Speculative Execution in Distributed Controllers for High-Level Synthesis

Miho Shimizu*
Nagisa Ishiura
Sayuri Ota
Wakako Nakano
Kwansei Gakuin University
School of Science and Technology
Sanda, Hyogo, Japan

ABSTRACT

This paper proposes a method of incorporating speculative execution into distributed control which enables efficient dynamic scheduling. In the presence of variable latency units, the static scheduling scheme in conventional high-level synthesis causes wasteful waits. Distributed control enables dynamic scheduling which adjust the execution timing of the operations dynamically. In this paper, we attempt to further enhance speed performance by introducing speculative execution based on branch prediction into distributed control. Experimental results on two examples showed that the execution cycles were reduced by 11.1% to 21.9% when the prediction hit rate was 75%.

CCS CONCEPTS

• **Hardware → Hardware-software codesign**; • **Computer systems organization → Embedded hardware**;

KEYWORDS

High-level synthesis, variable latency units, distributed control, speculative execution, dynamic scheduling

ACM Reference Format:
Miho Shimizu, Nagisa Ishiura, Sayuri Ota, and Wakako Nakano. 2017. Speculative Execution in Distributed Controllers for High-Level Synthesis. In *Proceedings of RSP'17, Seoul, Republic of Korea, October 15–20, 2017,* 7 pages.
https://doi.org/10.1145/3130265.3130319

1 INTRODUCTION

With recent progress in the integrated circuit technology, the scale and the complexity of the hardware implemented in a chip are growing rapidly. While such systems need enormous design efforts, there is a strong demand to reduce time to market. High-level synthesis is one of the promising means to prototype hardware design from existing software assets on which many researches have been conducted [1].

In conventional high-level synthesis methods, operations are scheduled assuming that functional units take the same number of clock cycles for the same operations. The scheduling is determined statically and does not change during run time. However, in actual datapaths, some functional units exhibit different latencies for the same operation depending on operands or their environment. Although the traditional scheduling is still valid if the maximum latencies are assumed for such units, wasteful waits occur when the units take less latencies than those of the worst case.

One way to solve this problem is to adjust operation scheduling dynamically [2]. However, the conventional datapath control using a single state machine ends up with impractical circuit size because an enormous amount of states are necessary to express all the combination of delay variations of the functional units. As a promising alternative, distributed control has been recently proposed [3, 4] which controls a datapath with multiple finite state machines.

Different schemes for distributed controllers have been proposed by Del Barrio [3] and Pilato [4]. All these methods, however, deal with a case where computation is expressed with a single DFG. This may be appropriate for accelerating kernels in DSP applications, but not for prototyping control centric systems.

In order to apply the distributed control to larger systems, an extension of the Del Barrio's distributed control to handle multiple DFGs was proposed in [5]. Rather than simply patchworking the state transition graphs for the DFGs, it enabled dynamic operation motion across multiple DFGs, which realized efficient execution like loop scheduling and trace scheduling. However, the effect of dynamic code motion was limited when branch conditions were computed at the end of the DFGs, for early units must idle-wait until the branch target was fixed.

To address this issue, we extend the method of [5] with speculative execution based on branch prediction. By this method, the idle units may compute operations in predicted next DFGs even if branch targets have not been fixed. Experimental results on two examples shows that the execution cycles were reduced by 11.1% to 21.9% when the prediction hit rate was 75%.

*Currently with Japan System Techniques Co., Ltd. (JAST), Osaka, Japan

RSP'17, October 15–20, 2017, Seoul, Republic of Korea

ACM ISBN 978-1-4503-5418-9/17/10…$15.00
https://doi.org/10.1145/3130265.3130319

2 VARIABLE LATENCY UNITS AND DISTRIBUTED CONTROL

2.1 Variable latency units

Functional units in datapaths may exhibit different latencies for the same operation, depending on operand values, states of the units, and environment factors. For example, shift/add-based multipliers and dividers can omit part of the computation when some part of multiplicand or intermediate remainder becomes zero. Memory accesses may take different cycles depending on address histories.

Let us consider executing a DFG in Figure 1 (a) with an adder A which takes 1 cycle and a multiplier M which takes either 1 or 2 cycles. In the conventional high-level synthesis, the operations are scheduled as (b) assuming operations 2 and 4 may take 2 cycles. Even when operation 2 completes in 1 cycle, scheduling is unchanged as shown in (c), where the second cycle is wasteful.

Toda [2] proposed dynamic scheduling based on the completion signals from functional units which enables scheduling as shown in Figure 1 (d). However, the controller for this dynamic scheduling needed huge amount of states, so the resulting circuit might be impractically large.

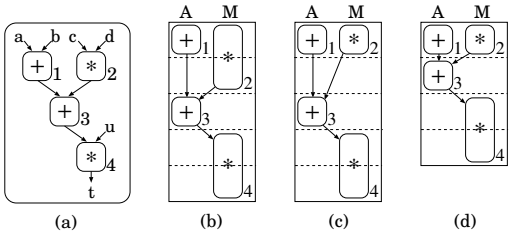


Figure 1: DFG with variable latency operations.

2.2 Distributed control

As an approach to realizing dynamic scheduling with reasonable circuit size, distributed control has recently been proposed which controls functional units using multiple finite state machines (FSMs).

In the Del Barrio's method [3], an FSM is assigned to each function unit in a datapath. The function unit to execute each operation is determined beforehand. The order of the operations executed by each unit is also fixed and the controller dynamically decides the timing to execute the operations. The Pilato's method [4] controls the datapath by assigning a state variable to each operation. As Del Barrio's method, the function unit to execute each operation is determined beforehand but the order as well as the timing of execution of the operations is determined dynamically.

However, these methods only discusses the case where computation is expressed by a single DFG. They have not addressed the issue of handling a CDFG consisting of multiple DFGs, which is essential in synthesizing hardware from specification expressed in programming languages like C.

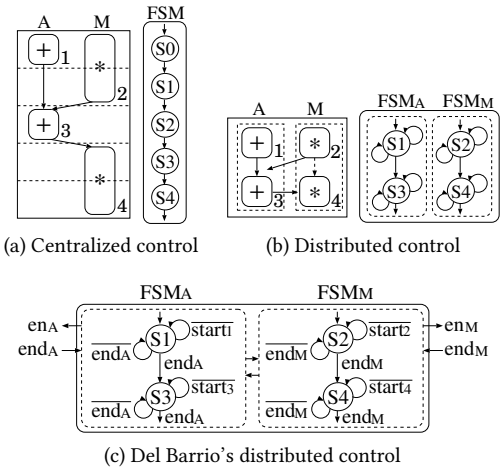


Figure 2: Centralized and distributed control.

2.3 Del Barrio's distributed control

In the conventional control method, a whole datapath is controlled by a single FSM (finite state machine), as shown in Figure 2 (a). On the other hand, in Del Barrio's distributed control, separate FSM_A and FSM_M control function units A and M, respectively, as in (b). This allows each unit to choose execution timing independently.

Figure 2 (c) is the details of the FSMs in (b). The formulation in this paper is slightly different from the original one in [3], but essentially the same. One state S_i is assigned to an operation i in the DFG, in which S_i controls the execution of i. At S_i, the FSM waits for $start_i = 1$ where $start_i$ becomes 1 if all the operations on which i depends are finished. In the case of Figure 2,

$$start_1 = 1,\ start_2 = 1,$$
$$start_3 = (s_2 \wedge end_M) \vee Done_2,$$
$$start_4 = (s_3 \wedge end_A) \vee Done_3,$$

where s_i means that the FSM is in state S_i, and end_u is the completion signal from unit u. $Done_i$ stands for completion of operation i, whose initial value is 0 and is updated as follows.

if $s_1 \wedge end_A$ **then** $Done_1 = 1$,
if $s_2 \wedge end_M$ **then** $Done_2 = 1$,
if $s_3 \wedge end_A$ **then** $Done_3 = 1$,
if $s_4 \wedge end_M$ **then** $Done_4 = 1$.

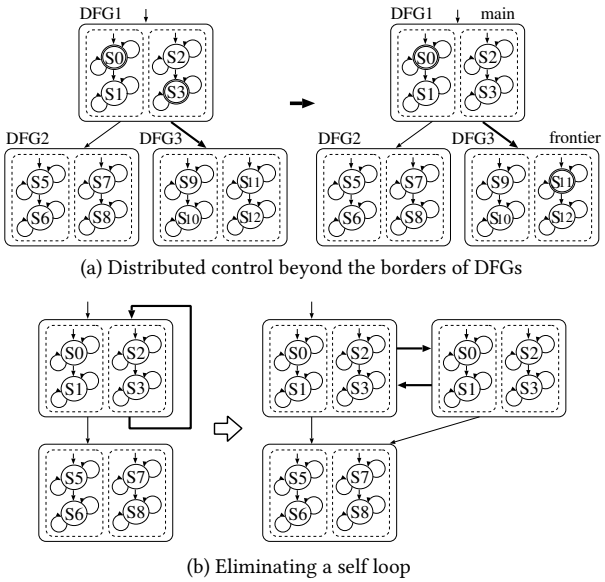
When state is S_i and $start_i = 1$, FSM_u sets the enable signal en_u of unit u.

$$en_A = ((s_1 \wedge start_1) \vee (s_3 \wedge start_3)) \wedge \overline{Exe_A},$$
$$en_M = ((s_2 \wedge start_2) \vee (s_4 \wedge start_4)) \wedge \overline{Exe_M},$$

where Exe_u means that unit u is in operation, which is defined as follows.

if en_A **then** $Exe_A = 1$ **else if** end_A **then** $Exe_A = 0$,
if en_M **then** $Exe_M = 1$ **else if** end_M **then** $Exe_M = 0$.

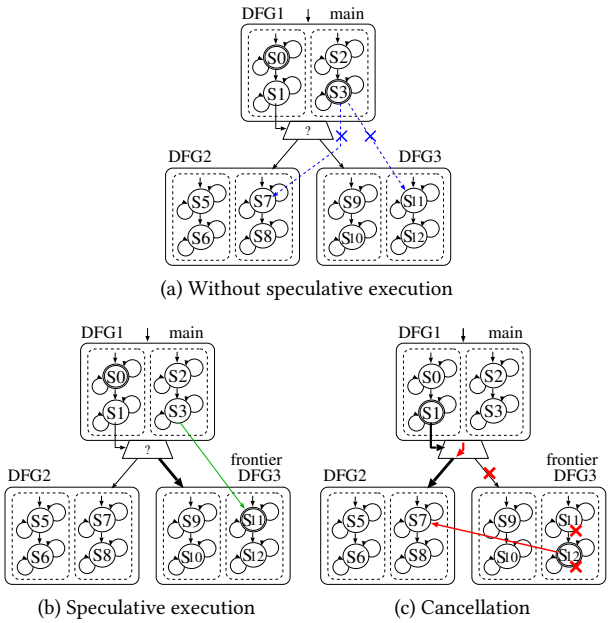
Del Barrio's method in [3] covers a loop with a single DFG, where operations in the next iteration may be executed without waiting for the completion of a certain iteration. However, it does not handle a CDFG consisting of multiple DFGs; it does not cover conditional jumps, for example.



(a) Distributed control beyond the borders of DFGs

(b) Eliminating a self loop

Figure 3: Extending distributed control to multiple DFGs.



(a) Without speculative execution

(b) Speculative execution

(c) Cancellation

Figure 4: Speculative execution.

2.4 Distributed control beyond the borders of dataflow graphs

Shimizu [5] proposed an extension of the Del Barrio's distributed control to handle multiple DFGs, where dynamic motion of operations across multiple DFGs was achieved.

In the case of the centralized FSM, the control is transferred to one of the next DFGs after all the operations in the current DFG are finished. However, in the presence of variable latency operations, not all the units finish their task in an DFG simultaneously, so some of them must wait for each other at the end of the DFG. To overcome this inefficiency, the method of [5] enabled operations in the next DFG to start execution before concluding the current DFG. For example, in Figure 3 (a), where two units are controlled by two controllers, the operations corresponding to $S0$ and $S3$ are finished, then the control of the second FSM can be transferred from $S3$ to $S11$, without waiting for the completion of $S1$, if DFG3 is known to be the next DFG of DFG1. In [5] the extension of the distributed control was formulated under the restriction that only one DFG ahead of the current DFG may be executed, i.e. at most two DFGs may be executed at the same time. A self loop is eliminated by duplicating the DFG, as shown in Figure 3 (b).

3 SPECULATIVE EXECUTION IN DISTRIBUTED CONTROL

3.1 Overview

In the control method in [5], however, dynamic scheduling across DFGs works only when the next DFG is fixed. Suppose $S1$ in DFG1 in Figure 4 (a) determines the target of the conditional branch. Although the second (right) unit has finished the computation at $S3$, it cannot proceed to the next state until $S1$ decides the next DFG. Since it is often the case that conditional operations to decide branch directions are computed at the very end of the DFGs, conservative synchronization limits the effect of dynamic scheduling.

To address issue, we propose to extend the distributed control of [5] with speculative execution. At the situation where next DFG to be executed is not known, one of them is chosen based on branch prediction. As shown in Figure 4 (b), the second unit proceed from $S3$ to $S11$ without waiting for the completion of $S1$, if the branch from DFG1 to DFG3 is predicted. If the prediction hits, there may be reduction in execution cycles. If the prediction misses, the intermediate results of the speculative execution are cancelled and the state is forced to the head of the correct DFG, as shown in Figure 4 (c). This causes no loss in execution cycles on prediction misses, and we have a chance of speeding-up circuits on prediction hits.

To avoid complication, we place the same restriction as in [5] that only one DFG ahead of the current DFG may be executed. We call a DFG a *main* DFG if some FSMs are executing operations in the DFG and the other FSMs have completed execution for the DFG. We call a DFG a *frontier* DFG if it is a successor DFG of the main DFG and operations in the DFG are being executed. In Figure 4 (b), DFG1 is a main DFG and DFG3 is a frontier DFG. The frontier DFG does not always exist; there is no frontier DFG in Figure 4 (a).

Cancellation of execution is done based on a register save/restore policy. Just before speculative execution starts, the contents of the registers that may be written during speculative execution are copied to the save registers. In the case of prediction misses, the saved values are written back to the registers when the FSMs transition to the right states.

3.2 Formulation

The formulation of the distributed control method in this section is basically based on that of [5], but it is refined partly to make it

easier to incorporate speculative execution, and partly to enhance readability.

Let U be the set of units in the datapath. e_u is the completion signal of $u \in U$ at the current cycle. In our scheme, each $u \in U$ is controlled by an FSM, denoted as F_u, where each state of F_u controls the execution of an operation. Let σ_u be the current state of F_u. Let D be the set of DFGs in the CDFG. Let $S_{u,d} = \{s_0^{u,d}, s_1^{u,d}, \cdots, s_f^{u,d}\}$ be the set of the states for u in DFG d, where σ_u transitions $s_0^{u,d}$ through $s_f^{u,d}$ in this order. Namely, $s_0^{u,d}$ and $s_f^{u,d}$ are the initial and the final states in the DFG d for unit u. For design convenience, we assume that $S_{u,d} \neq \phi$. If there is no operation for u in DFG d, we insert a dummy state.

3.2.1 End and ready signals. For $d \in D$ and $s \in S_{u,d}$, $r(s)$, $e(s)$, $E(s)$, and $e(d)$ are defined as follows. Intuitively, $r(s)$ means that the operation at s is ready for execution. $e(s)$ means that the execution of the operation at s ends at the current cycle, $E(s)$ means that the execution of the operation at s has been completed before the current cycle, and $e(d)$ means that execution of all the operations in DFG d finishes at the current cycle. Let P_s be the set of the states (operations) that (the operation of) s depends on. $E_0(s)$ and $E'(s)$ are the initial value and the next value (the value at the next cycle) of $E(s)$, respectively; $E(s)$ is set when $e(s) = 1$ and reset at the end of the DFG execution.

$$r(s) = \bigwedge_{s_p \in P_s} E(s_p), \tag{1}$$

$$e(s) = (\sigma_u = s) \wedge r(s) \wedge e_u, \tag{2}$$

$$e(d) = \bigwedge_{u \in U} (e(s_f^{u,d}) \vee E(s_f^{u,d})), \tag{3}$$

$$E_0(s) = 0, \tag{4}$$

$$E'(s) = \textbf{if } e(d) \textbf{ then } 0 \textbf{ else } e(s) \vee E(s). \tag{5}$$

The operation of s starts execution when $(\sigma_u = s) \wedge r(s)$.

3.2.2 State transition. When $\sigma_u = s_i^{u,d}$, where $0 \leq i < f$, the next state σ'_u within a DFG d is defined as follows.

$$\sigma'_u = \textbf{if } r(s_i^{u,d}) \wedge e(s_i^{u,d}) \textbf{ then } s_{i+1}^{u,d} \textbf{ else } s_i^{u,d}. \tag{6}$$

State transition across DFGs is a little complicated. The state machine F_u can transition from the last state of DFG d to the first state of a next DFG d_1 if and only if the two conditions are met.

(1) Transition from d to d_1 is fixed.
(2) d_1 is executed either as a main DFG or as a frontier DFG in the next cycle.

The first condition is formulated using $\delta(d, d_1)$ and $\Delta(d, d_1)$ for two DFGs d and d_1. $\delta(d, d_1)$ means that transition from d to d_1 is triggered at the current cycle, and $\Delta(d, d_1)$ mean that transition from d to d_1 has been fixed before the current cycle. If d_1 is the unique successor to d, then $\delta(d, d_1) = 1$ and $\Delta(d, d_1) = 1$. Otherwise, $\delta(d, d_1)$ and $\Delta(d, d_1)$ are defined as follows, assuming that the transition from d to d_1 is determined by the output o_u of unit u at state $s_c \in S_{u,d}$. $\Delta_0(d, d_1)$ and $\Delta'(d, d_1)$ are the initial and the next values of $\Delta(d, d_1)$, respectively; $\Delta(d, d_1)$ is set when $\delta(d, d_1) = 1$ and is reset when the execution of d completes.

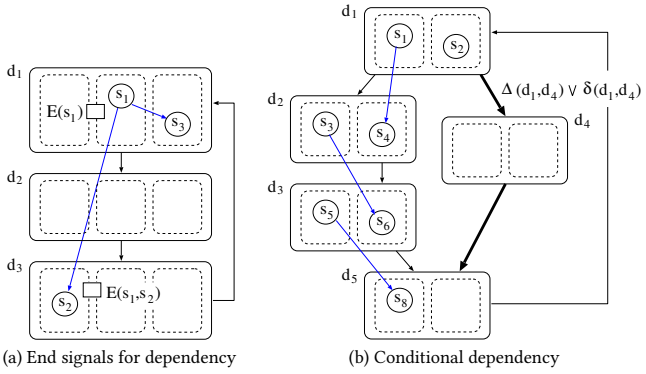


Figure 5: Inter-DFG dependency.

$$\delta(d, d_1) = (\sigma_u = s_c) \wedge e_u \wedge o_u, \tag{7}$$

$$\Delta_0(d, d_1) = 0, \tag{8}$$

$$\Delta'(d, d_1) = \textbf{if } e(d) \textbf{ then } 0 \textbf{ else } \delta(d, d_1) \vee \Delta(d, d_1). \tag{9}$$

For the second condition, we define $M(d)$ and $m(d)$ for a DFG d. $M(d)$ intuitively means that DFG d is being executed as a main DFG in the current cycle, and $m(d)$ that d will be executed as a main DFG in the next cycle. $M_0(d)$ and $M'(d)$ are the initial and the next values of $M(d)$.

$$m(d) = \bigvee_{d_p \in P_d} (e(d_p) \wedge \delta(d_p, d)), \tag{10}$$

$$M_0(d) = \textbf{if } d \text{ is the starting DFG } \textbf{then } 1 \textbf{ else } 0, \tag{11}$$

$$M'(d) = \textbf{if } e(d) \textbf{ then } 0 \textbf{ else } M(d) \vee m(d). \tag{12}$$

Let DFGs $d_1, d_2, \cdots, d_k$ be successors of DFG d. They may be executed as a main or a frontier DFG in the next cycle if $M(d) \vee m(d)$. Thus, when $\sigma_u = s_f^{u,d}$, the next state σ'_u is written as follows.

$$\begin{aligned}\sigma'_u = \ & \textbf{if } r(s_f^{u,d}) \wedge e(s_f^{u,d}) \wedge (e(d) \vee m(d)) \textbf{ then} \\ & \quad \textbf{if } \delta(d, d_1) \vee \Delta(d, d_1) \textbf{ then } s_0^{u,d_1} \\ & \quad \textbf{else if } \delta(d, d_2) \vee \Delta(d, d_2) \textbf{ then } s_0^{u,d_2} \\ & \quad \cdots \\ & \quad \textbf{else if } \delta(d, d_k) \vee \Delta(d, d_k) \textbf{ then } s_0^{u,d_k} \\ & \quad \textbf{else } s_f^{u,d} \\ & \textbf{else } s_f^{u,d}.\end{aligned} \tag{13}$$

3.2.3 Inter-DFG dependency. For inter-DFG dependency, such as $s_1 \rightarrow s_2$ in Figure 5 (a), the end signals $E(s)$ do not work well. For example, in Figure 5 (a), if s_2 waits for $E(s_1)$ signal which will be reset as soon as DFG1 finishes, s_2 can never start execution. If reset of $E(s_1)$ is postponed until the end of s_2, then s_3 in the next iteration starts without waiting for the completion of s_1 in the next iteration.

To solve this problem, we introduce $E(s_1, s_2)$ for every inter-DFG dependency from s_1 to s_2. It is set when s_1 is finished and reset when s_2 is reset.

Note that s_1 and s_2 in $E(s_1, s_2)$ are not always executed. For example, in Figure 5 (b), suppose branch from DFG d_1 to DFG d_4 is taken. If s_8 would waits for $E(s_5, s_8)$, s_8 could never start execution, because s_5 is not executed and $E(s_5, s_8)$ will never be set. Moreover, $E(s_1, s_4)$ would not reset in this iteration. Then if branch from d_1 to d_2 might be taken in the next iteration, s_4 would start execution without waiting for the completion of s_1.

This kind of confusion is resolved by setting/resetting the end signal $E(s_1, s_2)$ when directions of branches are determined.

- If it is known that the DFG of s_1 will never be executed in the current iteration, then $E(s_1, s_2)$ is set.
- If it is known that the DFG of s_2 will never be executed in the current iteration, then $E(s_1, s_2)$ is reset.
- If it is known that both the DFGs of s_1 and of s_2 will never be executed in the current iteration, then the initial value is set to $E(s_1, s_2)$.

For example, in Figure 5 (b), branch from d_1 to d_4 is determined when $\Delta(d_1, d_4) \vee \delta(d_1, d_4)$ becomes true, and at this point, it is also determined that d_2 and d_3 will not be executed in this iteration. Thus, $E(s_1, s_4) = 0$, $E(s_3, s_6) = 0$, and $E(s_5, s_8) = 1$ are enforced.

The initial and next values of $E(s_1, s_2)$, denoted as $E_0(s_1, s_2)$ and $E'(s_1, s_2)$, respectively, are defined as follows, where $\gamma(\overline{s_1})$ and $\gamma(\overline{s_2})$ are the conditions in which s_1 and s_2, respectively, will not be executed in the current iteration.

$$\begin{aligned} E_0(s_1, s_2) &= \textbf{if } s_1 \rightarrow s_2 \text{ is loop carried dependency} \\ &\quad \textbf{then } 1 \textbf{ else } 0, \end{aligned} \tag{14}$$

$$\begin{aligned} E'(s_1, s_2) &= \textbf{if } \gamma(\overline{s_1}) \wedge \gamma(\overline{s_2}) \textbf{ then } E_0(s_1, s_2) \\ &\quad \textbf{else if } e(s_2) \vee \gamma(\overline{s_2}) \textbf{ then } 0 \\ &\quad \textbf{else if } e(s_1) \vee \gamma(\overline{s_1}) \textbf{ then } 1 \\ &\quad \textbf{else } E(s_1, s_2). \end{aligned} \tag{15}$$

With the extended end signals, equation (1) defining ready signal $r(s)$ is revised as follows, where P_s is the set of the states that s depends on in the same DFG as s , and X_s is the set of the states that s depends on in the other DFGs.

$$r(s) = \Big(\bigwedge_{t \in P_s} E(t)\Big) \wedge \Big(\bigwedge_{t \in X_s} E(t, s)\Big). \tag{16}$$

3.2.4 Speculative execution. We assume $B(d, d_1)$ denotes that branch from d to d_1 is predicted. The prediction may be either static or dynamic.

Branch prediction changes the state transition in the following way: 1) it allows transition to the state in the predicted DFG, and 2) it forces the transition to the correct states. This is incorporated into the formulation by revising the equations (6) and (13).

Let $d_1, d_2, \cdots, d_k$ be successor DFGs of DFG d and let branch from d to d_1 be predicted (namely, $B(d, d_1) = 1$ is assumed). Then state transition is modified as follows.

- State transition from the last state of d
  State transition regarding branch prediction occurs only when none of the other branches occur. Thus we only have to correct the last **else** clause of equation (13).

$$\begin{aligned} \sigma'_u = \ & \textbf{if } r(s_f^{u,d}) \wedge e(s_f^{u,d}) \wedge (e(d) \vee m(d)) \textbf{ then} \\ & \quad \textbf{if } \delta(d, d_1) \vee \Delta(d, d_1) \textbf{ then } s_0^{u,d_1} \\ & \quad \cdots \\ & \quad \textbf{else if } \delta(d, d_k) \vee \Delta(d, d_k) \textbf{ then } s_0^{u,d_k} \\ & \quad \underline{\textbf{else if } B(d, d_1) \textbf{ then } s_0^{u,d_1}} \\ & \quad \textbf{else } s_f^{u,d} \\ & \textbf{else } s_f^{u,d}. \end{aligned} \tag{17}$$

- State transition within d_1
  Equation (6) must be corrected so that branch prediction misses are handled.

$$\begin{aligned} \sigma'_u = \ & \underline{\textbf{if } \delta(d, d_2) \vee \Delta(d, d_2) \textbf{ then } s_0^{u,d_2}} \\ & \quad \underline{\cdots} \\ & \quad \underline{\textbf{else if } \delta(d, d_k) \vee \Delta(d, d_k) \textbf{ then } s_0^{u,d_k}} \\ & \quad \underline{\textbf{else}} \textbf{ if } r(s_i^{u,d_1}) \wedge e(s_i^{u,d_1}) \textbf{ then } s_{i+1}^{u,d_1} \textbf{ else } s_i^{u,d_1} \end{aligned} \tag{18}$$

- No change on the state transition within $d_2, d_3, \cdots, d_k$.

As for the register values, save must be done when $B(d, d_1)$ holds in equation (17), restore must be done when either of $\delta(d, d_2) \vee \Delta(d, d_2)$ through $\delta(d, d_k) \vee \Delta(d, d_k)$ holds in (18). At the same time as the register restore, the end signals for inter-DFG dependency must be adjusted according to the equation (15).

4 EXPERIMENTAL RESULTS

RTL circuits have been designed in Verilog HDL for two benchmark circuits, in order to compare centralized control, distributed control without speculative execution [5], and proposed distributed control with speculative execution. In this experiment, we assume that multiplication takes 1 or 2 cycles depending on operands, and all the other operations take 1 cycle. We assume static branch prediction both in centralized and distributed control. In the both benchmarks, conditional operations to decide the branch directions can be scheduled only in the last steps of the DFGs.

(1) bicubic: The CDFG in Figure 6 (a) is executed with two ALUs (A1 and A2) and two multipliers (M1 and M2). Figure 6 (b) is an example of binding and scheduling for distributed control (determined manually). DFG0 is duplicated to remove the self loop. $d0$, $d0c$, $d1$, and $d2$ are dummy states. Figure 6 (c) is a result of loop and trace scheduling for centralized scheduling.

(2) m-lerp: The CDFG in Figure 7 (a) is executed with a comparator (EQ), two ALUs (A1 and A2), and two multipliers (M1 and M2). The binding and scheduling examples for distributed and centralized control are shown in Figure 7 (b) and (c), respectively.

TABLE 1 (a) summarizes the comparison of the execution cycles for the three control schemes. The computation shown in CDFGs in Figure 6 (b) and Figure 7 (b) were iterated for 128 times. r is the probability where the multiplication takes two cycles; $r = 1.0$ means all the multiplications took 2 cycles, while $r = 0.0$ means all the multications took 1 cycle.

Table 1: Experimental result.

(a) Execution cycles

	pred. hit rate	CC	$r = 1.0$ DC	$r = 1.0$ **SE**	$r = 0.5$ DC	$r = 0.5$ **SE**	$r = 0.0$ DC	$r = 0.0$ **SE**
(1) bicubic	87.5%	1,026	1,215	1,087 (−10.5%)	1,097	922 (−16.0%)	982	750 (−23.6%)
	75.0%	959	1,138	1,012 (−11.1%)	1,039	876 (−15.7%)	933	729 (−21.9%)
	50.0%	775	919	825 (−10.2%)	854	741 (−13.2%)	793	667 (−15.9%)
	33.3%	695	831	754 (−9.3%)	786	699 (−11.1%)	737	643 (−12.8%)
	0.0%	516	580	580 (±0.0%)	580	580 (±0.0%)	580	580 (±0.0%)
(2) m-lerp	87.5%	739	966	782 (−19.0%)	906	718 (−20.8%)	846	651 (−23.0%)
	75.0%	726	933	781 (−16.3%)	860	699 (−18.7%)	819	650 (−20.6%)
	50.0%	656	872	774 (−11.2%)	823	714 (−13.2%)	767	649 (−15.4%)
	33.3%	647	812	750 (−7.6%)	754	682 (−9.5%)	717	638 (−11.0%)
	0.0%	609	695	695 (±0.0%)	649	649 (±0.0%)	617	617 (±0.0%)

CC: centralized control (trace and loop scheduling), DC: distributed control [5], **SE**: distributed control with speculative execution (**proposed**)
multiplication $1 \sim 2$ cycles, r: probability of 2 cycle multiplication

(b) Synthesis result.

	CC			DC			**SE**		
	FFs	LUTs	delay [ns]	FFs	LUTs	delay [ns]	FFs	LUTs	delay [ns]
(1) bicubic	142	268	8.398	261	585	8.534	261	603	8.643
(2) m-lerp	205	375	8.061	293	696	7.595	296	721	8.205

synthesizer: Xilinx Vivado (2016.4), target: Xilinx Artix-7 (xc7a100tcsg324-3)

Column "pred. hit rate" shows the hit rate of branch prediction (the input data were generated randomly). Column "CC" shows the results of the classical centralized control scheme, with the scheduling and biding in Figure 6 (c) and Figure 7 (c) where all the multiply operations were assumed to take 2 cycles. Columns "DC" and "**SE**" are both results of distributed control, where the former is without speculative execution [5] and the latter is with speculative execution (the proposed method). They both are based on the same scheduling and binding in Figure 6 (b) and Figure 7 (b)

Comparison between DC and SE shows that the proposed method needed fewer execution cycles than the previous method. Naturally, the reduction rate is higher when the prediction hit rate is higher and r is lower. When the prediction hit rate was 75%, the execution cycles were reduced by 11.1% to 21.9%. As compared with CC, the proposed method took little more cycles when $r = 1.0$. This is because of the limitation of our method that only 2 DFGs can be executed simultaneously. However, when r is smaller, the execution cycles were reduced significantly due to the effect of dynamic scheduling.

TABLE 1 (b) summarizes the result of logic synthesis. The logic synthesizer was Xilinx Vivado (2016.4) and the target was Artix-7 (xc7a100tcsg324-3). Subcolumns "FFs," "LUTs,", and "delay," show the number of the flip-flops, the number of the look-up tables, and the critical path delay of each circuit. Introduction of speculative execution increases circuit size in terms of LUT count by 3.3% on average, though the LUT count is as much as 2 times larger than that of the centralized control. However, the critical path delay is almost the same. The critical path delay is increased by speculative execution by 4.6% on average, but it is only 2.4% larger than that of the centralized control.

The cancellation of speculative execution based on the formulation in 3.2.4 worked only for the two benchmarks with manual design. However, in order to handle more general cases, especially with complicated inter-DFG dependency, we might add some limitations or further refine the formulation.

5 CONCLUSION

This paper has proposed a method of introducing speculative execution in the distributed control beyond the border of DFGs. This contributes to enhance speed performance by increasing the opportunity of dynamic operation motion across DFGs.

Currently, the circuits are designed by hand as well as the scheduling and biding are determined manually. We are now working on a scheduling and biding method suitable for our distributed control and automatic circuit generation framework.

ACKNOWLEDGMENTS

Authors would like to express their appreciation to Dr. Hiroyuki Kanbara of ASTEM/RI, Prof. Hiroyuki Tomiyama of Ritsumeikan University, and Mr. Takayuki Nakatani (formerly with Ritsumeikan University) for their valuable comments. We would also like to thank to the members of Ishiura Lab. of Kwansei Gakuin University for their cooperation and discussion. This work is partly supported by JSPS KAKENHI under Grant No. 16K00088.

REFERENCES

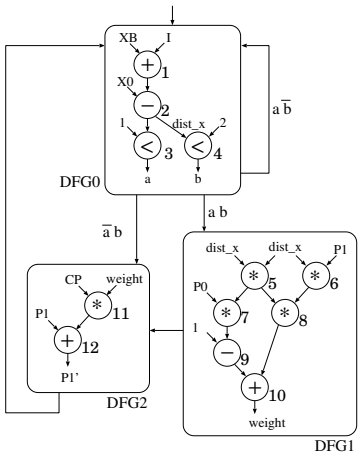
[1] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin. 1992. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers.

[2] Yuki Toda, Nagisa Ishiura, and Kousuke Sone. 2009. Static scheduling of dynamic execution for high-level synthesis. In *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2009)*. (March 2009), 107–112.

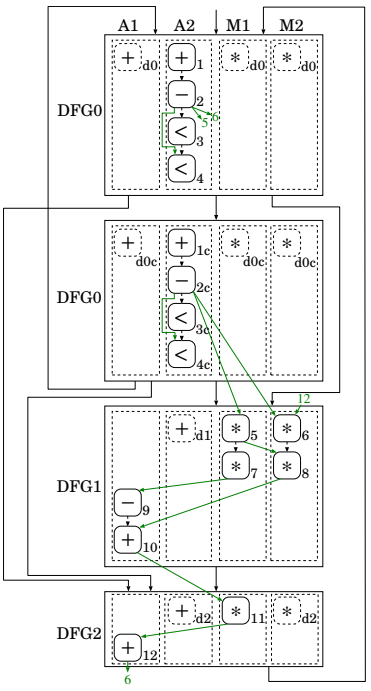
[3] Alberto A. Del Barrio, Seda Ogrenci Memik, María C. Molina, José M. Mendías, and Román Hermida. 2011. A distributed controller for managing speculative functional units in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 30, 3 (March 2011), 350–363. DOI:http://doi.org/10.1109/TCAD.2010.2089565

[4] Christian Pilato, Vito Giovanni Castellana, Silvia Lovergine, and Fabrizio Ferrandi. 2011. A runtime adaptive controller for supporting hardware components with variable latency. In *Proceedings of 2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2011)*. (June 2011), 153–160. DOI:http://doi.org/10.1109/AHS.2011.5963930

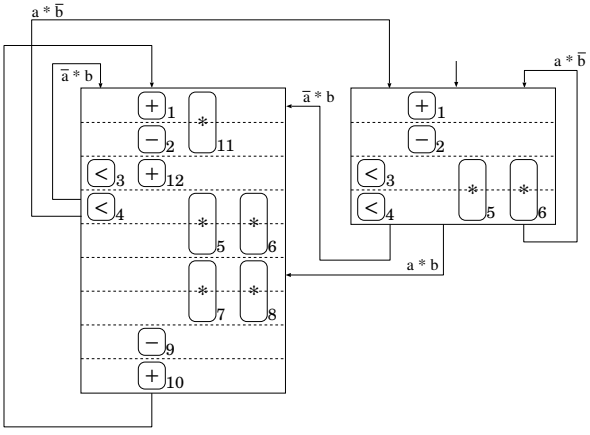
[5] Miho Shimizu and Nagisa Ishiura. 2016. Extending Distributed Control for High-Level Synthesis beyond Borders of Basic Blocks. In *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2016)*. (October 2016), 172–177.



(a) Given CDFG

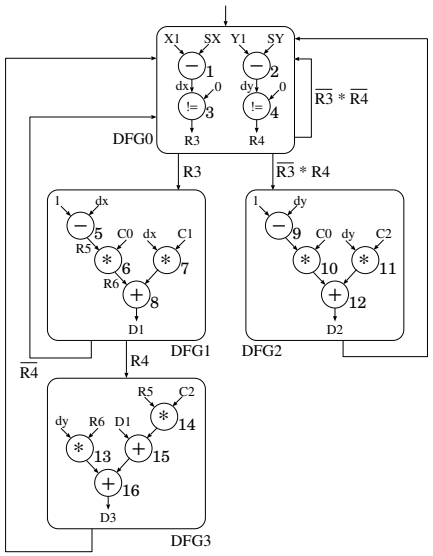


(b) Scheduling and binding for the proposed method.

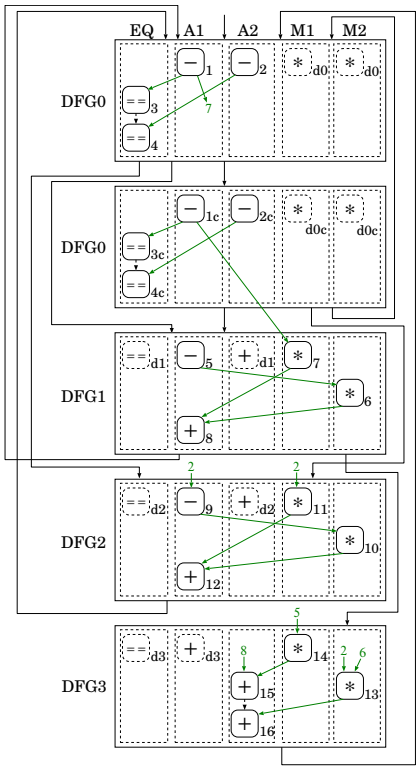


(c) Loop and trace scheduling (centralized control).

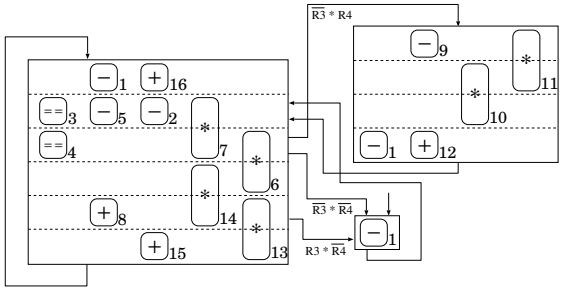
Figure 6: Benchmark bicubic.



(a) Given CDFG



(b) Scheduling and binding for the proposed method.



(c) Loop and trace scheduling (centralized control).

Figure 7: Benchmark m-lerp.