Extending Distributed Control for High-Level Synthesis beyond Borders of Basic Blocks

Miho SHIMIZU Nagisa ISHIURA

School of Science and Technology, Kwansei Gakuin University
2-1 Gakuen, Sanda, Hyogo, 669–1337, Japan

Abstract— This paper proposes an extension of distributed control, which enables efficient run-time scheduling of variable latency operations, to multiple dataflow graphs. Conventional high-level synthesis methods determine the execution schedule of operations statically assuming that their latencies are fixed. However, actual circuits contain so-called variable latency units whose execution cycles may vary depending on various run-time factors. Although Del Barrio, Pilato, and Yamashita have proposed distributed control methods which can efficiently control circuits with such units, they only handles a single dataflow graphs. Our method extends the Del Barrio's distributed control to handle multiple dataflow graphs. It enables dynamic scheduling of operations beyond the boundaries of basic blocks which results in fewer execution cycles than those by conventional centralized control with loop scheduling and trace scheduling.

I. Introduction

Recent progress in semiconductor technology has enabled development of larger and more sophisticated systems on VLSI chips. While such systems needs enormous design efforts, there is a strong demand to reduce time to market. High-level synthesis is one of the promising measures to expedite hardware design on which many researches have been conducted [1].

In conventional high-level synthesis methods, operations are scheduled assuming that functional units take same latencies for the same operations. Namely, the scheduling is determined ahead of the time 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 waiting cycles arise when the units take less latencies than those of the worst case.

To address this issue, [2] proposed a adaptable controller which adjusts scheduling of operations at run-time based on the completion signals from the variable latency units. Although it enables efficient execution without wasteful waits, the number of its states increased explosively with the size of dataflow graphs (DFGs). As a promising alternative, distributed control has been recently developed in which a datapath is controlled by multiple finite state machines. Some different schemes for distributed controllers have been proposed by Del Barrio [3], Pilato [4], and Yamashita [5].

However, all these methods deal with a simple case where computation is expressed with a single DFG. In order to apply the distributed control in control centric high-level synthesis, control data flow graphs (CDFGs) consisting of multiple DFGs must be handled.

This paper, therefore, proposes a method of extending the distributed controller to rein the variable latency units to handle multiple DFGs. It is an extension of Del Barrio's method. Rather than simply patchworking the state transition graphs for the DFGs, it enables dynamic operation motion across multiple DFGs, which achieves efficient execution like loop scheduling and trace scheduling. An experiment on two benchmarks has proved that the total execution cycles were drastically reduced by the proposed method with about 24% increase in the circuit size.

II. Variable latency units and distributed control

A. 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. NBTI may cause long term change on delays and latencies of the functional units.

Let us consider executing a DFG in Fig. 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 based on fixed scheduling, 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

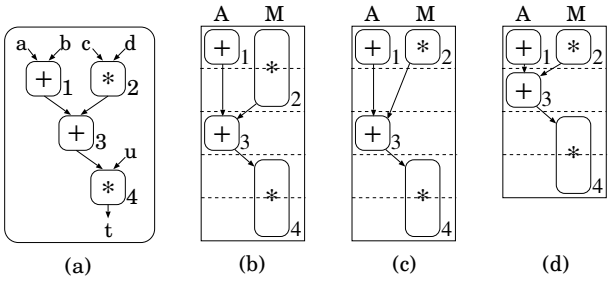
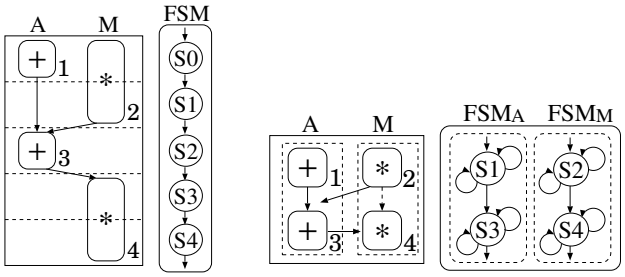


Fig. 1. Execution flow with variable latency units.



(a) Centralized control (b) Distributed control of Del Barrio

Fig. 2. Centralized and distributed control

scheduling as shown in Fig. 1 (d). However, the controller for this dynamic scheduling needed huge amount of states, so the resulting circuit would be impractically large.

B. Distributed control

As an approach to realizing dynamic scheduling without increasing 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 timings 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. The Yamashita's method [5] allows hardware to dynamically decide functional unit to execute each operation as well as the order and the timing.

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.

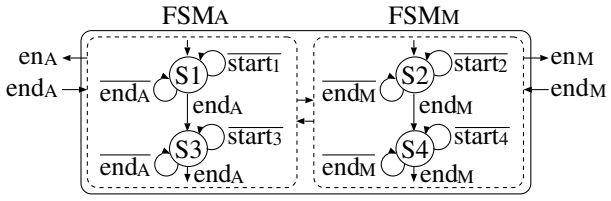


Fig. 3. Distributed control of Del Barrio

C. Del Barrio's distributed control

In the conventional centralized control method, a whole datapath is controlled by a single FSM (finite state machine), as shown in Fig. 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 separately.

Fig. 3 is the details of the FSMs in Fig. 2 (b). The formulation in this paper is slightly different from the original one in [3], but substantially the same. One state S_i is assigned to an operation i in the DFG, where 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 Fig. 3,

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

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.

$$\text{if } (s_1 \wedge end_A)\ \{Done_1 <= 1\},$$
$$\text{if } (s_2 \wedge end_M)\ \{Done_2 <= 1\},$$
$$\text{if } (s_3 \wedge end_A)\ \{Done_3 <= 1\},$$
$$\text{if } (s_4 \wedge end_M)\ \{Done_4 <= 1\}.$$

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

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

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

$$\text{if } (end_A)\ \{Exec_A <= 0\}$$
$$\quad \text{else if } (en_A)\ \{Exec_A <= 1\},$$
$$\text{if } (end_M)\ \{Exec_M <= 0\}$$
$$\quad \text{else if } (en_M)\ \{Exec_M <= 1\}.$$

Del Barrio's distributed controller exhibits less freedom than Pilato's and Yamashita's, because it can not change the order of the operations nor the function units to execute the operations at run time. However, it is the simplest of the three, and the size and the delay of implemented hardware will be the smallest.

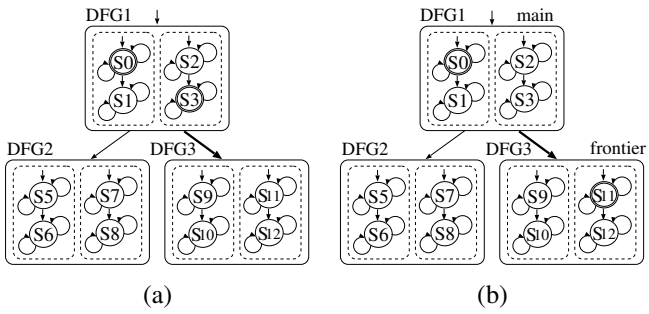


Fig. 4. Distributed control beyond the borders of DFGs.

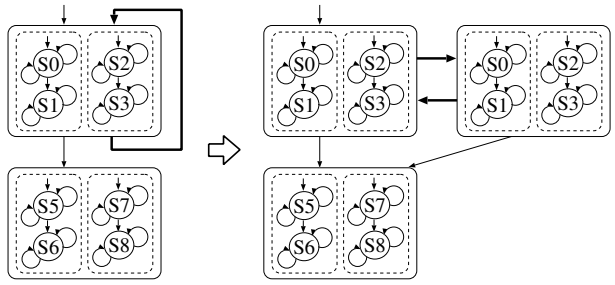


Fig. 5. Eliminating a self loop.

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.

III. Distributed Control beyond the Borders of Dataflow Graphs

A. Overview

This paper extends the Del Barrio's distributed control to handle multiple DFGs, where dynamic scheduling of operations is achieved across multiple DFGs.

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. By applying the same policy, the distributed control may be extended to multiple DFGs. However, in the presence of variable latency operations, not all the units finish their task in an DFG simultaneously but they must wait for each other at the end of the DFG. To reduce the inefficiency, in our method, operations in the next DFG can start execution before concluding the current DFG. For example, in Fig. 4 where two units are controlled by two controllers, the operations corresponding to $S0$ and $S3$ are finished (as in (a)), 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 this paper, we assume that only one DFG ahead of the current DFG may be executed. The current DFG is hereafter referred to as a *main* DFG and the partially executed next DFG a *frontier* DFG. In Fig. 4 (b), DFG1 is the main DFG and DFG3 is the frontier DFG. A self loop is eliminated by duplicating the DFG, as shown in Fig. 5.

For design convenience, each FSM of a unit must have at least one state per DFG; if there is no state for a DFG, a dummy state in which no operation is executed is inserted.

B. Formulation

Let U be the set of units in the given datapath. For $u \in U$, o_u and e_u are the output and the completion signal of u at the current cycle, respectively. Let D be the set of DFGs. An FSM to control unit $u \in U$ and DFG $d \in D$ is denoted as $F_{d,u}$. Let $S_{d,u}$ and $f_{d,u}$ be the set of states and the last sate of $F_{d,u}$, respectively, and $\sigma_{d,u}$ be the current state of $F_{d,u}$.

For $d \in D$ and $s \in S_{d,u}$, $\Gamma(s)$, $\gamma(s)$, and $e(d)$ are defined as follows. Intuitively, $\Gamma(s)$ means that execution of the operation at state s has been finished before the current cycle, $\gamma(s)$ means that execution of the operation at state s is finished at the current cycle, and $e(d)$ means that execution of DFG d completes at the current cycle. $\Gamma_0(s)$ and $\Gamma'(s)$ are the initial value and the next value (the value at the next cycle) of $\Gamma(s)$, respectively.

$$\gamma(s) = \Gamma(s) \vee ((\sigma_{d,u} = s) \wedge e_u)$$
$$e(d) = \bigwedge_{u \in U} \gamma(f_{d,u})$$
$$\Gamma_0(s) = 0$$
$$\Gamma'(s) = \textbf{if } e(d) \textbf{ then } 0 \textbf{ else } \gamma(s)$$

When $e(d)$ becomes 1, the next state of $F_{d,u}$ is the initial state for every $u \in U$.

For $e \in D$ and $d \in D$, $\Delta(e,d)$ means that transition from e to d has been fixed before the current cycle, and $\delta(e,d)$ means that transition from e to d is fixed at the current cycle. If d is the unique successor of e, then $\Delta(e,d) = \delta(e,d) = 1$. Otherwise, $\delta(e,d)$ and $\Delta(e,d)$ are defined as follows assuming that the transition from e to d is determined by the output of unit u at state s.

$$\delta(e,d) = \Delta(e,d) \vee ((\sigma_{e,d} = s) \wedge e_u \wedge o_u)$$
$$\Delta_0(e,d) = 0$$
$$\Delta'(e,d) = \textbf{if } e(e) \textbf{ then } 0 \textbf{ else } \delta(e,d)$$

$\mathcal{M}(d)$ means that $d \in D$ is executed as the main DFG, whose initial value $\mathcal{M}_0(d)$ and next value $\mathcal{M}'(d)$ is defined as follows, where P_d is the set of DFGs preceding d.

$$\mathcal{M}_0(d) = \textbf{if } d \text{ is the starting DFG } \textbf{then } 1 \textbf{ else } 0$$
$$\mathcal{M}'(d) = \textbf{if } e(d) \textbf{ then } 0 \textbf{ else } \mathcal{M}(d) \vee \bigvee_{e \in P_d} (e(e) \wedge \delta(e,d))$$

$\mathcal{F}(d)$ means that DFG $d \in D$ is executed as the frontier DFG, whose initial value $\mathcal{F}_0(d)$ and next value $\mathcal{F}'(d)$ is defined as follows.

$$\mathcal{F}_0(d) = 0$$

TABLE I
EXPERIMENTAL RESULT (#CYCLE)

	centralized	distributed (proposed)		
		$r = 1.0$	$r = 0.5$	$r = 0.0$
diffeq	770	770	632	513
iprod64	710	740	541	422

Loop iteration: 128, multiplication: 1∼2 cycles,
r: probability where multiplication took 2 cycles,

$$\mathcal{F}'(d) = \textbf{if } \mathcal{M}'(d) \textbf{ then } 0 \textbf{ else } \mathcal{F}(d) \vee \bigvee_{e \in P_d} (\mathcal{M}'(e) \wedge \delta(e,d))$$

The operations in DFG d are executed only if $\mathcal{M}(d) \vee \mathcal{F}(d)$. All the other conditions are same as the Del Barrio's control in subsection II-C.

IV. Experimental Result

RTL circuits based on the proposed distributed control were designed in Verilog HDL for two benchmark CDFGs and compared with ones based on the conventional centralized control. Throughout the experiments, it was assumed that multiplication took either 1 or 2 cycles and the other operations took 1 cycle.

A. Benchmark CDFG

(1) diffeq

A DFG in Fig. 6(a) is executed with two adders A1 and A2, and two multipliers M1 and M2. Fig. 6(b) is an example of binding and scheduling for distributed control. The DFG is duplicated to remove the self loop. For comparison, the same DFG is scheduled for centralized control, as shown in Fig. 6(c).

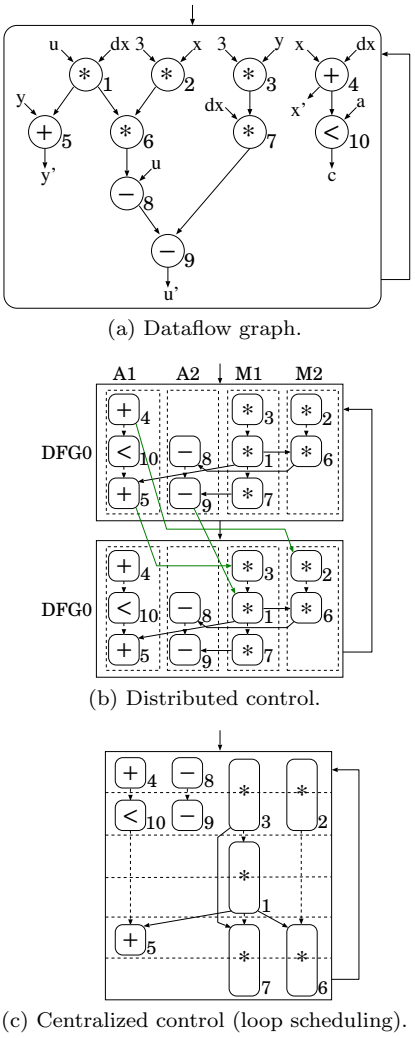
(2) iprod64

The DFG in Fig. 7(a) computes the inner product of 64-bit data, where 64-bit multiplication is implemented by four 32-bit multiplications and three additions. However, some of the computation may be skipped, if the higher or the lower half of the multiplier is 0. The DFG is computed with an ALU (A), an ALU+shifter (AS), a multiplier (M), a load/store unit (L), a comparator (E). C in the DFG represents data transfer without operation.

Fig. 7(b) shows an example of scheduling/biding for our distributed control. The self loops are eliminated by duplicating DFG0. Since our method requires that the FSM for each unit must have at least one state for each DFG, dummy states (depicted by the broken lines), which do not issue any operation and just transfer to the next state, are inserted for such DFGs. For comparison, the same computation is scheduled for centralized control with the same numbers of the units. Fig. 7(c) is the CDFG obtained by trace scheduling.

B. Performance (cycle count)

Table I compares the cycle counts on the two benchmarks. The loop was iterated 128 times on the both



(a) Dataflow graph.

(b) Distributed control.

(c) Centralized control (loop scheduling).

Fig. 6. Benchmark "diffeq."

benchmarks. The column "centralized" shows the result for the conventional fixed centralized control with the scheduling in Fig. 6(c) and Fig. 7(c), where the latency of all the multiplications were fixed at two cycles. The column "distributed" shows the cycle counts for our variable and distributed control. The parameter r is the probability where multiplication took 2 cycles; $r = 1.0$ means all the multiplications took two cycles, while $r = 0.5$ means half of the multiplications took two cycles and the others one cycle. In the iprod benchmark, the each of the upper 32-bit and the lower 32-bit became 0 with a probability of 50%.

On the diffeq benchmark, the proposed method achieved the same cycle count as the loop scheduling on $r = 1.0$. Furthermore, our distributed control is much efficient than the centralized scheduling on $r = 0.5$ and $r = 0.0$.

On the iprod benchmark, however, our method resulted in an increase in the cycle count. This is due to insertion of the dummy state and its root cause is that our method allows operation execution in the DFG only one transition ahead of the current active DFG. Nevertheless, the pro-

TABLE II
Experimental result (circuit size)

	centralized		distributed (proposed)	
	slices (FFs)	delay [ns]	slices (FFs)	delay [ns]
diffeq	567 (331)	15.044	702 (441)	14.878
iprod64	724 (464)	14.492	890 (556)	15.871

posed variable scheduling reduced the cycle count when $r = 0.5$ and $r = 0.0$.

Fig. 8 shows examples of execution traces of the diffeq benchmark by our method. (a) is the case where all the multiplications take two cycles. Units A1, M1, and M2 can start the second iteration (highlighted by red) before A2 finishes the first iteration, so cycle per iteration is reduced to 6. This is equivalent to the loop scheduling (determined ahead of the execution). (b) shows the case where 6 out of 12 multiplications finish in 1 cycle. Scheduling is dynamically adjusted beyond the border of the DFGs which contributes to the drastic reduction in the total cycle count.

An execution trace example for iprod64 is also shown in Fig. 9 with $r = 1.0$. The result is almost equivalent to the conventional trace scheduling, except that the dummy state, indicated by (1), increases the cycle count by one. Note that this occurs only when all the multiplications take 2 cycles; if one of them finishes in 1 cycle, the dummy state is not on the critical path and does not cause the increase in the cycle count.

C. Circuit size

TABLE II summarizes the result of logic synthesis by Xilinx ISE (14.7) targeting FPGA (Spartan–3E). The columns slices, FFs, delay show the circuit size in terms of slices and flip-flops, and the critical path delay. The circuits in our control method is about 24% larger than those by the conventional implementation. This is due to the increase in the number of registers (and consequently increase in the total multiplexer cost) because the lifetimes of the values overlap more frequently in variable latency scheduling. The critical path delay is almost the same. Thus we may conclude that the proposed controller can be implemented in hardware of a feasible size.

V. Discussion

Like the loop scheduling and trace scheduling methods, the proposed method yields efficient computation scheduling by moving operations across DFGs. In addition, our distributed control allows run-time adjustment of the scheduling in response to varying latency of operations. On the other hand, our method may be less efficient than the conventional (fixed) scheduling because it allows operations to move only to the adjacent DFGs.

It is possible to apply Del Barrio's distributed control to a DFG obtained by loop scheduling and trace scheduling, which enables both operation motion across DFGs and run-time scheduling. This combination may outperform our method on the presumed execution trace on which the scheduling is determined, because operations can be moved by more than one DFGs. However, our distributed control may work better when the prediction fails, because it can handle both the predicted/unpredicted branches equally.

In our method, as well as in trace and loop scheduling, operation motion does not work when branch taken/untaken is determined at the final step of a DFG. This issue may be addressed by introducing branch prediction and speculative execution.

VI. Conclusion

This paper has presented a distributed control method that allows dynamic operation motion across DFGs for efficient scheduling under the existence of variable latency operations. Experiment on two benchmarks demonstrated drastic reduction in execution cycles with about 24% increase in hardware size.

We are now working on adapting our method to other distributed control methods such as [4] and [5], and on introducing branch prediction and speculative execution to further enhance performance.

Acknowledgements

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 was partly supported by JSPS KAKENHI Grant Number 16K00088.

References

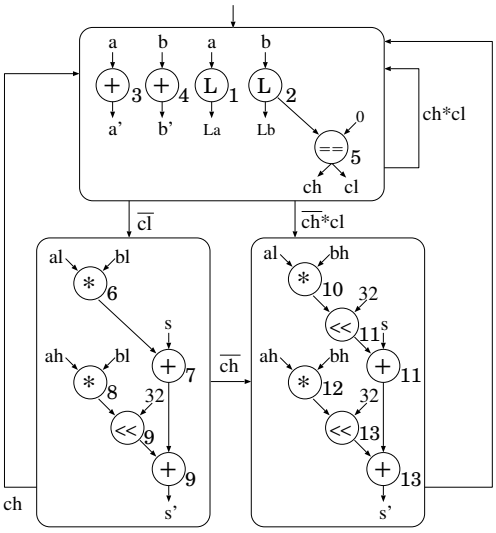
[1] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic (1992).

[2] Yuki Toda, Nagisa Ishiura, and Kousuke Sone: "Static scheduling of dynamic execution for high-level synthesis," in *Proc. SASIMI 2009*, pp. 107–112 (Mar. 2009).

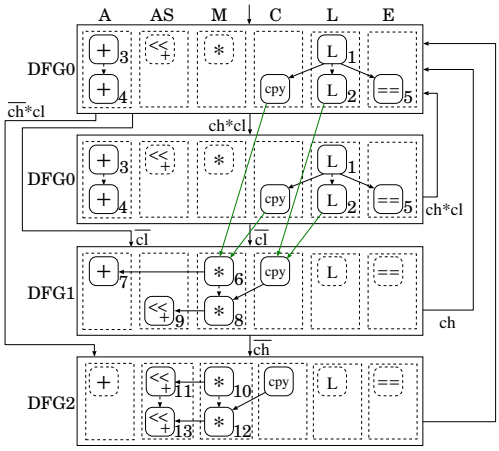
[3] Alberto A. Del Barrio, Seda Ogrenci Memik, María C. Molina, José M. Mendías, and Román Hermida: "A distributed controller for managing speculative functional units in high-level synthesis," *IEEE Trans. CAD*, pp. 350–363 (Mar. 2011).

[4] Christian Pilato, Vito Giovanni Castellana, Silvia Lovergine, and Fabrizio Ferrandi: "A runtime adaptive controller for supporting hardware components with variable latency," in *Proc. NASA/ESA AHS-2011*, pp. 153–160 (June 2011).

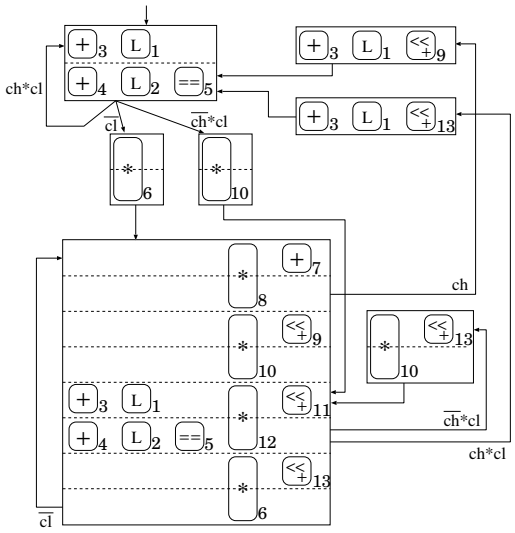
[5] Shinji Yamashita and Nagisa Ishiura: "Dynamic operation binding in distributed controller for supporting functional units with variable latency" (in Japanese), in *Technical Report of IEICE*, VLD2013–128 (Jan. 2014).



(a) Dataflow graph.



(b) Distributed scheduling.



(c) Centralized scheduling (trace scheduling).

Fig. 7. Benchmark "iprod64."

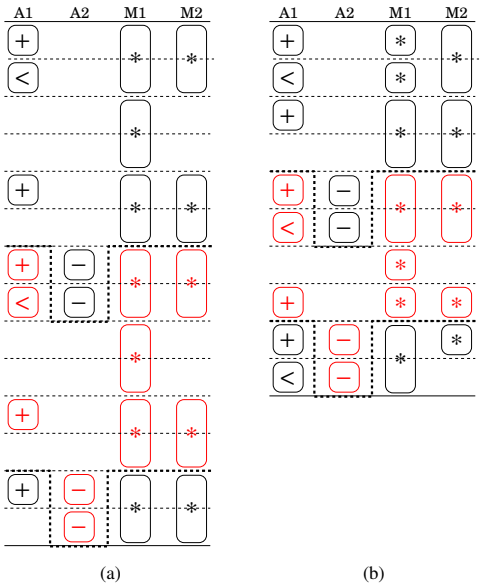


Fig. 8. Example execution flow of diffeq.

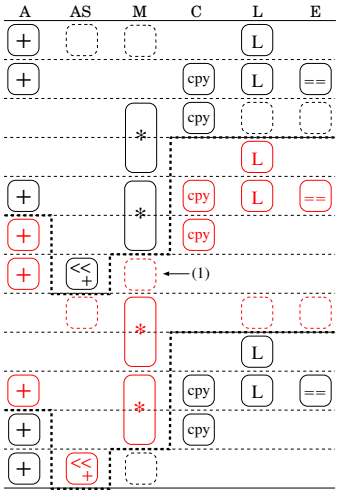


Fig. 9. Example execution flow of iprod64.