IPSJ Transactions on System LSI Design Methodology Vol.12 13–21 (Feb. 2019)

[DOI: 10.2197/ipsjtsldm.12.13]

Regular Paper

Scalar Replacement with Circular Buffers

Kenshu Seto[1,a)]

Received: May 28, 2018, Revised: September 4, 2018,
Accepted: October 22, 2018

Abstract: Scalar replacement is one of effective array access optimizations that can be applied before High-level synthesis (HLS). The successful application of scalar replacement removes local memories, and as a result, it decreases hardware area. In addition, scalar replacement reduces the numbers of hardware execution cycles by reducing memory access conflicts. In scalar replacement, shift registers are introduced to remove local arrays, and reuse distances corresponds to the lengths of the shift registers. Previous scalar replacement methods implement the shift registers with chains of registers, so that the hardware area becomes large when the reuse distances are large. In addition, when reuse distances are unknown at compile time, previous scalar replacement methods require multiplexers with large numbers of inputs, which further increase on hardware area. In this paper, we propose a new technique to resolve the issues. In particular, we implement the shift registers with circular buffers instead of chains of registers. Large shift registers implemented by RAM-based circular buffers are more compact than those implemented by the chains of registers. We also show that the proposed method requires no multiplexers to realize scalar replacement for loops with statically unknown reuse distances, which leads to area-efficient hardware implementation. We developed a tool that implements the method and applied the tool to the benchmark programs which require large shift registers or have statically unknown reuse distances. We found that the hardware area is reduced with the proposed method compared to the previous method without sacrificing the hardware performance. We conclude that the proposed method is an area efficient scalar replacement method for programs that have large or unknown reuse distances at compile time.

Keywords: high level synthesis, scalar replacement, shift register, circular buffer, loop pipelining

1. Introduction

High-level synthesis (HLS) [1], [2] automatically generates RTL descriptions from C programs, so it significantly reduces the hardware design time. HLS maps large arrays to RAMs, and such RAMs usually provides limited numbers of ports such as 1 port or 2 ports. So, intensive accesses to specific arrays in C programs lead to access conflicts to the corresponding RAMs, which degrade the performance of the generated hardware. To generate efficient hardware with HLS without memory access conflicts, array access optimizations are usually applied to input C programs. Memory partitioning [3], [4], [5], [6] is one of such array access optimizations, and it partitions an array into smaller arrays to increase the number of ports.

Scalar replacement is another array access optimization that exploits data reuse among array accesses. In C programs with loops that contain accesses to arrays (or RAMs), it is common that an array access reuses the data by another array access after a constant amount of loop iterations. The constant amount of the loop iterations is called reuse distance. In scalar replacement, data that will be reused in later loop iterations are stored in a shift register whose length is the maximum value of the reuse distances for all reusing array accesses. When an array access can always reuse data in the shift register, the array access is replaced by the shift register access. As a result, scalar replacement resolves access conflicts to the RAM by the use of the shift register. If scalar replacement successfully removes all the array accesses to target local arrays, we can completely remove the access conflicts to the corresponding RAMs. The removal of the access conflicts leads to performance improvement. In addition, after the successful application of scalar replacement, we can completely replace all the target local arrays, or the corresponding RAMs with the shift registers. Since the numbers of elements in the shift registers are typically much smaller than the numbers of elements in the corresponding local arrays, significant reduction in hardware area is achieved. So, the successful application of scalar replacement brings significant benefits in hardware designs with HLS as discussed in the previous work [7].

The pioneering work on scalar replacement [8] focused on exploiting data reuse carried by innermost loops. In the work, the code transformation called unroll-and-jam is necessary to exploit data reuse carried by outer loops. Unfortunately, unroll-and-jam increases the code size that will lead to the degradation of the clock frequency of generated hardware. In Ref. [9], scalar replacement for nested loops was proposed and it does not require unroll-and-jam. In Ref. [7], the scalar replacement algorithm in Ref. [9] was extended with polyhedral model to handle array accesses with constant subscripts. In Ref. [10], array SSA is used to perform scalar replacement in the LLVM compiler infrastructure.

All the previous methods including [7], [8], [9], [10] implement the shift registers by chains of registers, so that they have two problems. One problem is the large hardware area due to long shift registers that are implemented by the chains of registers. When the lengths of the shift registers become longer, the shift

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

registers occupy the large portion of the hardware area. So, it is important to reduce the area of the shift registers to generate compact hardware with HLS. Another problem is the introduction of large-input multiplexers when the reuse distances of reusing array accesses are statically unknown, in other words, unknown at compiler time. When reuses are carried by outer loops in a nested loop, and the numbers of loop iterations of the inner loops contain parameters, for example, N, which are unknown at compile time, the reuse distances include the parameter N and also become unknown at compile time. When the parameter N takes various values, the reusing accesses require large-input multiplexers that select appropriate elements in the shift registers. Such large multiplexers can significantly increase hardware area. In this paper, we propose a method that addresses the above two problems by implementing the shift registers with RAM-based circular buffers instead of chains of registers.

The organization of this paper is as follows. In Section 2, we explain the problems of the previous scalar replacement methods. In Section 3, we present our proposed method for scalar replacement that generates circular buffers to implement the shift registers, followed by experimental results in Section 4. Finally, we conclude this paper in Section 5.

2. The problems of the previous scalar replacement methods

In this section, we highlight the problems of the previous scalar replacement methods [7], [8], [9], [10] with example C programs. Although we use the previous scalar replacement algorithm [7] for explanation, the problems are not limited to Ref. [7] but are common to other scalar replacement methods [8], [9], [10].

2.1 Preliminaries

Before describing the problems, we define some terminologies that are necessary to understand the problems. In this paper, for simplicity, we call static array accesses (array accesses that appear in program texts) array access.

As in Ref. [7], we assume input C programs are stencil computations [5] in Static Control Part (SCoP) format [11]. We also assume that the input C program consists of a fully nested loop as shown in **Fig. 1** where all statements are contained in the innermost loop and no statement exists outside the innermost loop. For example, when statements exist between lines 1 and 2, or between lines 9 and 10 in Fig. 1, the loop is still a nested loop but not a fully nested loop. We also assume that each target array is accessed only inside the loop body, that each target array has only one write access and at least one read access in the loop body, and that the increment value of each loop index is 1.

```
 1 for(y=0; y<=31; y++){
 2   for(x=0; x<=31; x++){
 3     A[y][x] =……;
 4     if (x>=2)    t+= A[y][x-2];
 5     if (y>=1) {
 6       if (y<=2)  t+= A[0][x];
 7       else       t+= A[y-3][x];
 8     }
 9   }
10 }
```

Fig. 1 Example code with large reuse distances.

Definition 2.1 (Reuse) When an array element accessed by an array access s will be accessed later by an array access d, we say that the array access d reuses the data accessed by the array access s, or we say that there exists a **reuse** from s to d. We call s a **reuse source** and d a **reuse destination**. An array access can be both a reuse source and a reuse destination at the same time.

Definition 2.2 (Generator) When an array access is a reuse source but is not a reuse destination, the access is called a **generator**. A generator s starts to access each array element and the data accessed by the generator will be reused later by reuse destinations d.

Definition 2.3 (Reuse vector) A **reuse vector** $\langle d_1, d_2, \ldots \rangle$ represents the numbers of loop iterations (or simply, iterations) between the access to an array element by a reuse source s and the later access to the same element by a reuse destination d. In the reuse vector, $d_1, d_2, \ldots$ correspond to iterations for outermost loop, 2nd outermost loop, …, respectively. More specifically, an element d_i in a reuse vector $\langle d_1, d_2, \ldots \rangle$ means a difference $I_{d,i} - I_{s,i}$ between i-th loop iterator value $I_{d,i}$ for a reuse destination d and i-th loop iterator value $I_{s,i}$ for a reuse source s when the reuse occurs. In this paper, we assume each element of reuse vectors is a constant value.

Definition 2.4 (Reuse distance) A **reuse distance** is calculated from a reuse vector $\langle d_1, d_2, \ldots, d_n \rangle$ and it means the number of innermost loop iterations in which the reuses from s to d occur. For a given reuse vector $\langle d_1, d_2, \ldots, d_n \rangle$, the reuse distance is computed by the following formula where I_k represents the number of loop iterations for the k-th loop from the outermost loop.

$$\sum_{l=1}^{n-1} \left\{ \left(\prod_{k=l+1}^{n} I_k \right) \times d_l \right\} + d_n \tag{1}$$

2.2 The previous scalar replacement [7] and the problems

Given the input C program shown in Fig. 1, the previous work [7] builds the reuse information table shown in **Table 1** which is a key data structure in the previous work. The C program in Fig. 1 contains 4 array accesses to the array A and the write access A[y][x] in the line 3 is the generator. The previous work [7] assumes that there is a single generator for each reuse information table. As shown in Table 1, the data accessed by the generator A[y][x] is reused by the three array accesses A[y][x-2], A[0][x] and A[y-3][x]. The reuse destination A[0][x] has different reuse distances according to the different reuse conditions as shown in Table 1, so we separate the reuses by A[0][x] into two parts according to the different reuse distances. Since the number of the inner-loop iterations is 32, the maximum reuse distance by A[y-3][x] becomes 96 according to the formula (1).

Table 1 The reuse information table for Fig. 1 by the previous method [7].

	Access type	Array access	Reuse vector	Reuse distance	Scalar variable	Reuse condition
0	Generator	A[y][x]	N/A	N/A	s0	N/A
1	Reuse	A[y][x-2]	⟨0, 2⟩	2	s2	always
2	Reuse	A[0][x]	⟨1, 0⟩	32	s32	y==1
3	Reuse	A[0][x]	⟨2, 0⟩	64	s64	y==2
4	Reuse	A[y-3][x]	⟨3, 0⟩	96	s96	always

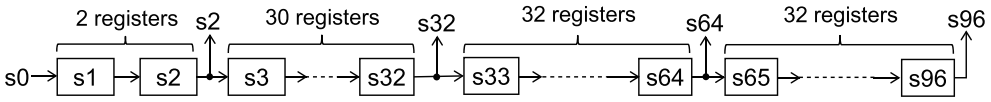

Fig. 2 Shift register implemented by previous scalar replacement methods for the C program in Fig. 1.

```
 1 for(y=0; y<=31; y++){
 2   for(x=0; x<=31; x++){
 3     s0=……;
 4     if (x>=2)  t+= s2;
 5     if (y>=1) {
 6       if (y<=2) {
 7         if (y==1)       s=s32;
 8         else if (y==2) s=s64;
 9         t += s;
10       }
11       else  t+=s96;
12     }
13     s96 = s95
14     s95 = s94
15     s94 = s93
       …
107    s2 = s1;
108    s1 = s0;
109   }
110 }
```

Long shift register

Fig. 3 Code after applying the previous scalar replacement [7] to Fig. 1.

```
 1 for(y=0; y<=N; y++){
 2   for(x=0; x<=M; x++){
 3     A[y][x] =……;
 4     if (x>=2)    t+= A[y][x-2];
 5     if (y>=1) {
 6       if (y<=2)  t+= A[0][x];
 7       else       t+= A[y-3][x];
 8     }
 9   }
10 }
```

Fig. 4 Example code with parameters N and M in loop counts.

Table 1 shows the names s0, s2, s32, . . ., of scalar variables that will replace the corresponding array accesses after scalar replacement. For each target array, the names of such scalar variables are uniquely generated with the corresponding reuse distances.

After building the reuse information table in Table 1, the previous method [7] performs the following code transformations:

(1) Insert the declaration of the shift registers and their shifting behavior at the bottom of the loop body

(2) Replace each array access with the corresponding scalar variables with the conditions.

The result of the code transformation is shown in **Fig. 3**. Since the maximum reuse distance is 96 in Table 1, the previous method [7] prepairs the shift register with the length of 96. In the previous method, the shift register is implemented by a set of 96 scalar variables whose shift behavior is described by the chain of 96 assignments to the scalar variables (or registers) at the end of the loop body from the line 13 to the line 108 shown in Fig. 3. **Figure 2** depicts the generated shift register. In the shift register, we use the outputs of the registers s2, s32, s64, s96 to replace the corresponding array accesses. The shift register generated by this way occupies the large portion of the area in the generated hardware.

In real-life C programs, we commonly have loops with unknown numbers of iterations at compile time as shown in **Fig. 4**.

Table 2 The reuse information table for Fig. 4.

	Access type	Array access	Reuse vector	Reuse distance	Scalar variable	Reuse condition
0	Generator	A[y][x]	N/A	N/A	s0	N/A
1	Reuse	A[y][x-2]	⟨0, 2⟩	2	s2	always
2	Reuse	A[0][x]	⟨1, 0⟩	M+1	sM	y==1
3	Reuse	A[0][x]	⟨2, 0⟩	2(M+1)	s2M	y==2
4	Reuse	A[y-3][x]	⟨3, 0⟩	3(M+1)	s3M	always

```
 1 for(y=0; y<=N; y++){
 2   for(x=0; x<=M; x++){
 3     s0=……;
 4     if (x>=2)  t+= s2;
 5     if (y>=1) {
 6       if (y<=2) {
 7         if (M==31)       sM = s32;
 8         else if (M==30) sM = s31;
 9         else if (M==29) sM = s30;
           …
30         else if (M==8)   sM = s9;
31         if (M==31)       s2M = s64;
32         else if (M==30) s2M = s62;
33         else if (M==29) s2M = s60;
           …
54         else if (M==8)   s2M = s18;
55       if (y==1)       s=sM;
56       else if (y==2) s=s2M;
57       t += s;
58     } else {
59       if (M==31)       s3M = s96;
60       else if (M==30) s3M = s93;
61       else if (M==29) s3M = s90;
           …
82       else if (M==8)   s3M = s27;
83       t+=s3M;
84     }
85     s96 = s95
86     s95 = s94
87     s94 = s93
       …
179    s2 = s1;
180    s1 = s0;
181   }
182 }
```

3 Multiplexers with 24 inputs

Fig. 5 Code after applying the previous scalar replacement [7] to Fig. 4.

In Fig. 4, the loop bounds for the outer loop and the inner loop are represented by the parameters N and M, respectively. The computation of the program in Fig. 4 is exactly the same as the program in Fig. 1. We assume that we know the maximum values of the parameters at compiler time but the values of the parameters can vary at runtime for each instance of the loop execution. For the example in Fig. 4, we assume the maximum value is 31 and the minimum value is 8 for both N and M. The reuse information table built by the previous method [7] is shown in **Table 2**. As shown in Table 2, the values of the reuse distances for the reuse destinations A[0][x] and A[y-3][x] depend on the value of the parameter M. Since M takes one of the 24 values from 8 to 31, each reuse distance of these reuse destinations takes one of 24 values, so that we have to generate a program that select an appropriate register from the 24 possible registers for these reuse destinations as shown in **Fig. 5**. The generated hardware with HLS from the

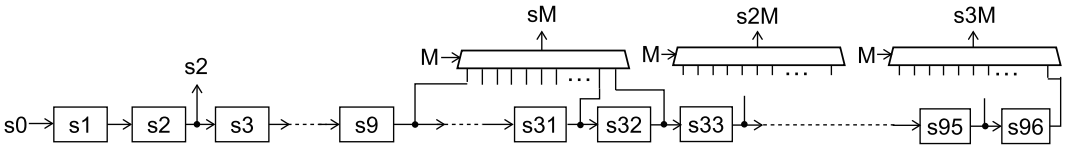

Fig. 6 Shift register and multiplexers implemented by previous scalar replacement methods for the C program in Fig. 4.

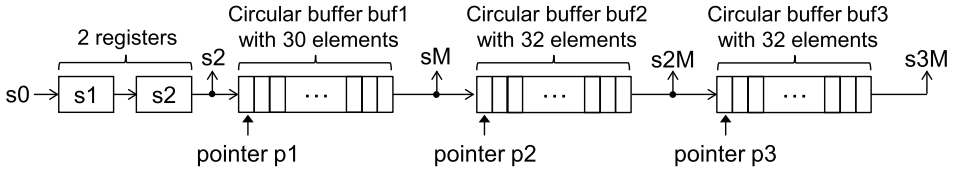

Fig. 7 Shift register implemented with circular buffers by the proposed scalar replacement for the C program in Fig. 4.

program in Fig. 5 is shown in **Fig. 6** and it contains 3 large multiplexers with 24 inputs which significantly increase the hardware area.

3. Proposed scalar replacement method with circular buffers

In this section, we present the proposed scalar replacement method in order to solve the problems highlighted in Section 2 and illustrate the proposed method with the example in Fig. 4. The proposed method is directly applicable to the programs with constant loop counts, as shown in Fig. 1 as well. As far as the author knows, all the previous approaches for scalar replacement [7], [8], [9], [10] describe the behavior of a shift register as a chain of assignments to scalar variables that represent registers as shown in the line 85 to the line 180 in Fig. 5. Instead of implementing a shift register by a single chain of registers as shown in Fig. 5 or as illustrated in Fig. 6, we propose a method that implements the shift register by an appropriate combination of circular buffers and chains of registers as shown in **Fig. 7**. The proposed descriptions of circular buffers can be directly implemented by RAMs with HLS. Since RAMs are typically more area-efficient than registers for storing large numbers of bits, we can expect the reduction in hardware area with the proposed approach. In addition, the proposed method removes all the multiplexers in Fig. 6 since we can flexibly change the lengths of shift registers implemented by circular buffers by changing the reset conditions of the pointers for the circular buffers. In the following, we describe the detail of the proposed method.

The proposed method uses the same method as the previous method [7] to build the reuse information table as shown in Table 2, although we extended the tool in Ref. [7] in order to handle parameters such as N and M in the input C programs using ISL (Integer Set Library) [12], [13]. In addition, we extended the tool [7] so that it can read the maximum values of the parameters as command line options. Other scalar replacement methods can also be used as long as they generate the reuse information tables as shown in Fig. 2. In the reuse information table, reuse vectors, reuse distances and scalar variable names are important, and other entries such as reuse conditions are not used in the proposed method for generating shift register descriptions. In this work, we assume that the reuse information tables do not change when the values of parameters change within their ranges of the minimum and maximum values.

Table 3 The partitioned shift registers table for the C program in Fig. 4 by the proposed method.

ID	Input variable	Output variable	Length	MaxLength
0	s0	s2	2	2
1	s2	sM	M-1	30
2	sM	s2M	M+1	32
3	s2M	s3M	M+1	32

The proposed method performs the following code transformations based on the reuse information table.

(1) Insert the declaration of the shift registers and their shifting behavior at the bottom of the loop body

(2) Replace each array access with the corresponding scalar variables with the conditions.

As for the second transformation (2), the proposed method is the same as the previous method [7]. On the other hand, the first transformation (1) of the proposed method is different from that of the previous method [7] which has been explained in Section 2.2. In the following, we will explain the first transformation (1) of the proposed method which utilizes circular buffers in shift registers.

Intuitively, the proposed method partitions the shift register shown in Fig. 2 into the sequence of smaller shift registers by using the intermediate scalar variables, s2, s32 and s64 as separators. Each of the partitioned shift registers is implemented as a chain of registers or as a circular buffer as shown in Fig. 7. In the proposed method, we use a table as shown in **Table 3** where each row represents a partitioned shift register. We call the table partitioned shift registers table. The partitioned shift registers table is important, since it contains all information necessary for generating circular buffer descriptions. As shown in Table 3 from the left, each row of partitioned shift registers tables contains an ID number (ID), an input variable name (Input variable), an output variable name (Output variable), a length possibly with parameters (Length), and the maximum length (MaxLength) for a partitioned shift register.

Figure 8 shows the proposed algorithm to build the partitioned shift registers tables such as the one shown in Table 3. One of the two inputs to the algorithm is a reuse information table T_{reuse} as shown in Table 2. In the Table 2, we set the reuse vector and reuse

Algorithm *BuildPartitionedShiftRegistersTable*

```
BuildPartitionedShiftRegistersTable(T_reuse, Max)
   Input   T_reuse  : A reuse information table
   Input   Max      : Max values for parameters
   Output  T_sreg   : A partitioned shift registers table
 1: T_tmp1 ← SortRowsByReuseVector(T_reuse)
 2: T_tmp2 ← RemoveRowsWithSameReuseVector(T_tmp1)
 3: id ← 0
 4: for (i = 1; i < size(T_tmp2); i++) {
 5:    inputVar  ← T_tmp2[i − 1].ScalarVariable
 6:    OutputVar ← T_tmp2[i].ScalarVariable
 7:    Length    ← T_tmp2[i].ReuseDistance−T_tmp2[i−1].ReuseDistance
 8:    if (hasParameters(Length))
 9:       MaxLength ← gist(Length, Max)
10:    else
11:       MaxLength ← Length
12:    AddElement(T_sreg, id, inputVar, outputVar, Length, MaxLength)
13:    id ← id + 1
14: }
```

Fig. 8 Procedure for building a partitioned shift registers table from a reuse information table.

distance for the generator to ⟨0, 0⟩ and 0, respectively, instead of N/As. Another input to the algorithm in Fig. 8 is the maximum values *Max* of parameters that appears in the input C program, such as N = 31 and M = 31. In the line 1 in Fig. 8, we sort the rows of the input reuse information table T_{reuse} in the ascending lexicographic order in terms of the reuse vectors and store the result to the table T_{tmp1}. The reuse information table shown in Table 2 is already sorted in such an order. In the line 2 in Fig. 8, we remove duplications in rows in terms of reuse vectors in the sorted table T_{tmp1} and store the result to the table T_{tmp2}. In other words, in the line 2 in Fig. 8, we remove rows that have the same reuse vector as other rows in the table T_{tmp1}. As a result, the table T_{tmp2} has rows whose reuse vectors are ordered in the ascending lexicographic order and has no two rows that have the same reuse vector. The table in Table 2 is already ordered in such a way and has no row with the same reuse vector as other rows. In general, however, input reuse information tables may have unordered rows and have rows with the same reuse vectors as other rows. In the line 3 in Fig. 8, we initialize the ID number that is attached to each partitioned shift register. From the 1st row to the final row in the table T_{tmp2}, we repeat the processing from the line 5 to the line 13 in Fig. 8 to generate a row in the partitioned shift register table in order as shown in Table 3. We denote the i-th row of a table T as $T[i]$. In the line 5 in Fig. 8, we set the scalar variable name of the previous row $T_{tmp2}[i-1]$ to the input variable name for the row $T_{sreg}[i-1]$ of the partitioned shift register table T_{sreg}. Similarly in the line 6, we set the scalar variable name of the current row $T_{tmp2}[i]$ to the output variable name for the row $T_{sreg}[i-1]$ of the partitioned shift register table T_{sreg}. In the line 7, the length of the partitioned shift register $T_{sreg}[i-1]$ is computed by subtracting the reuse distance of the previous row $T_{tmp2}[i-1]$.*ReuseDistance* from the reuse distance $T_{tmp2}[i]$.*ReuseDistance* of the current row $T_{tmp2}[i]$ of the sorted table T_{tmp2}. If the computed length contains parameters, we substitute the maximum values in *Max* to the parameters by using the gist operation in ISL [12], [13] in order to compute the maximum length of the shift registers as shown in the line 9 in Fig. 8. If the computed length is a constant value and does not contain parameters, the length is used as the maximum length of the shift register as shown in the line 11 in Fig. 8. Finally, the computed row of the partitioned shift registers table is stored in T_{sreg} and the ID number is incremented by one as shown in the lines 12 and 13 in Fig. 8.

Algorithm *GenerateShiftRegisterImplementation*

```
GenerateShiftRegisterImplementation(T_sreg, Threshold)
   Input   T_sreg     :  A partitioned shift registers table
   Input   Threshold  :  Integer
   Output  Code fragment that implements the shift register
 1: for (i = size(T_sreg) − 1; i >= 0; i − −) {
 2:    MaxLength ← T_sreg[i].MaxLength
 3:    if (MaxLength == 1)
 4:       generateSimpleAssignment(T_sreg[i])
 5:    if (MaxLength >= Threshold)
 6:       generateCircularBuffer(T_sreg[i])
 7:    else
 8:       generateRegisters(T_sreg[i])
 9: }
```

Fig. 9 Procedure for generating implementations of shift registers from a partitioned shift registers table.

```
 1 for(y=0; y<=N; y++){
 2   for(x=0; x<=M; x++){
 3     s0=……;
 4     if (x>=2)  t+= s2;
 5     if (y>=1) {
 6       if (y<=2) {
 7         if (y==1)       sM_s2M=sM;
 8         else if (y==2) sM_s2M=s2M;
 9         t += sM_s2M;
10       }
11       else  t+=s3M;
12     }
13   s3M=buf3[p3];                          ③
14   buf3[p3]=s2M;
15   p3++;
16   if(p3==M+1) p3=0;
17   s2M=buf2[p2];                          ②   Partitioned shift registers
18   buf2[p2]=sM;                               implemented by
19   p2++;                                      3 circular buffers
20   if(p2==M+1) p2=0;
21   sM=buf1[p1];                           ①
22   buf1[p1]=s2;
23   p1++;
24   if(p0==M-1) p1=0;
25    s2 = s1;                              ⓪   A partitioned shift register
26    s1 = s0;                                   implemented by
                                                 a chain of 2 registers
27 }
28 }
```

Fig. 10 Code after applying the proposed scalar replacement to Fig. 4.

Figure 9 shows the proposed algorithm to generate code fragment for the partitioned shift register behaviors as shown from the line 13 to the line 26 in **Fig. 10**. One of the inputs to the algorithm in 9 is the partitioned shift registers table T_{sreg} that is built by the algorithm in Fig. 8. Another input is a user-specified integer *Threshold* which is larger than 1. The value of *Threshold* is used to switch the implementations of the partitioned shift registers. Since the implementation of a circular buffer requires additional hardware such as a pointer register, a comparator, an incrementer, sense amplifiers and decoders, the partitioned shift registers whose lengths are short will be implemented more efficiently with chains of registers rather than with circular buffers. In addition, RAMs in ASICs usually have the minimum size re-

quirements. In the proposed algorithm, partitioned shift registers whose maximum lengths are greater than or equal to $Threshold$ are implemented as circular buffers, while those whose maximum lengths are less than $Threshold$ are implemented by chains of registers. In this paper, we set the $Threshold$ to 16 in the experiments in Section 4.

The algorithm in Fig. 9 processes each row of the partitioned shift registers table T_{sreg} in the reverse order, namely, from the bottom row to the top row. Depending on the MaxLength of the row $T_{sreg}[i]$ in the partitioned shift registers table T_{sreg}, the proposed algorithm generates different types of implementations for each partitioned shift register. If the MaxLength is larger than or equal to $Threshold$, a circular buffer is generated as shown from the line 13 to the line 16 in Fig. 10. On the other hand, if the MaxLength is smaller than the $Threshold$, a chain of registers is generated as shown in the lines 25 and 26 in Fig. 10. When the MaxLength is 1, a simple assignment such as
$T_{sreg}[i].OutputVar = T_{sreg}[i].InputVar$;
is generated, where the left hand side (LHS) is the output variable of $T_{sreg}[i]$ and the right hand side (RHS) is the input variable of $T_{sreg}[i]$. When the MaxLength is 0, the value of the generator is reused in the same iteration. So, we do not generate any description for shift registers in such a case.

Figure 10 shows the optimized code after applying the proposed scalar replacement method and Fig. 7 illustrates the HLS-generated shift register from the code in Fig. 10. Differently from the HLS-generated shift register with the previous method as shown in Fig. 6, the HLS-generated shift register with the proposed method in Fig. 7 requires no multiplexer.

In Fig. 10, the code fragments ③, ②, ① and ⓪ correspond to the rows whose IDs are 3, 2, 1 and 0 in Table 3, respectively. For example, the code fragment ③ in Fig. 10 corresponds to the row whose ID is 3 (3rd row) in Table 3. Since the code fragment ③ in Fig. 10 corresponds to the row with ID = 3, we use the symbol buf3 and p3 for the array variable corresponding to the circular buffer and the pointer for the circular buffer, respectively. Although not shown in Fig. 10, buf3 is declared as an array variable whose size is 32 as specified in the column MaxLength in Table 3. In other words, the maximum number of elements in the circular buffer buf3 is set to 32 as specified in the column MaxLength in Table 3. buf3 as well as buf2 and buf1 are declared as 2-port memories in order to perform loop pipelining with the initiation interval (II) of 1. We focus on generating hardware whose inner loops are pipelined with the initiation interval (II) of 1.

The behavior of the code fragment ③ in Fig. 10 is explained as follows. The 4 lines of the code fragment from the line 13 to the line 16 in Fig. 10 are executed in a single cycle in parallel, since the inner loop is pipelined with the initiation interval (II) of 1. In the line 13, the result of the read access to the circular buffer buf3 is stored in the output variable s3M. In the line 14, the circular buffer buf3 stores the value of the input variable s2M. In the line 15, the pointer p3 for the circular buffer is incremented. In the line 16, the pointer p3 of the circular buffer is reset to 0 when the pointer exceeds the length of the circular buffer buf3. When the value of the parameter M changes at runtime, the line 16 in Fig. 10 changes the length of the circular buffer automatically in accordance with the changed value of M.

4. Experimental results

In this section, we show the impacts of the proposed method on hardware performance and area. In particular, we compare the performance and area by the proposed method with those by the previous scalar replacement method [7].

4.1 Experimental setups

We implemented the proposed method in Section 3 based on the tool presented in Ref. [7]. To determine the $Threshold$ value in the algorithm shown in Fig. 9, we can generate chains of registers and circular buffers for different shift register lengths, synthesize the generated circuits and build a table that contains the circuit area of a chain of registers and of a circular buffer for a given shift register length. Based on this table, we can determine the $Threshold$ value as the shift register length when the circuit area of a circular buffer becomes less than that of a chain of registers. Unfortunately, we could not access memory compiler, so we used the following simple method to determine the $Threshold$ value. We assumed that the circuit area of a circular buffer is smaller than that of a chain of registers when the total bit size of the corresponding shift register is more than 512 bits. Since the variables stored in the shift registers have 32 bits in the benchmark programs used in this paper, the circuit area of a circular buffer is smaller than that of a chain of registers when the lengths of shift registers are more than 16. So, we set the $Threshold$ to 16 (except two cases with $prop.(Th = 64)$ in $loop4$ and $loop4\_par$ in **Table 4**).

We applied the proposed method and the previous method [7] to 12 benchmark programs, shown in Table 4, which are fully nested loops with the nested level of two. These benchmark programs are basically the same as the benchmark programs used in [7], but the loop counts were modified. More specifically, we set the loop counts of the inner and the outer loops to the constant value of 32 in this experiment. ex is the example code shown in Fig. 1. ex_d60 and $ex1$ are modified versions of ex and have different reuse distances compared to ex. $ex2$ is a 1-dimensional filter code. $filter$ is a 2-dimensional filter code which originally consists of two loops but fused into a single loop with loop fusion [14]. $loop4$ is a real-life image processing application originally consisting of four loops, each of which performs image difference, vertical 1-dimensional filtering, horizontal 1-dimensional filtering, and image blending, respectively. In $loop4$, these four loops are also fused into a single fully nested loop with loop fusion [14]. In the benchmark programs whose names have $\_par$ as suffixes, such as ex_par, we set the loop counts of the inner loops to M which is unknown at compile time, but takes one of the values from 8 to 32 at runtime. In other words, the minimum and the maximum values of M were set to 32 and 8, respectively. The loop counts for the outer loops in the benchmark programs with the suffix $\_par$ were set to 32, since these loop counts of the outer loops do not affect the reuse distances or the scalar replacement results. Because the benchmark programs $ex1\_x5$ and $ex1\_x10$ used in Ref. [7] are similar to $ex1$ and are different from $ex1$ only in terms of the reuse conditions, we omitted

Table 4 Comparison between the previous method [7] and the proposed method when initiation interval (II) is 1.

Benchmark program	Type	# of execution cycles [cycles]	Gate counts (Computation) [gates]	Gate counts (Shift regs) [gates]	Gate counts [gates]	Total gate counts [gates]	# of shift register bits [bits]	# of circular buffer bits [bits]	# of I/O RAM bits [bits]
ex	*prev.*	1025	2991 (1.00)	19968 (1.00)	22959 (1.00)	72111 (1.00)	3072	0	32768
	prop.	1025	3927 (1.31)	9440 (0.47)	13367 (0.58)	62519 (0.87)	3072	3008	32768
ex_par	*prev.*	1024	7156 (1.00)	19968 (1.00)	27124 (1.00)	76276 (1.00)	3072	0	32768
	prop.	1024	4653 (0.65)	9440 (0.47)	14093 (0.52)	63245 (0.83)	3072	3008	32768
ex_d60	*prev.*	1025	4032 (1.00)	39936 (1.00)	43968 (1.00)	93120 (1.00)	6144	0	32768
	prop.	1025	4574 (1.13)	18432 (0.46)	23006 (0.52)	72158 (0.77)	6144	6144	32768
ex_d60_par	*prev.*	1024	9649 (1.00)	39936 (1.00)	49585 (1.00)	98737 (1.00)	6144	0	32768
	prop.	1024	5436 (0.56)	18432 (0.46)	23868 (0.48)	73020 (0.74)	6144	6144	32768
ex1	*prev.*	1025	2271 (1.00)	6656 (1.00)	8927 (1.00)	58079 (1.00)	1024	0	32768
	prop.	1025	2596 (1.14)	3408 (0.51)	6004 (0.67)	55156 (0.95)	1024	928	32768
ex1_par	*prev.*	1024	4411 (1.00)	6656 (1.00)	11067 (1.00)	60219 (1.00)	1024	0	32768
	prop.	1024	3561 (0.81)	3408 (0.51)	6969 (0.63)	56121 (0.93)	1024	928	32768
ex2	*prev.*	1090	3777 (1.00)	13312 (1.00)	17089 (1.00)	115393 (1.00)	2048	0	65536
	prop.	1090	4466 (1.18)	6144 (0.46)	10610 (0.62)	108914 (0.94)	2048	2048	65536
ex2_par	*prev.*	1090	6458 (1.00)	13312 (1.00)	19770 (1.00)	118074 (1.00)	2048	0	65536
	prop.	1090	4844 (0.75)	6144 (0.46)	10988 (0.56)	109292 (0.93)	2048	2048	65536
filter	*prev.*	1122	6471 (1.00)	13312 (1.00)	19783 (1.00)	118087 (1.00)	2048	0	65536
	prop.	1122	7536 (1.16)	6256 (0.47)	13792 (0.70)	112096 (0.95)	2048	2016	65536
filter_par	*prev.*	1122	9330 (1.00)	13312 (1.00)	22642 (1.00)	120946 (1.00)	2048	0	65536
	prop.	1122	8224 (0.88)	6256 (0.47)	14480 (0.64)	112784 (0.93)	2048	2016	65536
loop4	*prev.*	1186	15812 (1.00)	41600 (1.00)	57412 (1.00)	204868 (1.00)	6400	0	98304
	prop.	1186	17225 (1.09)	19760 (0.48)	36985 (0.64)	184441 (0.90)	6400	6240	98304
	prop.(Th=64)	1186	15983 (1.01)	34096 (0.82)	50079 (0.87)	197535 (0.96)	6400	2144	98304
loop4_par	*prev.*	1186	23078 (1.00)	41600 (1.00)	64678 (1.00)	212134 (1.00)	6400	0	98304
	prop.	1186	18717 (0.81)	19760 (0.48)	38477 (0.59)	185933 (0.88)	6400	6240	98304
	prop.(Th=64)	1186	17207 (0.75)	34096 (0.82)	51303 (0.79)	198759 (0.94)	6400	2144	98304

them in this experiment.

We generated RTL code and gate-level netlists with a commercial high-level synthesis (HLS) tool (Stratus) and a commercial logic synthesis tool (Genus) from Cadence, respectively, from the code optimized by the previous method [7] and the code optimized by the proposed method. In HLS, we used the loop pipelining directives to all the innermost loops with the initiation intervals (IIs) of 1. The clock constraints for both the HLS and the logic synthesis were set to 500 MHz and we used a 45*nm* technology library for the target cell library. All arrays that contain input or output data, which correspond to input or output buffers, were mapped to 1-port RAMs, since these arrays are accessed only once in one iteration of the innermost loops. On the other hand, the arrays that correspond to circular buffers, which are accessed twice in one iteration of the innermost loops, were mapped to 2-port RAMs in order to achieve the initiation intervals (IIs) of 1 after loop pipelining.

4.2 Results and discussions

Table 4 shows the experimental results for the 12 benchmark programs. In the table, *prev.* and *prop.* show the synthesis results for the code optimized by the previous SR method [7] and the code optimized by the proposed SR method, respectively. For the benchmark programs *loop*4 and *loop*4*_par* with code type of *prop.* (*Th=64*), we used the $Threshold$ value of 64 instead of 16. All the designs satisfied the clock constraints of 500 MHz and achieved the initiation interval (II) of 1 for the innermost loops. Under the clock constraints of 500 MHz, the numbers of execution cycles for *prev.* and *prop.* were the same for all benchmark programs as shown in Table 4. In addition, we observe that the numbers of execution cycles in Table 4 are almost the same between the benchmark programs with constant loops counts, such as *ex*, and the corresponding benchmark programs with the unknown loop counts, such as *ex_par*. In summary, we found that the proposed scalar replacement method did not negatively impact the performance of the hardware generated with HLS.

Table 4 also shows gate counts in terms of NAND2 gates. In Table 4, we approximated the gate counts for 1-port RAMs such as the I/O RAMs by 1.5 NAND gates per 1 bit and those for 2-port RAMs such as RAMs implementing circular buffers by 3 NAND gates per 1 bit. As shown in the second column from the right (# of circular buffer bits) of Table 4, we see that the proposed method generates circular buffers while the previous method does not. The generated hardware by the proposed method and by the previous method both use the same amount of I/O RAMs (input and output buffers) as shown in the rightmost column. The sixth column from the left (Gate counts) shows the gate counts of the synthesized hardware including the gate counts due to the RAMs implementing the circular buffers but excluding the gate counts due to the I/O RAMs. In the column, the figures in the parentheses show the ratios of the gate counts by the proposed method against those by the previous method. The fourth column from the left (Gate counts (Computation)) and the fifth column from the left (Gate counts (Shift regs)) show the breakdown of the the sixth column from the left (Gate counts) and represent gate counts for computation parts and gate counts for shift register parts, respectively. The seventh column from the left (Total gate counts) shows all the gate counts in the synthesized hardware including both the gate counts due to the RAMs implementing circular buffers and the gate counts due to the I/O RAMs. As shown in Gate counts or Total gate counts in Table 4, the proposed method reduced the gate counts in all the benchmark programs compared to the previ-

Table 5 Comparison between the previous method [7] and the proposed method when initiation interval (II) is 2.

Benchmark program	Type	# of execution cycles [cycles]	Gate counts (Computation) [gates]	Gate counts (Shift regs) [gates]	Gate counts [gates]	Total gate counts [gates]	# of shift register bits [bits]	# of circular buffer bits [bits]	# of I/O RAM bits [bits]
ex	*prev.*	2113	3806 (1.00)	19968 (1.00)	23774 (1.00)	72926 (1.00)	3072	0	32768
	prop.	2113	5809 (1.53)	4928 (0.25)	10737 (0.45)	59889 (0.82)	3072	3008	32768
ex_par	*prev.*	2048	7235 (1.00)	19968 (1.00)	27203 (1.00)	76355 (1.00)	3072	0	32768
	prop.	2048	4690 (0.65)	4928 (0.25)	9618 (0.35)	58770 (0.77)	3072	3008	32768
ex_d60	*prev.*	2113	4833 (1.00)	39936 (1.00)	44769 (1.00)	93921 (1.00)	6144	0	32768
	prop.	2113	5656 (1.17)	9216 (0.23)	14872 (0.33)	64024 (0.68)	6144	6144	32768
ex_d60_par	*prev.*	2048	9601 (1.00)	39936 (1.00)	49537 (1.00)	98689 (1.00)	6144	0	32768
	prop.	2048	5546 (0.58)	9216 (0.23)	14762 (0.30)	63914 (0.65)	6144	6144	32768
ex1	*prev.*	2113	3135 (1.00)	6656 (1.00)	9791 (1.00)	58943 (1.00)	1024	0	32768
	prop.	2113	4051 (1.29)	2016 (0.30)	6067 (0.62)	55219 (0.94)	1024	928	32768
ex1_par	*prev.*	2048	3831 (1.00)	6656 (1.00)	10487 (1.00)	59639 (1.00)	1024	0	32768
	prop.	2016	3460 (0.90)	2016 (0.30)	5476 (0.52)	54628 (0.92)	1024	928	32768
ex2	*prev.*	2114	3871 (1.00)	13312 (1.00)	17183 (1.00)	115487 (1.00)	2048	0	65536
	prop.	2114	4632 (1.20)	3072 (0.23)	7704 (0.45)	106008 (0.92)	2048	2048	65536
ex2_par	*prev.*	2146	7579 (1.00)	13312 (1.00)	20891 (1.00)	119195 (1.00)	2048	0	65536
	prop.	2146	5757 (0.76)	3072 (0.23)	8829 (0.42)	107133 (0.90)	2048	2048	65536
filter	*prev.*	2115	6471 (1.00)	13312 (1.00)	19783 (1.00)	118087 (1.00)	2048	0	65536
	prop.	2114	7536 (1.16)	3232 (0.24)	10768 (0.54)	109072 (0.92)	2048	2016	65536
filter_par	*prev.*	2115	8942 (1.00)	13312 (1.00)	22254 (1.00)	120558 (1.00)	2048	0	65536
	prop.	2114	7384 (0.83)	3232 (0.24)	10616 (0.48)	108920 (0.90)	2048	2016	65536
loop4	*prev.*	2242	17350 (1.00)	41600 (1.00)	58950 (1.00)	206406 (1.00)	6400	0	98304
	prop.	2242	18853 (1.09)	10400 (0.25)	29253 (0.50)	176709 (0.86)	6400	6240	98304
loop4_par	*prev.*	2274	26388 (1.00)	41600 (1.00)	67988 (1.00)	215444 (1.00)	6400	0	98304
	prop.	2274	20925 (0.79)	10400 (0.25)	31325 (0.46)	178781 (0.83)	6400	6240	98304

ous method [7]. The proposed method reduced the Gate counts by 38% and 43% on average for the benchmark programs with the constant loop counts and those with the unknown loop counts, respectively. The reason for the reductions is because the proposed method implements the shift registers with RAM-based circular buffers instead of only with chains of registers. In addition, the proposed method gave further reductions in the case of the benchmark programs with the unknown loop counts compared to the case of the benchmark programs with the constant loop counts. This is because the proposed method did not generate 24-input multiplexers in the case of the benchmark programs with the unknown loop counts while the previous method generated the multiplexers in the same case. If we took the gate counts of the I/O RAMs into account, the proposed method reduced the Total gate counts by 10% and 13% on average for the benchmark programs with the constant loop counts and for those with the unknown loop counts, respectively. For the benchmark programs of *loop*4 and *loop*4*_par*, Table 4 also shows the experimental results when the $Threshold$ value was changed to 64. The two benchmark programs have four shift registers with the length of 32 and one shift register with the length of 67. In the case of $Threshold = 16$, all the shift registers were implemented as circular buffers. When we changed the $Threshold$ value to 64 from 16, only one shift registers with the length of 67 was implemented as a circular buffer and the other four shift registers were implemented by chains of registers which have large circuit area compared to circular buffers. So, the circuit area for the two benchmark programs *loop*4 and *loop*4*_par* was increased in the case of $Threshold = 64$ compared to the case when $Threshold = 16$ as shown in Table 4. When we compare the benchmark programs *ex_d*60 and *ex*1, for example, which perform similar computations but have different numbers of shift register bits (6,144 and 1,024, respectively), the reduction percentage of the gate counts for *ex_d*60 were larger than that for *ex*1, since *ex_d*60 has the larger number of shift register bits than *ex*1 does. In summary, we found that the proposed method reduced the gate counts compared to the previous method without sacrificing the hardware performance.

Table 5 shows the experimental results when the pipeline initiation interval (II) constraint was increased to 2 from 1. When the II is 2, the memory bandwidth requirement for the circular buffers can be relaxed, so we set the number of ports for the RAMs corresponding to the circular buffers to 1. For all benchmark programs, II = 2 was achieved and the clock constraints of 500 MHz were satisfied. The numbers of execution cycles for *prev.* and *prop.* were almost the same. In general, we have more chances of resource sharing with increased II, because increasing II results in less required parallelism. Since the benchmark C programs we used do not contain area-consuming computing units such as multipliers, the increased II did not consistently decrease the hardware area of the computing unit parts. Instead, the hardware area of the circular buffers was reduced by half from 3 NAND gates per 1 bit to 1.5 NAND gates per 1 bit since the number of ports for the circular buffers was reduced to 1. So, the reductions of the total gate counts in the case of II = 2 were larger than those in the case of II = 1. In the case of II = 2, the proposed method reduced the Gate counts by 52% and 58% on average for the benchmark programs with the constant loop counts and those with the unknown loop counts, respectively. In summary, we found that the proposed method reduced the gate counts compared to the previous method without sacrificing the hardware performance even in the case of II = 2.

When given application programs satisfy the assumptions described in Section 2.1, we can successfully remove target arrays by scalar replacement [7]. However, when the lengths of shift registers are less than the $Threshold$ value, the circular buffers proposed in this paper are not generated since chains of registers

are more compact than the circular buffers in such cases. So, the lengths of generated shift registers for a given application program should be more than the *Threshold* value for the proposed technique to be effective compared to the previous scalar replacement technique [7].

The proposed technique assumes that the given loop programs are in the form of stencil computations [5]. In stencil computations, dimensions of arrays must be the same as the nesting depth of a fully nested loop. In real-life applications, it is common that target C programs are not in the form of the stencil computations. Future work will be extending the proposed technique to non-stencil computations.

5. Conclusion

Scalar replacement is a memory access optimization for hardware area reduction and performance boost in HLS. Scalar replacement generates shift registers, which can occupy large portions of the hardware area when the shift registers are long. All of the previous scalar replacement methods implement the shift registers by chains of registers, but the implementation is not compact compared to the shift register implementation with RAMs. In addition, the previous methods generate large-input multiplexers when reuse distances vary widely due to loop counts unknown at compile-time. To resolve the problems, we proposed a method to partition the shift registers and implement each partitioned shift register either by a chain of registers or by a RAM-based circular buffer. We implemented the proposed method as a tool, and used the tool to perform experiments with benchmark programs. From the experiments, we found that the proposed method reduces the hardware area without negatively impacting the hardware performance compared to the previous method. We conclude that the proposed method is effective in reducing hardware area when scalar replacement generates long shift registers, or target programs contain loop counts that are unknown at compile time and vary widely.

References

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

[2] Vivado Design Suite User Guide: High-Level Synthesis (UG902), Xilinx (2017).

[3] Cong, J., Jiang, W., Liu, B. and Zou, Y.: Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization, *ACM Trans. Design Automation of Electronic Systems* (2011).

[4] Cong, J., Jiang, W., Liu, B. and Zou, Y.: Theory and algorithm for generalized memory partitioning in high-level synthesis, *International Symposium on Field-Programmable Gate Arrays* (*FPGA*) (2014).

[5] Cong, J., Li, P., Xiao, B. and Zhang, P.: An Optimal Microarchitecture for Stencil Computation Acceleration Based on Nonuniform Partitioning of Data Reuse Buffers, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.35, pp.407–418 (2016).

[6] Zhou, Y., Al-Hawaj, K. and Zhang, Z.: A New Approach to Automatic Memory Banking using Trace-Based Address Mining, *FPGA'17* (2017).

[7] Seto, K.: Scalar Replacement with Polyhedral Model, *IPSJ Trans. System LSI Design Methodology*, Vol.12 (2018).

[8] Callahan, D., Carr, S. and Kennedy, K.: Improving Register Allocation for Subscript Variables, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI*) (1990).

[9] So, B. and Hall, M.W.: Increasing the Applicability of Scalar Replacement, *Compiler Construction* (*CC*) (2004).

[10] Surendran, R., Barik, R., Zhao, J. and Sarkar, V.: Inter-iteration Scalar Replacement Using Array SSA Form, *International Conference on Compiler Construction* (2014).

[11] Bastoul, C.: Clan: A Polyhedral Representation Extractor for High Level Programs, available from ⟨http://icps.u-strasbg.fr/people/bastoul/public_html/development/clan/docs/clan.pdf⟩ (accessed 2018-05).

[12] Verdoolaege, S.: Integer Set Library: Manual, available from ⟨http://isl.gforge.inria.fr/manual.pdf⟩ (accessed 2018-03).

[13] Verdoolaege, S.: isl: An Integer Set Library for the Polyhedral Model, *International Congress on Mathematical Software* (*ICMS*) (2010).

[14] Kato, Y. and Seto, K.: Loop Fusion with Outer Loop Shifting for High-level Synthesis, *IPSJ Trans. System LSI Design Methodology*, Vol.6 (2013).

Kenshu Seto received his B.S. degree in electrical engineering, M.S. and D. Eng. degrees 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*)