IPSJ Transactions on System LSI Design Methodology Vol.7 119–124 (Aug. 2014)

[DOI: 10.2197/ipsjtsldm.7.119]

Regular Paper

Forwarding Unit Generation for Loop Pipelining in High-level Synthesis

Shingo Kusakabe[1,a)] Kenshu Seto[1,b)]

Received: December 6, 2013, Accepted: February 14, 2014, Released: August 4, 2014

Abstract: In the loop pipelining of high-level synthesis, the reduction of initiation intervals (IIs) is very important. Existing loop pipelining techniques, however, pessimistically assumes that dependences whose occurrences can be determined only at runtime always occur, resulting in increased IIs. To address this issue, recent work achieves reduced II by a source code transformation which introduces runtime dependence analysis and performs pipeline stalls when the dependences actually occur. Unfortunately, the recent work suffers from the increased execution cycles by frequent pipeline stalls under the frequent occurrences of the dependences. In this paper, we propose a technique to reduce IIs in which data written to memories are also written to registers for such dependences of read-after-write (RAW) type. In our technique, registers which are faster than memories are accessed when the RAW dependences occur. Since the proposed technique achieved the reduction of the execution cycles by 34% with 15% gate count increase on average for three examples compared to the state-of-the-art technique, the proposed technique is effective for synthesizing high-speed circuits with loop pipelining.

Keywords: high level synthesis, loop pipelining, memory access, dependence, RAW, forwarding

1. Introduction

Loop pipelining in high-level synthesis (HLS) [1] is an effective technique for quickly generating high-performance hardware accelerators from time-consuming loops. In loop pipelining [2], a loop iteration is executed after a fixed number of execution cycles before the completion of the previous loop iteration. The fixed number of execution cycles are called initiation interval (II) and the reduction of IIs is the key to generating high-performance accelerators.

Existing loop pipeline techniques [2], [3], [4], [5], [6], however, fail to reduce IIs when loops contain dependences whose occurrences can be determined only at runtime. Hereafter, we call such dependences *possible dependence*. **Figure 1** shows a loop that contains a possible dependence from the array write A[i] in a loop iteration to the array read A[B[i]] in the next iteration. Since the values of the array read accesses B[i] can typically be known only at runtime, the array dependence from the write access A[i] at an iteration to the read access A[B[i]] at the next iteration is a possible dependence. When loops contain such dependences, existing loop pipelining techniques [2], [3], [4], [5], [6] pessimistically assume that the dependences always occur, so that the reductions of IIs are limited.

To resolve the problem of increased execution cycles by possible dependences, Ref. [7] proposed a scheduling method that checks the occurrences of possible dependences at runtime by comparing the values of array subscripts. The possible dependences are nullified when the values of the array subscripts are different so that the number of execution cycles after scheduling are reduced. Since Ref. [7] generates multiple schedules depending on the occurrences of the possible dependences, the generated controllers are likely to be complex which may affect the operating frequencies of generated circuits. As a similar approach to Ref. [7], Alle et al. [8] proposed a source-level transformation that is applied before loop pipelining in order to resolve the problem of the suboptimal IIs by existing loop pipelining techniques. Reference [8] reduces IIs by checking whether possible dependences actually occur and stalling pipelined loops until the dependences are resolved. Although IIs are reduced by Ref. [8], the reductions of execution cycles are likely to be limited when possible dependences often occur and pipelined loops stall frequently.

```
1: for ( i=0; i<N; i++ ){
2:       A[i] = A[B[i]] + c;
3: }
```

Fig. 1 Loop with a dependence whose occurrence can be determined only at runtime.

In this paper, we propose a source-code transformation that is applied before loop pipelining as in Ref. [8] and that reduces the numbers of execution cycles for loops with possible dependences, or more specifically, loop-carried RAW (Read-After-Write) possible dependences. Similar to Ref. [8], the transformed code by the proposed transformation checks the possible dependences at runtime. Unlike [8], however, the written data to arrays in the RAW possible dependences are also written to scalar variables in the proposed technique and the scalar variables are accessed instead of the arrays when the possible dependences actually occur. The proposed technique replaces the array accesses in the

[1] Tokyo City University, Setagaya, Tokyo 158-8557, Japan
[a)] g1015026@tcu.ac.jp
[b)] kseto@tcu.ac.jp

maybe RAW dependences with the scalar variable accesses, so that pipeline stalls that arise in Ref. [8] due to array accesses in the maybe RAW depdences are avoided and the numbers of execution cycles are significantly reduced compared to Ref. [8].

2. Problem of Existing Loop Pipelining Techniques When Loops Have Possible Dependences

In this section, we first show the problem of existing loop pipelining techniques [2], [3], [4], [5], [6] when loops contain *possible dependences*, namely, dependences whose occurrences can be determined only at runtime. Next, we overview the previous work [8] that addresses the problem of the existing techniques and illustrate the problem of Ref. [8] which is the state-of-the-art.

In this and the following sections, we use the example code in Fig. 1 for illustration. **Figure 2** shows the data dependence graph (DDG) for the loop body in Fig. 1 where each box represents an operation and each edge represents a data dependence. The broken line in Fig. 2 drawn from the array write access WR A[i] to the array read access RD A[B[i]] shows a possible dependence which is a RAW (Read After Write) dependence and a loop-carried dependence. A loop-carried dependence is a dependence from a variable access in a loop iteration to a variable access in the subsequent iterations.

Figure 3 shows the loop pipelining result for the DDG in Fig. 2 by the existing loop pipelining techniques [2], [3], [4], [5], [6]. We assume that addition (ADD) and memory accesses (RD and WR) take one clock cycle, respectively. Figure 3 presents the case where the initiation interval (II) is 3 which means each loop iteration starts 3 cycles after the previous iteration.

The occurrence of the possible dependence from WR A[0] to RD A[B[1]] in Fig. 3 can be checked by comparing the values of the array subscripts of the array accesses, namely, 0 and B[1]. If B[1] equals to 0, the values of the array subscripts are the same so that the possible dependence from WR A[0] to RD A[B[1]] actually occurs. In this case, the II should be at least 3 in order to follow the possible dependence that actually occurred. If B[1] does not equal to 0, the values of the array subscripts are different so that the possible dependence from WR A[0] to RD A[B[1]] does not actually occur. In this case, the II can be the minimum value, 1, since the possible dependence does not occur. By the existing loop pipelining techniques, the II becomes 3 since these techniques pessimistically assume the possible dependences such as the one from WR A[0] to RD A[B[1]] always occur.

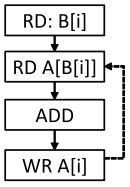

Fig. 2 Data dependence graph (DDG) of the loop body in Fig. 1.

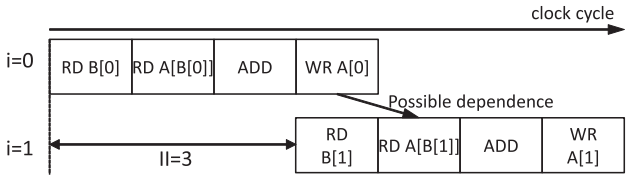

Fig. 3 Loop pipelining result by existing loop pipelining techniques.

To address the increased IIs by the existing loop pipelining techniques, the previous work [8] proposed a technique that checks occurrences of possible dependences at runtime and stalls pipeline execution when the possible dependences actually occur. By introducing pipeline stalls, Ref. [8] successfully reduces IIs without violating the possible dependences. The main technique proposed in Ref. [8] is a source-level transformation and the resulting code after applying the technique to the code in Fig. 1 is shown in **Fig. 4**.

The code in Fig. 4 incorporates a pipeline stall logic to achieve reduced IIs. More specifically, the stall logic is described from line 3 to line 8 in Fig. 4 where *stall* represents a flag or a control signal that stalls pipeline execution and inserts pipeline bubbles. To detect violations of the possible dependence from the write access A[i] in a loop iteration to the read accesses A[B[i]] in the following iterations, the subscripts i and B[i] are stored in the scalar variables A_waddr_0 and A_raddr, respectively. The subscripts of the write access A[i] of the previous iterations are also kept in the scalar variables A_waddr_1 and A_waddr_2 where A_waddr_k represents the value of the array subscript k iterations before the current loop iteration. The scalar variables A_waddr_0, A_waddr_1 and A_waddr_2 that holds array subscripts of current and previous iterations are shift registers and are shifted in each loop iteration from line 12 to 13 in Fig. 4. To detect occurrences of the possible dependence, the subscript values of the array write and read accesses are compared as shown from line 4 to line 5 in Fig. 4 and *stall* flag is set to one when the possible dependence actually occurs. When *stall* flag is one, the loop body and the increment of the loop index *i* are not executed as shown from line 8 to line 11 in Fig. 4 resulting in a pipeline stall.

Figures 5 and **6** show the loop pipelining results of applying

```
 1: while(i<N){
 2:       A_raddr = B[i];
 3:       stall = 0;
 4:       stall = stall | (A_raddr == A_waddr_2);
 5:       stall = stall | (A_raddr == A_waddr_1);
 6:       if(stall) A_waddr_0 = -1;
 7:       else      A_waddr_0 = i;
 8:       if(!stall){
 9:            A[i] = A[A_raddr] + c;
10:            i++;
11:       }
12:       A_waddr_2 = A_waddr_1;
13:       A_waddr_1 = A_waddr_0;
14: }
```

Fig. 4 Code transformation result by the previous technique [8] applied to the example in Fig. 1.

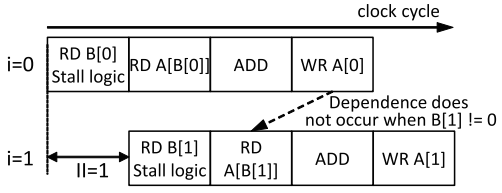

Fig. 5 Loop pipelining result after applying the previous technique [8] (the case when the stall does not occur).

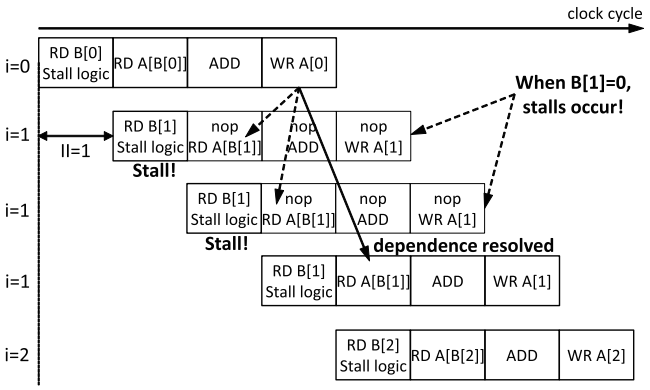

Fig. 6 Loop pipelining result after applying the previous technique [8] (the case when stalls occur).

the previous technique [8]. As shown in the figures, the technique checks the actual occurrences of the possible dependence by the stall logic at the first stage of pipelined execution of each loop iteration. In the case when the possible dependence is found not to occur in an iteration, the iteration executes without pipeline stall as shown in Fig. 5. On the other hand, in the case when the possible dependence is found to occur in an iteration, the pipeline execution is stalled in the iteration until the dependence is resolved as shown in Fig. 6. In both cases, the previous technique [8] achieves II=1, however, the number of execution cycles is likely to increase because of pipeline stalls.

3. Proposed Technique: Source-level Transformation for Generating Forwarding Units

In this section, we first overview how the proposed source-level transformation overcomes the problem of the previous work [8] illustrated in Section 2. Next, we describe an algorithm for the proposed transformation in detail.

3.1 Overview of the Proposed Source-level Transformation

The reason of pipeline stalls by the previous work [8] as shown in Fig. 6 is delays associated with the sequence of the memory write A[i] followed by the memory read A[B[i]] in the loop-carried RAW (Read After Write) possible dependence. In high-level synthesis, arrays are typically mapped to synchronous memories with 1 clock cycle delay for read and write accesses. As a result, 2 clock cycle delays, which is the sum of the 1 clock cycle delay by the write access to A[i] and 1 clock cycle delay by the dependent read access to A[B[i]], arise when the possible dependence actually occurs.

Based on the observation that [8] results in the 2 clock cycle delays due to the dependent sequences of the array accesses in the loop-carried RAW possible dependences, the proposed technique stores data not only in arrays but also in scalar variables for such dependences. In high-level synthesis, scalar variables are mapped to registers and registers can be written or read without clock cycle delay, so we can expect the 2 clock cycle delays are successfully eliminated and the numbers of execution cycles are significantly reduced by the proposed approach.

Figure 7 shows the resulting code after applying the proposed transformation to the code in Fig. 1. In the similar way as the code in Fig. 4 by the previous technique [8], the code in Fig. 7

```
 1: for(i=0; i<N; i++){
 2:       A_raddr = B[i];
 3:       if(A_raddr == A_waddr_2)       A_rdata = A_wdata_2;
 4:       else if(A_raddr == A_waddr_1) A_rdata = A_wdata_1;
 5:       else                                  A_rdata = A[A_raddr];
 6:       A_waddr_0 = i;
 7:       A_wdata_0 = A_rdata + c;
 8:       A[i] = A_wdata_0;
 9:       A_waddr_2 = A_waddr_1;
10:       A_waddr_1 = A_waddr_0;
11:       A_wdata_2 = A_wdata_1;
12:       A_wdata_1 = A_wdata_0;
13: }
```

Fig. 7 Code transformation result by the proposed technique applied to the example in Fig. 1.

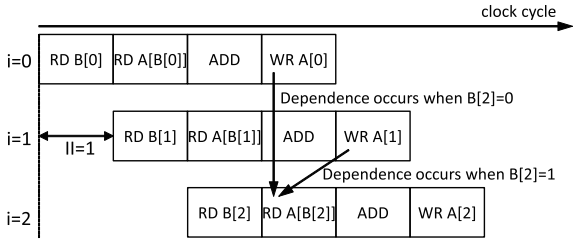

Fig. 8 Loop pipelining result after applying the proposed technique.

also stores the subscripts i and B[i] of the array accesses A[i] and A[B[i]] in the loop-carried RAW possible dependence to the scalar variables A_waddr_0 and A_raddr in line 6 and line 2, respectively, and detects occurrences of the dependence by comparing the values of the subscripts from line 3 to 4 in Fig. 7. In contrast to Ref. [8], the proposed transformation stores the data written in the maybe RAW dependence not only to the array A[i] but also to the scalar variable A_wdata_0 from line 7 to 8 in Fig. 7. In the same way as the scalar variables A_waddr_k that store the values of the array subscript i of B[i] k iterations before the current loop iteration, the scalar variable A_wdata_k holds the data written to the array element A[i] k iterations before the current loop iteration. A_wdata_0, A_wdata_1 and A_wdata_2 are shift registers and are shifted in each loop iteration from line 11 to 12 in Fig. 4 similarly to A_waddr_0, A_waddr_1 and A_waddr_2.

Figure 8 shows the loop pipelining result for the code in Fig. 7 which is obtained by applying the proposed transformation to the code in Fig. 1. As shown in Fig. 8, pipeline stalls do not arise even when the possible dependence actually occurs and II of Fig. 8 is always 1. **Figure 9** illustrates why the loop pipelining result for the code generated by the proposed transformation achieves II=1 even when the possible dependence actually occurs. As shown in the pipelined loop in Fig. 9, the array read access to A[B[2]] arise for the iteration i=2 and in the 4th clock cycle. Depending on the value of the array subscript B[2], the array read access A[B[2]] may depend on the array write accesses to A[0] or A[1] of the previous iterations at i=0 and i=1. Both of the array write accesses A[0] at i=0 and A[1] at i=1 do not finish before the 4th clock cycle so that the array read access A[B[2]] at i=2 does not provide the correct value when B[2] is either 0 or 1. To avoid this problem, we use the values stored in the shift registers A_wdata_2 or A_wdata_1 instead of the value stored in A[B[2]] in order to perform the addition for the iteration i=2 in the 5th clock cycle. More specifically, A_wdata_2 is used when B[2]=0 and

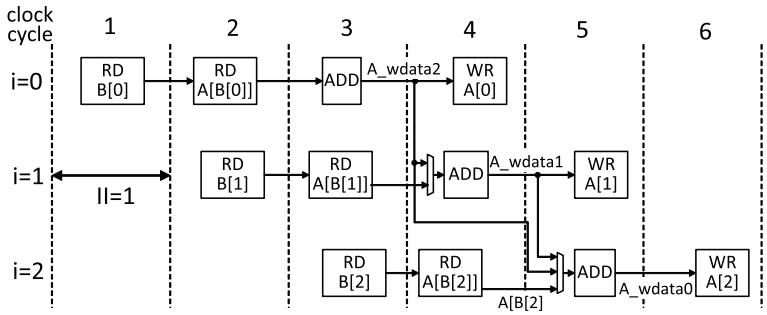

Fig. 9 Forwarding unit generated by the proposed technique.

Algorithm *GenerateForwardingUnit*

GenerateForwardingUnit(*L*, *n*)
Input *L*: code for a loop body
Input *n*: depth of shift registers
1: *LCDs* ← all loop-carried RAW possible dependences in *L*
2: *DSTs* ← all (unique) array read accesses in *LCDs*
3: *SRCs* ← all (unique) array write accesses in *LCDs*
4: **for each** *src* in *SRCs*
5: Replace "*src*" in *stmt*(*src*) with scalar variable "*srcID_wdata_0*"
6: Add statement "*src = srcID_wdata_0;*" just below *stmt*(*src*)
7: **end for**
8: **for each** *dst* in *DSTs*
9: Replace "*dst*" in *stmt*(*dst*) with scalar variable "*dstID_rdata*"
10: **end for**
11: **for each** *dst* in *DSTs*
12: Add statement "*dstID_raddr=subsc*(*dst*);" just before *stmt*(*dst*)
13: Add the following conditional statement just before *stmt*(*dst*)
14: **for each** *src* in *SRCs*
15: **for** *k* = depth of shift registers *n* to 1
16: Condition: "if/else if (*dstID_raddr == srcID_waddr_k*)"
17: Statement: "*dstID_rdata = srcID_wdata_k;*"
18: **end for**
19: Condition: "else"
20: Statement: "*dstID_rdata = src;*"
21: **end for**
22: **end for**
23: **for each** *src* in *SRCs*
24: **for** *k* = depth of shift registers *n* to 1
25: Add the following statement at the end of loop body
26: "*srcID_waddr_k = srcID_waddr_k*-1;"
27: **end for**
28: **for** *k* = depth of shift registers *n* to 1
29: Add the following statement at the end of loop body
30: "*srcID_wdata_k = srcID_wdata_k*-1;"
31: **end for**
32: **end for**

Fig. 10 Procedure for generating forwarding units.

A_wdata_1 is used when B[2]=1 to avoid pipeline stalls. When B[2] does not equal to neither 0 nor 1, the array read access to A[B[2]] provides the correct value because of the absense of the possible dependence, so that the value of A[B[2]] can be used for the addition for the iteration i=2 in the 5th clock cycle. This switching of the operands for the addition for the iteration i=2 in the 5th clock cycle is described from line 3 to line 5 in Fig. 7 and depicted by the 3-to-1 multiplexer for the iteration i=2 in the 5th clock cycle in Fig. 9. As seen from Fig. 9, the proposed technique automatically generates forwarding units which is a well known mechanism to accelerate processor pipelines [9]. As far as the authors know, this paper proposes automatic forwarding unit generation to reduce IIs in loop pipelining for the first time.

3.2 Algorithm for the Proposed Source-level Transformation

In this section, we present an algorithm for the proposed source-level transformation illustrated in Section 3.1 that automatically transforms the code in Fig. 1 into the code in Fig. 7 and generates forwarding units to reduce the numbers of execution cycles for pipelined loops. The proposed algorithm *GenerateForwardingUnit* is shown in **Fig. 10**.

The inputs to the algorithm are C code *L* for a loop body and the depth *n* of the shift registers. The depth *n* is set to the number of execution cycles for one loop iteration, which is obtained by running high-level synthesis tools. In the algorithm in Fig. 10, *stmt*(*a*) means the statement where an array access *a* belongs to and *subsc*(*a*) means the subscript of an array access *a*. *LCDs* represents the set of all loop-carried RAW possible dependences in the loop body *L*. *DSTs* and *SRCs* contain the set of all but unique array read accesses in *LCDs* and the set of all but unique array write accesses in *LCDs*, respectively. From line 4 to 7 in Fig. 10, data written in maybe RAW dependences are also stored to scalar variables, which corresponds to the code fragment from line 7 to 8 in Fig. 7. From line 8 to 10 in Fig. 10, array read accesses in maybe RAW dependences are replaced with scalar variables, which corresponds to line 7 (right hand side) of the code in Fig. 7. From line 11 to 22 in Fig. 10, conditional statements are generated in order to select appropriate scalar variables when loop-carried RAW possible dependences actually occur, which corresponds to the code fragment from line 3 to 5 in Fig. 7. This conditional statements are implemented as multiplexers in synthesized circuits. From line 23 to 32 in Fig. 10, statements that shift scalar variables that store written data and corresponding subscripts of array write accesses in maybe RAW dependences are generated, which corresponds to the code fragment from line 9 to 12 in Fig. 7.

4. Experiments

In this section, we show the impacts of the proposed transformation on design metrics including initiation intervals (IIs), the numbers of execution cycles, critical path delays and gate counts of the synthesized circuits. In particular, we compare these design metrics by the proposed approach with those by the previous state-of-the-art approach [8].

4.1 Experimental Setups

We implemented the proposed algorithm in Fig. 10 in our pro-

Table 1 Comparison between the previous technique (stall) [8] and the proposed technique.

Benchmarks	code type	II [cycles]	# of execution cycles [cycles]	# of pipeline stalls [cycles]	critical path delays [ps]	total gate counts [gates]	gate counts for FFs [gates]
Example	original	3 (1.00)	772 (1.00)	-	3494 (1.00)	270 (1.00)	127 (1.00)
	stall	1 (0.33)	471 (0.61)	211	3829 (1.10)	567 (2.10)	268 (2.11)
	proposed	1 (0.33)	261 (0.34)	0	3829 (1.10)	596 (2.21)	297 (2.34)
Histgram (*sunflower*)	original	3 (1.00)	922k (1.00)	-	3822 (1.00)	347 (1.00)	96 (1.00)
	stall	1 (0.33)	396k (0.43)	88k	3837 (1.00)	713 (2.05)	263 (2.74)
	proposed	1 (0.33)	309k (0.34)	0	3831 (1.00)	983 (2.83)	414 (4.31)
Histgram (*night scene*)	original	-	922k (1.00)	-	-	-	-
	stall	-	718k (0.78)	409k	-	-	-
	proposed	-	309k (0.34)	0	-	-	-
SMVM	original	7 (1.00)	64 (1.00)	-	3813 (1.00)	2157 (1.00)	652 (1.00)
	stall	2 (0.29)	51 (0.80)	24	3897 (1.02)	2480 (1.15)	839 (1.29)
	proposed	3 (0.43)	34 (0.53)	0	3816 (1.00)	2333 (1.08)	755 (1.16)
Average	original	4.33 (1.00)	- (1.00)	-	3710 (1.00)	- (1.00)	292 (1.00)
	stall	1.33 (0.31)	- (0.70)	-	3854 (1.04)	- (1.77)	457 (1.57)
	proposed	1.67 (0.39)	- (0.40)	-	3825 (1.03)	- (2.04)	489 (1.68)

totype tool, and applied the algorithm to 3 benchmark loops: the example in Fig. 1, Histgram which computes a histogram for gray scale images and is also used in Ref. [8] and sparse matrix vector multiply (SMVM) which computes the product of a sparse matrix in the coordinate format and a vector, and all the benchmarks have loop-carried RAW dependences.

We applied loop pipeling with a commercial high-level synthesis (HLS) tool to the original benchmark code, the code optimized by the previous technique [8] and the code optimized by the proposed technique followed by logic synthesis. The clock constraints for both HLS and logic synthesis were set to 250 MHz and the target cell library was a 45 nm library. Arrays were mapped to synchronous memories with 1 clock cycle delay for both read and write accesses. Since we aim to minimize IIs, we allocated enough functional units and memory ports in synthesized hardware to achieve the minimum IIs. For example, we allocated a dual-port memory for array A for the example in Fig. 1 to achieve II=1. For all benchmarks, array sizes, and hence memory sizes, were the same, so the memory area was not included in the gate count results in **Table 1**.

4.2 Results and Discussions

Table 1 shows the experimental results for the 3 benchmarks. There are two results for Histgram, namely, Histgram (*sunflower*) and Histgram (*night scene*) with different input images (sunflower and night scene, respectively). The sizes of both images were the same size of 640x480. The differences in input images do not affect IIs, critical path delays and gate counts. In the table, *original*, *stall*, *proposed* show the synthesis results after loop pipelining and logic synthesis for the original benchmark code, the code optimized by the previous technique [8] and the code optimized by the proposed technique, respectively. In the table, the last 3 rows show the average results for the 3 benchmarks. In *original* the IIs were larger than those of *stall* and *proposed* for all the benchmarks because of pessimistic handling of loop-carried RAW possible dependences. Both the previous techniques [8] and the proposed technique could successfully reduce IIs by 69% and 61% on average compared to the case without the optimizations (*original*). For SMVM, both *stall* and *proposed* could not reduce the IIs to 1.

Although the IIs for the proposed technique were the same as the IIs for the previous technique [8] except SMVM, the numbers of execution cycles for Ref. [8] were 75% larger on average than those for the proposed technique, since pipeline stalls arose by Ref. [8] and the numbers of execution cycles were increased. As for Histgram, pipeline stalled more frequently for image *night scene* than for image *sunflower*. Since the image *night scene* has much more successive (black) pixels with the same intensity compared to the image *sunflower*, the possible dependences between successive iterations, and hence pipeline stalls, occured more frequently for the image *night scene* than for the image *sunflower*. For all benchmarks, the clock period constraints of 250 MHz were satisfied.

The total gate counts of *stall* and *proposed* increased by 77% and 104% on average compared to *original*. These increases in the total gate counts is mainly caused by the increase of functional units and registers due to the parallel processing by loop piplining. In addition, comparators to check the occurrences of possible dependences also contribute to the increases in the gate counts. Compared to *stall*, *proposed* has 15% more total gate counts on average. This increase in gate counts is mainly due to the fact that the proposed technique needs to keep both subscripts and data in shift registers to perform forwarding, while the previous technique [8] only need to keep subscripts in shift registers to perform pipeline stall.

Since the proposed technique could reduce the numbers of execution cycles by 34% on average compared to the previous work [8] that is the state-of-the-art with the moderate increases in gate counts by 15% on average without violating the clock constraints, we claim that the proposed technique is effective for quickly generating high-performance hardware accelerators.

5. Conclusion

In this paper, we proposed a source-level transformation that reduces IIs in the loop pipelining when loops contain loop-carried RAW possible dependences whose occurrences can be determined only at runtime. In the transformed code by the proposed transformation, the possible dependences are checked at runtime

and the written data to the arrays in the possible dependences are also written to scalar variables and the scalar variables are accessed instead of the arrays when the possible dependences actually occur. The proposed technique replaces slow array accesses in the possible dependences with fast scalar variable accesses, so that the numbers of execution cycles are reduced without pipeline stalls. The proposed technique automatically generates forwarding units at behavioral level. We implemented a tool that automatically performs the source-level transformation and applied the tool to three benchmark loops. The transformed code were synthesized to gates with loop pipelining in HLS and logic synthesis for evalutation. The proposed technique could reduce the numbers of execution cycles by 34% on average compared to the state-of-the-art previous work with the moderate increases in gate counts by 15% on average, so we claim that the proposed technique is effective for quickly generating high-performance hardware accelerators with high-level synthesis.

References

[1] Gajski, D.D. et al.: *High Level Synthesis: An Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

[2] Aho, A.V., Lam, M.S., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Second Edition, Prentice Hall (2006).

[3] Snider, G.: Performance-Constrained Pipelining of Software Loops onto Reconfigurable Hardware, *FPGA '02 Proc. 2002 ACM/SIGDA 10th International Symposium on Field-programmable Gate Arrays*, pp.177–186 (2002).

[4] Gao, L., Zaretsky, D., Mittal, G., Schonfeld, D. and Banerjee, P.: A Software Pipelining Algorithm in High-Level Synthesis for FPGA Architectures, *ISQED '09 Proc. 2009 10th International Symposium on Quality of Electronic Design*, pp.297–302 (2009).

[5] Morvan, A., Derrien, S. and Quinton, P.: Efficient Nested Loop Pipelining in High Level Synthesis using Polyhedral Bubble Insertion, *IEEE International Conference on Field-Programmable Technology*, pp.1–10 (2011).

[6] Kondratyev, A., Lavagno, L., Meyer, M. and Watanabe, Y.: Realistic performance-constrained pipelining in high-level synthesis, *Design, Automation & Test in Europe Conference & Exhibition*, pp.1382–1387 (2011).

[7] Ravi, S., Lakshminarayana, G. and Jha, N.K.: Removal of memory access bottlenecks for scheduling control-flow intensive behavioral descriptions, *ICCAD* (1998).

[8] Alle, M., Morvan, A. and Derrien, S.: Runtime Dependency Analysis for Loop Pipelining in High-Level Synthesis, *Design Automation Conference* (2013).

[9] Patterson, D.A. and Hennessy, J.L.: *Computer Organization & Design: The Hardware/Software Interface*, Fourth Edition, Morgan Kaufmann (2008).

Shingo Kusakabe received his B.S. in electrical and electronic engineering from Tokyo City University in 2014. He is currently working toward M.S. degree in the same university. He is a student member of IPSJ.

Kenshu Seto received his B.S. in electrical engineering, M.S. and D.Eng. in electronics engineering from the University of Tokyo in 1997, 1999 and 2004, respectively. From 2004 to 2006, he was a researcher at VLSI Design and Education Center (VDEC), the University of Tokyo. He joined the department of electrical and electronic engineering, Tokyo City University (renamed from Musashi Institute of Technology) in 2007. His primary research interests include high-level synthesis and compiler techniques for System-on-Chips (SoCs).

(Recommended by Associate Editor: *Takashi Takenaka*)