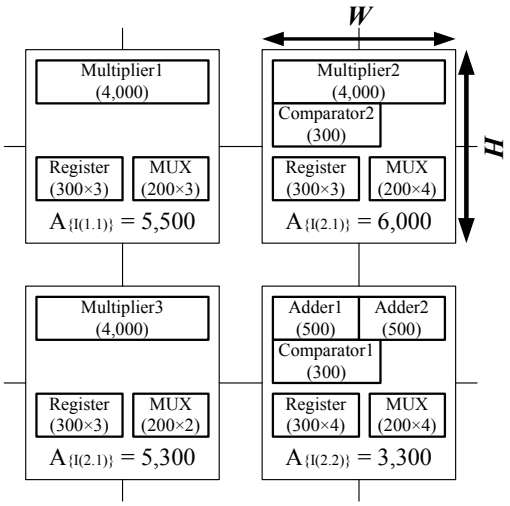


(b) Area cost.

Figure 5.10: Scheduling/binding '+5'' to $\langle S4, M1 \rangle$.



(a) Scheduled/bound re-computational DFG.



(b) Area.

Figure 5.11: An output example of Step(1.2).

Table 5.1: Experimental results.

App. #Island	Algorithm	Island size (μm^2)	Max area (μm^2)	T_{clk} (ns)	#Steps	Latency (ns)	Area Overhead (%)	Time Overhead (%)
DCT	Normal	8,100	6,336	3.0	11	33.0	–	–
3×2	[24]	16,900	15,811	3.0	39	117.0	108.64	254.55
		16,900	15,811	3.5	39	136.5	108.64	313.64
	Ours	12,150	9,014	3.0	23	69.0	50.00	109.09
		12,150	8,327	3.5	20	70.0	50.00	112.12
EWF3	Normal	9,025	8,845	3.0	54	162.0	–	–
2×2	[24]	25,600	24,708	3.0	114	342.0	183.66	111.11
		25,600	25,061	3.5	113	395.5	183.66	144.14
	Ours	13,538	13,138	3.0	86	258.0	50.00	59.26
		13,538	11,989	3.5	70	245.0	50.00	51.23
FIR	Normal	8,100	5,683	3.0	30	90.0	–	–
3×2	[24]	16,900	12,168	3.0	43	129.0	108.64	43.33
		14,400	12,168	3.5	43	150.5	77.78	67.22
	Ours	12,150	12,109	3.0	37	111.0	50.00	23.33

Table 5.2: Comparison of area and performance.

App.	[24]		Ours	
	Island size (μm^2)	Latency (ns)	Island size (μm^2)	Latency (ns)
DCT	16,900 (1.00)	117.0 (1.00)	12,150 (0.72)	69.0 (0.59)
EWF3	25,600 (1.00)	342.0 (1.00)	13,538 (0.53)	258.0 (0.75)
FIR	16,900 (1.00)	129.0 (1.00)	12,150 (0.72)	111.0 (0.86)

5.4 Experimental Results

We have implemented our algorithm in C++. The algorithm has been run on AMD Opteron 2360 SE (2.5 GHz, Quad core) with 16 GB memory and applied to DCT (48 operation nodes), EWF3 (102 operation nodes), and FIR (75 operation nodes). Table 4.1 shows the area and delay of FUs, a register, and a multiplexer used in this experiment, which are all assumed to have 16 bit width and have been synthesized by Synopsys Desgin Compiler under the CMOS 90 nm technology. A controller has been synthesized for each island by Synopsys Desgin Compiler after applying our algorithm.

We have compared our experimental results with those obtained by MCAS [7] and the conventional fault-secure HLS algorithm for RDR architecture [24]. The experimental results are shown in Table 5.1 and Table 5.2. Experimental results demonstrate that our proposed algorithm reduces area by up to 47% and improves

performance by up to 41% compared to [24].

5.5 Conclusion

In this chapter, we have proposed a low-overhead fully redundant fault-secure HLS algorithm for RDR architecture. The experimental results have shown that our algorithm can reduce area by up to 47% and improve performance by up to 41% compared to the conventional fault-secure HLS algorithm for RDR architecture.

Chapter 6

Conclusion

In this dissertation, a thermal-aware HLS algorithm and fault-secure HLS algorithms for distributed-register SoC architectures have been proposed. By focusing on HLS for distributed-register (DR) architectures, improvements in performance and designing cost of SoCs have been realized. In particular, this dissertation has covered RDR architecture which is one of the DR architectures.

In **Chapter 3**, a novel thermal-aware HLS algorithm for RDR architecture has been proposed. Not only interconnect delays but also hot-spots can be estimated at HLS phase by utilizing RDR architecture, which enables us to deal with hot-spots in HLS flow and expects to reduce the designing cost drastically. Our proposed algorithm has balanced the energy consumption among islands in RDR architecture by focusing on the number of operations executed in each island. Balancing the energy consumption can reduce the temperature of hot-spots with no performance degradation. Allocating new functional units has further balanced the energy consumption. Experimental results have demonstrated that our proposed algorithm can reduce the peak temperature inside a chip by up to 15.5% compared to the conventional approach.

In **Chapter 4**, a novel fault-secure HLS algorithm for RDR architecture which partially duplicates operations based on overhead constraints has been proposed. Within a set of performance and area constraints, this algorithm has attempted to maximize reliability which is evaluated by the output probability when a soft error occurs. Our proposed algorithm has adopted a greedy search method by removing duplicated operations. Experimental results have demonstrated that our proposed algorithm can improve reliability by up to 24% without any performance/area overhead compared to the conventional approach.

In **Chapter 5**, a novel low-overhead fault-secure HLS algorithm for RDR architecture which fully duplicates operations has been proposed. In contrast to conventional approaches, our proposed algorithm has adopted an integrative approach

which concurrently performs scheduling, binding, allocation, and register synthesis. Since this approach has monitored the cost of not only functional units but also registers and multiplexers during scheduling/binding, the area/performance overhead can be estimated accurately and then reduced. Experimental results have demonstrated that our proposed algorithm can reduce area by up to 47% and improve performance by up to 41% compared to the conventional approach.

Our proposed algorithms should be extended in the future so that we can target various DR architectures. Using DR architectures enables us to estimate additional design information such as interconnect delay, area, and hot-spot at HLS phase based on the result of module floorplanning. The information can be utilized in a similar manner for constructing novel HLS algorithms achieving high-reliability at low-cost or low-overhead. In constructing these algorithms, we should also capture and then utilize the feature of architectures.

The RTL circuits generated by applying our proposed algorithms have not been implemented on FPGA yet. This is also one of our future works. Our proposed algorithms should be evaluated on their effectiveness for FPGA platforms.

Acknowledgment

First and foremost, I would like to offer my heartfelt thanks to Professor Nozomu Togawa at the department of Computer Science and Engineering of Waseda University for his thorough and enthusiastic guidance on my research. He has taught and advised me about the foundation of my research work, research techniques, writing and presentation skills, and the manners of researcher.

I'm deeply grateful to Professor Masao Yanagisawa at the department of Electronic and Physical Systems of Waseda University, Professor Shinji Kimura at the Graduate School of Information, Production, and Systems of Waseda University, and Professor Keiji Kimura at the department of Computer Science and Engineering of Waseda University for their strong support. Their technical comments have been greatly helpful to the development of my research.

I also thank Professor Youhua Shi at Waseda Institute for Advanced Study for his technical advice to my research work. I have also received technical support from Dr. Shin-ya Abe and Mr. Sho Tanaka for my research work.

I also thank all of the students in Professor Togawa's laboratory and Professor Yanagisawa's laboratory for their cooperation. Especially, I have received helpful advice to my reserch life from Mr. Hiroyuki Akasaka, Mr. Yuta Atobe, Mr. Hiroaki Igarashi, Mr. Yuta Shinoda, Ms. Manami Iwata, Ms. Yoko Oka, and Mr. Tomoharu Hoda.

This dissertation has been supported partially by JSPS KAKENHI Grand-in-Aid for JSPS Fellows.

References

[1] HotSpot 5.0 temperature modeling tool, http://lava.cs.virginia.edu/HotSpot/index.htm.

[2] S. Abe, Y. Shi, M. Yanagisawa, and N. Togawa, "Mh4: multiple-supply-voltages aware high-level synthesis for high-integrated and high-frequency circuits for hdr architectures," *IEICE Electronics Express*, vol. 9, no. 17, pp. 1414–1422, 2012.

[3] S. Abe, M. Yanagisawa, and N. Togawa, "Energy-efficient high-level synthesis for hdr architectures," *IPSJ Trans. on System LSI Design Methodology*, vol. 5, pp. 106–117, 2012.

[4] D. Alexandrescu, L. Anghel, and M. Nicolaidis, "New methods for evaluating the impact of single event transients in vdsm ics," in *Proc. of the 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, 2002, pp. 99–107.

[5] A. Antola, V. Piuri, and M. Sami, "High-level synthesis of data paths with concurrent error detection," in *Proc. of IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, 1998, pp. 292–300.

[6] R. Baumann, "The impact of technology scaling on soft-error rate performance and limits to the efficacy of error correction," in *Proc. of IEDM: International Electron Devices Meeting*, 2002, pp. 329–332.

[7] J. Cong, Y. Fan, G. Han, X. Yang, and Z. Zhang, "Architecture and synthesis for on-chip multi-cycle communication," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 4, pp. 550–564, 2004.

[8] J. Cong, Y. Fan, and J. Xu, "Simultaneous resource binding and interconnection optimization based on a distributed register-file microarchitecture," *ACM Trans. on Design Automation of Electronic Systems*, vol. 14, no. 3, pp. 35:1–35:31, 2009.

[9] Z. P. Gu, Y. Yang, J. Wang, R. P. Dick, and L. Shang, "Taphs: Thermal-aware unified physical-level and high-level synthesis," in *Proc. of the 2006 Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2006, pp. 879–885.

[10] W. Huangy, M. R. Stany, K. Skadronz, K. Sankaranarayananz, S. Ghoshyz, and S. Velusamyz, "Compact thermal modeling for temperature-aware design," in *Proc. of the 41st Design Automation Conference (DAC)*, 2004, pp. 878–883.

[11] J. Jeon, D. Kim, D. Shin, and K. Choi, "High-level synthesis under multicycle interconnect delay," in *Proc. of the 2001 Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2001, pp. 662–667.

[12] D. Kim, J. Jung, S. Lee, J. Jeon, and K. Choi, "Behavior-to-placed rtl synthesis with performance-driven placement," in *Proc. of the 2001 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2001, pp. 320–325.

[13] P. Lim and T. Kim, "Thermal-aware high-level synthesis based on network flow method," in *Proc. of the 2006 Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2006, pp. 124–129.

[14] K. Mohanram and N. A. Touba, "Cost-effective approach for reducing soft error failure rate in logic circuits," in *Proc. of 2003 IEEE International Test Conference (ITC)*, 2003, pp. 893–901.

[15] R. Mukherjee and S. O. Memik, "An integrated approach to thermal management in high-level synthesis," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 14, no. 11, pp. 1165–1174, 2006.

[16] R. Mukherjee, S. O. Memik, and G. Memik, "Peak temperature control and leakage reduction during binding in high level synthesis," in *Proc. of the 2005 International Symposium on Low Power Electronics and Design (ISLPED)*, 2005, pp. 251–256.

[17] R. Mukherjee, S. O. Memik, and G. Memik, "Temperature-aware resource allocation and binding in high-level synthesis," in *Proc. of the 42nd Design Automation Conference (DAC)*, 2005, pp. 196–201.

[18] A. Ohchi, N. Togawa, M. Yanagisawa, and T. Ohtsuki, "High-level synthesis algorithms with floorplaning for distributed/shared-register architectures," in *Proc. of 2008 IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2008, pp. 164–167.

[19] A. Ohchi, N. Togawa, M. Yanagisawa, and T. Ohtsuki, "Performance-driven high-level synthesis with floorplan for gdr architectures and its evaluation," in *Proc. of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2010, pp. 921–924.

[20] F. J. Pollack, "New microarchitecture challenges in the coming generations of cmos process technologies," in *Proc. of the 32nd annual ACM/IEEE international symposium on Microarchitecture (MICRO)*, 1999, p. 2.

[21] R. R. Rao, K. Chopra, D. Blaauw, and D. Sylvester, "An efficient static algorithm for computing the soft error rates of combinational circuits," in *Proc. of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, vol. 1, 2006, pp. 164–169.

[22] P. Shivakumar and M. Kistler, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proc. of International Conference on Dependable System and Networks (DSN)*, 2002, pp. 389–398.

[23] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," in *Proc. of the 30th annual International Symposium on Computer Architecture (ISCA)*, 2003, pp. 2–13.

[24] S. Tanaka, M. Yanagisawa, T. Ohtsuki, and N. Togawa, "A fault-secure high-level synthesis algorithm for rdr architectures," *IPSJ Trans. on System LSI Design Methodology*, vol. 4, pp. 150–165, 2011.

[25] S. Tosun, N. Mansouri, E. Arvas, M. Kandemir, and Y. Xie, "Reliability-centric high-level synthesis," in *Proc. of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, vol. 2, 2005, pp. 1258–1263.

[26] K. Wu and R. Karri, "Fault secure datapath synthesis using hybrid time and hardware redundancy," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 10, pp. 1476–1485, 2004.

[27] J. Yu, Q. Zhou, and J. Bian, "Peak temperature control in thermal-aware behavioral synthesis through allocating the number of resources," in *Proc. of the 2009 Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2009, pp. 85–90.

List of Publications

論文（学術誌原著論文）

⟨1⟩ ◯ **K. Kawamura**, M. Yanagisawa, and N. Togawa, "A thermal-aware high-level synthesis algorithm for RDR architectures through binding and allocation," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 96-A, no. 1, pp. 312–321, Jan. 2013.

⟨2⟩ K. Fujiwara, **K. Kawamura**, S. Abe, M. Yanagisawa, and N. Togawa, "A floorplan-driven high-level synthesis algorithm for multiplexer reduction targeting FPGA designs," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 98-A, no. 7, pp. 1392–1405, July 2015.

国際会議

⟨3⟩ K. Fujiwara, **K. Kawamura**, M. Yanagisawa, and N. Togawa, "Clock skew estimate modeling for FPGA high-level synthesis and its application," in *Proceedings of the IEEE 11th International Conference on ASIC (ASICON)*, Chengdu, China, Nov. 2015.

⟨4⟩ (招待講演) **K. Kawamura**, Y. Hagio, Y. Shi, and N. Togawa, "A floorplan-aware high-level synthesis technique with delay-variation tolerance," in *Proceedings of 2015 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, pp. 122–125, Singapore, Singapore, June 2015.

⟨5⟩ K. Fujiwara, S. Abe, **K. Kawamura**, M. Yanagisawa, and N. Togawa, "A floorplan-aware high-level synthesis algorithm for multiplexer reduction targeting FPGA designs," in *Proceedings of 2014 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pp. 244–247, Ishigaki, Japan, Nov. 2014.

⟨6⟩ (招待講演) **K. Kawamura** and N. Togawa, "Floorplan-driven architecture and high-level synthesis for hot-spot temperature optimization," in *Proceed-*

ings of the 29th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC), pp. 741–744, Phuket, Thailand, July 2014.

⟨7⟩ ◯ **K. Kawamura**, S. Tanaka, M. Yanagisawa, and N. Togawa, "A partial redundant fault-secure high-level synthesis algorithm for RDR architectures," in *Proceedings of the 2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1432–1435, Beiging, China, May 2013.

国内学会

⟨8⟩ (査読あり) 伊東光希, 川村一志, 田宮豊, 柳澤政生, 戸川望, "ローテータベースマルチプレクサネットワークによるフィールドデータ抽出器の構成手法," 情報処理学会 DA シンポジウム 2015 論文集, pp. 29–34, 加賀市, 2015 年 8 月.

⟨9⟩ 川村一志, 阿部晋矢, 史又華, 柳澤政生, 戸川望, "タイミングエラー予測回路による再構成可能デバイス上でのデータ依存最適化回路設計," 信学技報, vol. 114, no. 328, pp. 51–56, 別府市, 2014 年 11 月.

⟨10⟩ 伊東光希, 川村一志, 柳澤政生, 戸川望, 田宮豊, "マルチプレクサ木分割によるフィールドデータ抽出器の構成手法," 信学技報, vol. 114, no. 328, pp. 197–202, 別府市, 2014 年 11 月.

⟨11⟩ 川村一志, 柳澤政生, 戸川望, "フロアプラン統合化アーキテクチャを対象とした低面積指向フォールトセキュア高位合成," 電子情報通信学会 2014 年ソサイエティ大会, pp. 56, 徳島市, 2014 年 9 月.

⟨12⟩ (査読あり) 藤原晃一, 阿部晋矢, 川村一志, 柳澤政生, 戸川望, "フロアプランを考慮したマルチプレクサ削減 FPGA 高位合成手法," 情報処理学会 DA シンポジウム 2014 論文集, pp. 109–114, 下呂市, 2014 年 8 月.

⟨13⟩ 藤原晃一, 阿部晋矢, 川村一志, 柳澤政生, 戸川望, "フロアプランを考慮したマルチプレクサ入力数制限 FPGA 向け高位合成手法," 信学技報, vol. 114, no. 123, pp. 219–224, 札幌市, 2014 年 7 月.

⟨14⟩ 川村一志, 柳澤政生, 戸川望, "信頼性と時間オーバーヘッド間のトレードオフを考慮した面積制約にもとづく RDR アーキテクチャ向けフォールトセキュア高位合成手法," 信学技報, vol. 113, no. 320, pp. 129–134, 鹿児島市, 2013 年 11 月.

⟨15⟩ (査読あり) 川村一志, 柳澤政生, 戸川望, "RDR アーキテクチャを対象とした時間・面積制約にもとづくフォールトセキュア高位合成手法," 第 26 回 回路とシステムワークショップ, pp. 454–459, 淡路市, 2013 年 7 月.

⟨16⟩ 川村一志, 柳澤政生, 戸川望, “RDR アーキテクチャを対象とした時間及び面積オーバーヘッドのないフォールトセキュア高位合成手法,” 信学技報, vol. 113, no. 30, pp. 61–66, 北九州市, 2013 年 5 月.

⟨17⟩ 川村一志, 柳澤政生, 戸川望, “島内消費電力量見積もりにもとづく温度特性を考慮した RDR アーキテクチャ向け高位合成手法,” 信学技報, vol. 112, no. 320, pp. 13–18, 福岡市, 2012 年 11 月.

⟨18⟩ (査読あり) 川村一志, 柳澤政生, 戸川望, “温度特性を考慮した RDR アーキテクチャ向け高位合成手法,” 情報処理学会 DA シンポジウム 2012 論文集, pp. 133–138, 下呂市, 2012 年 8 月.

招待講演

⟨19⟩ 2015 年 8 月 IEEE SSCS Japan Chapter VDEC デザイナーズフォーラム 2015 Ph.D 企画セッション パネリスト.

業績賞等

⟨20⟩ 2015 年 8 月 アルゴリズムデザインコンテスト優秀賞.

⟨21⟩ 2014 年 11 月 情報処理学会 SLDM 優秀発表学生賞.

⟨22⟩ 2014 年 8 月 アルゴリズムデザインコンテスト特別賞.

⟨23⟩ 2013 年 11 月 情報処理学会 SLDM 優秀発表学生賞.

⟨24⟩ 2012 年 11 月 情報処理学会 SLDM 優秀発表学生賞.

日本学術振興会 科学研究費補助金

⟨25⟩ 日本学術振興会特別研究員奨励費, “配線遅延の温度依存性を考慮し回路性能を最適化する高位 LSI 設計技術,” 2015–2016 年度, 総額 190 万円 (2015 年度:100 万円, 2016 年度:90 万円).