IPSJ Transactions on System LSI Design Methodology Vol.13 2–9 (Feb. 2020)

[DOI: 10.2197/ipsjtsldm.13.2]

Regular Paper

Shift Register Initialization in Scalar Replacement for Reducing Code Size

Kenshu Seto[1,a)]

Received: June 9, 2019, Revised: September 5, 2019,
Accepted: October 27, 2019

Abstract: Scalar replacement is an effective technique to improve the performance of the RTL code generated by high-level synthesis (HLS) from C programs with intensive array accesses. In scalar replacement, data accessed from arrays are stored into shift registers, and later array accesses on the same data are replaced with the accesses to the shift registers instead of the arrays. Namely, scalar replacement replaces array accesses with shift register accesses. Since arrays in C programs are usually mapped to RAMs with limited numbers of ports, reducing array accesses with scalar replacement leads to the memory access reduction, which in turn improves the performance of the resulting RTL code. In real-life C programs, sometimes, shift registers must be initialized conditionally using multiple array accesses, which increases the number of array accesses in main loops. To reduce the conditional array access in the main loops, the previous scalar replacement method proposed the use of a loop transformation called loop peeling. Loop peeling brings significant increase in code size, leading to the negative impacts on performance or circuit area of the synthesized hardware. In this paper, we propose a new method to initialize shift registers without loop peeling. The proposed method works as a preprocessing of the input C program prior to scalar replacement. With experimental results, we demonstrate the proposed method reduces the numbers of execution cycles of the synthesized hardware compared to the previous method.

Keywords: high-level synthesis, memory access optimization, scalar replacement

1. Introduction

High-level synthesis (HLS) [1], [2] generates RTL code automatically from C programs, so HLS leads to significant reduction of the design time of digital circuits. Unfortunately, the application of HLS to unoptimized C programs sometimes generates RTL code whose quality in terms of area or performance is insufficient, so users of HLS usually have to optimize the input C programs manually to enhance the quality of the generated RTL. The manual code optimization is time consuming, since the users have to study the complex optimizations sufficiently and manually apply the optimizations to the C programs in a careful manner. Without sufficient understanding of the optimizations, the users lose the chance of optimizations and get the low quality RTL code. Thus, the automatic code optimization will greatly help improve the usefulness of HLS.

Memory access optimizations are often necessary to generate high-quality RTL code with HLS. In HLS, arrays in C programs are often implemented as RAMs. RAMs typically have at most 2 ports. So, memory accesses are usually performance bottlenecks in C programs with not a few memory accesses, such as the programs for image filtering or stencil computations. In order to address the bottlenecks, optimizations that address the limited memory bandwidth are necessary. Such optimizations include array partitioning (or memory partitioning) [3], [4], [5], and scalar replacement [6], [7], [8]. Array partitioning increases the memory bandwidth of RAMs by partitioning the RAMs into smaller RAMs. Scalar replacement resolves the limited memory access bottlenecks by storing the accessed data from a RAM into shift registers and by accessing the data in the shift registers instead of the RAM. Array partitioning is an effective technique to enhance the memory bandwidth of on-chip RAMs but it is sometimes not applicable to off-chip RAMs that cannot be partitioned. This work focus on the improvement of scalar replacement.

Scalar replacement usually must be applied to nested loops, since algorithms in image processing or stencil computations are usually represented by nested loops. The first scalar replacement technique for nested loops was proposed in Ref. [6]. In Ref. [7], scalar replacement based on the polyhedral model was proposed. The method proposed in Ref. [7] can handle array accesses with constant subscripts. Array accesses with constant subscripts are common in image filtering applications. In Ref. [8], the work [7] was extended to reduce the circuit area of the shift registers by replacing shift registers with circular buffers. In scalar replacement techniques [6], [7], [8], an array access that accesses data at the earliest iterations is called *generator*. For the example code in **Fig. 1**, the access B[i][j] is the generator. B[i-1][j-1] accesses the same data as the generator B[i][j] does in the later iterations. Such an access as B[i-1][j-1] which reuses data accessed by a generator is called a *reuse destination*, and reuse destinations are usually removed from the main loop body by scalar replacement. Unfortunately, B[i-1][j-1] also accesses data that are not accessed by

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

```
1 for (i=1; i <5; i++) {
2    for (j=1; j <5; j++) {
3       A[i][j] = B[i][j]+B[i-1][j-1];
4    }
5 }
```

Fig. 1 Example code used throughout this paper.

the generator B[i][j]. In this case, the access B[i-1][j-1] cannot be completely removed from the main loop body after the scalar replacement, since the initialization of shift registers with the array access B[i-1][j-1] is necessary in the main loop body. To reduce the array accesses in the main loop body as much as possible after scalar replacement, the previous algorithm [6] proposed the use of *loop peeling*, which is a loop transformation that removes parts of the iterations of a nested loop outside of the main loop. Unfortunately, loop peeling significantly increases the code size, and the increased code size likely increases the circuit area or degrades the circuit performance after HLS. In this work, we present a scalar replacement technique that does not use loop peeling even when reuse destinations access data that are not accessed by the generator.

In Section 2, we briefly review the preliminaries for scalar replacement and the polyhedral model. In Section 3, we discuss the problem of the previous scalar replacement with a simplified example in Fig. 1. In Section 4, we propose a technique to solve the problem explained in Section 3. Section 5 demonstrate the impact of the proposed technique, followed by the conclusions in Section 6.

2. Preliminaries

In this section, we briefly review the definitions on scalar replacement and the polyhedral model [10]. These definitions are used in this paper. For more details on the definitions, please refer to Refs. [7], [8].

Definition 2.1 (Reuse source and destination) When an array element accessed by an array access s is accessed in later loop iterations by an array access d, we say that the array access d **reuses** the data accessed by the array access s. 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 starts to access each array element and the data accessed by the generator will be reused later by its reuse destinations.

Definition 2.3 (Reuse vector) A reuse vector $\langle d_1, d_2, \ldots \rangle$ represents the numbers of 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, ... corresponds to iterations for outermost loop, 2nd outermost loop, ..., respectively.

Definition 2.4 (Reuse distance) A **reuse distance** is calculated from a reuse vector, $\langle d_1, d_2, \ldots \rangle$ by the following formula (1) where I_k represents the number of loop iterations for the k-th loop from the outermost loop, and n represents the depth of the nested loop. A reuse distance means the number of *innermost* loop iterations after which the reuses from a reuse source s to a reuse destination d occur.

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

Definition 2.5 (Iteration vector) A vector $\vec{i}$ whose elements are values of loop induction variables from the outermost loop to the innermost loop in a nested loop is called an **iteration vector**.

Definition 2.6 (Domain of array access) A set of iteration vectors, D_a, that contains all iterations in which an array access a in a nested loop is executed is called the **domain** of the array access a.

Definition 2.7 (Domain of loop) A set of iteration vectors, D_L, that contains all iterations in which a nested loop is executed is called the **domain** of the nested loop L. For Fig. 1, the domain of the nested loop is $\{(i, j) \mid 1 \leq i \leq 4 \wedge 1 \leq j \leq 4\}$

Definition 2.8 (Subscript vector of array accesses) We define a vector $\vec{s} = (s_1, s_2, \ldots)$ whose elements are subscript values of an array as a **subscript vector**. The number of dimensions of a subscript vector for an m-dimensional array equals to m, and the i-th element of a subscript vector corresponds to the i-th subscript from the leftmost array subscript. A subscript vector represents subscript values of an array access a at a specific iteration vector $\vec{i} \in D_a$. For example, the subscript vector of the array access B[i-1][j-1] at $\vec{i} = (i, j) = (1, 1)$ is $\vec{s} = (s_1, s_2) = (0, 0)$.

Definition 2.9 (Subscript space of array access) We call the set of all subscript vectors of an array access a a **subscript space** of the array access a and denote it by S_a. For example, the subscript space of the array access B[i-1][j-1] in Fig. 1 is $S_{\text{B[i-1][j-1]}} = \{(s_1, s_2) \mid 0 \leq i \leq 3 \wedge 0 \leq j \leq 3\}$.

Definition 2.10 (Access function of array access) The **access function** F_a of an array access a is a function $F_a : \vec{i} \in D_a \rightarrow \vec{s} \in S_a$. In other words, the access function of an array access a represents a mapping from each iteration vector $\vec{i}$ in D_a to a subscript vector $\vec{s}$ of the array access.

Definition 2.11 (Reuse relation between array accesses) The **reuse relation** $R_{s,d}$ from a reuse source s to a reuse destination d is a set of pairs $(\vec{i_s}, \vec{i_d})$ of iteration vectors $\vec{i_s} \in D_s$ and $\vec{i_d} \in D_d$ where the array access d at the iteration vector $\vec{i_d} \in D_d$ accesses the same array element which was previously accessed by the array access s at the iteration vector $\vec{i_s} \in D_s$.

3. The Problem of the Previous Shift Register Initialization

In this section, we illustrate the problem of the previous scalar replacement techniques [6], [7], [8], in particular, in the shift register initialization. In order to describe the problem, we briefly review scalar replacement with the simple code in Fig. 1. For the code in Fig. 1, scalar replacement tries to reduce the number of the array accesses to the array B from two to one. In Fig. 1, the generator is B[i][j] and the reuse destination is B[i-1][j-1]. The reuse destination B[i-1][j-1] accesses the data accessed by the generator B[i][j] one iteration later in loop i and one iteration later in loop j. In other words, the reuse vector from the generator B[i][j] to the reuse destination B[i-1][j-1] is $\langle 1, 1 \rangle$. By assigning the reuse vector $\langle 1, 1 \rangle$ to the formula (1) in Definition 2.4, the

Table 1 The reuse information table for Fig. 1 used in Ref. [7].

	Access type	Array access	Reuse vector	Reuse distance	Scalar variable
1	Generator	B[i][i]	N/A	N/A	B_reg_0
2	Reuse	B[i-1][j-1]	⟨1, 1⟩	5	B_reg_5

```
 1 for (i=1; i <5; i++) {
 2    for (j=1; j <5; j++) {
 3       B_reg_0 = B[i][j];
 4       if (i==1 | | j == 1)
 5          B_reg_5 = B[i-1][j-1];
 6       A[i][j] = B_reg_0 +B_reg_5;
 7       B_reg_5 = B_reg_4;
 8       B_reg_4 = B_reg_3;
          ...
11       B_reg_1 = B_reg_0;
12    }
13 }
```

Fig. 2 Code after the previous scalar replacement [6] (before loop peeling).

reuse distance from the generator B[i][j] to the reuse destination B[i-1][j-1] is calculated to be 5. The table that summarizes necessary information for scalar replacement (called *reuse information table*) is shown in **Table 1**. **Figure 2** shows the code after applying the previous scalar replacement and shift register initialization [6] to the code in Fig. 1. Since the reuse distance is 5, we prepare shift registers of length 5 consisting of B_reg_1, ..., B_reg_5. The input of the shift registers, B_reg_0, is initialized to the accessed data by the generator B[i][j] as shown in the line 3 of Fig. 2.

As shown in the lines 4 and 5 of Fig. 2, we also need to initialize the B_reg_5 in the shift registers to the data accessed by the reuse destination B[i-1][j-1]. This is because the reuse destination B[i-1][j-1] accesses the data that are not accessed by the generator B[i][j] when i is 1 or j is 1. For example, the reuse destination B[i-1][j-1] accesses B[0][0] when i is 1 and j is 1, however, the generator B[i][j] does not access B[0][0] for any iteration vector in the domain $D_{B[i][j]}$ of the generator where $D_{B[i][j]} = \{(i, j) \mid 1 \leq i \leq 4 \wedge 1 \leq j \leq 4\}$ and $S_{B[i][j]} = \{(i, j) \mid 1 \leq i \leq 4 \wedge 1 \leq j \leq 4\}$. In order to appropriately initialize the shift register B_reg_5 when i is 1 or j is 1, the lines 4 and 5 of Fig. 2 are added in the main loop body. Unfortunately, the addition of the line 5 in Fig. 2 increases the number of the array accesses to the array B by 1 in Fig. 2. Assuming that the array B is mapped to a 1-port RAM, the initiation interval (II) of the innermost loop in Fig. 2 after loop pipelining is not reduced, so that we cannot expect the reduction of the number of execution cycles although scalar replacement is performed.

In order to remove the array accesses incurred by the additional initialization of the shift registers from the main loop body, the previous work [6] proposed the application of a loop transformation called *loop peeling* to the loop body. The code shown in **Fig. 3** is the result of applying loop peeling to the code shown in Fig. 2. By peeling out both the outer loop iteration for $i = 1$ and the inner loop iteration for $j = 1$ from the loop body in Fig. 2, we can eliminate the conditional array access in the lines 4 and 5 in Fig. 2 and the resulting main loop body in lines 20 to 27 of Fig. 3 has only one access to the array B in line 21. Assuming that the array B is mapped to a 1-port RAM as before, the initiation interval (II) of the innermost loop from the lines 20 to 28 in Fig. 3 after loop pipelining is reduced to one from two, so that we expect the reduction of the number of execution cycles after scalar replacement. Unfortunately, after the loop peeling, the code size of Fig. 3 is increased by more than 2 times to 29 lines from 13 lines of Fig. 2. Significant increases in the code size of input C programs usually lead to the degradation in the performance or the circuit area of synthesized hardware by HLS.

```
 1 for (j=1; j <5; j++){
 2    B_reg_0 = B[1][j];
 3    B_reg_5 = B[0][j-1];
 4    A[1][j] = B_reg_0 + B_reg_5;
 5    B_reg_5 = B_reg_4;
       ...
 9    B_reg_1 = B_reg_0;
10 }
11 for (i=2; i <5; i++){
12    B_reg_0 = B[i][1];
13    B_reg_5 = B[i-1][0];
14    A[i][1] = B_reg_0 + B_reg_5;
15    B_reg_5 = B_reg_4;
       ...
19    B_reg_1 = B_reg_0;
20    for (j=2; j <5; j++){
21       B_reg_0 = B[i][j];
22       A[i][j] = B_reg_0 + B_reg_5;
23       B_reg_5 = B_reg_4;
          ...
27       B_reg_1 = B_reg_0;
28    }
29 }
```

Fig. 3 Code after the previous scalar replacement [6] (after loop peeling).

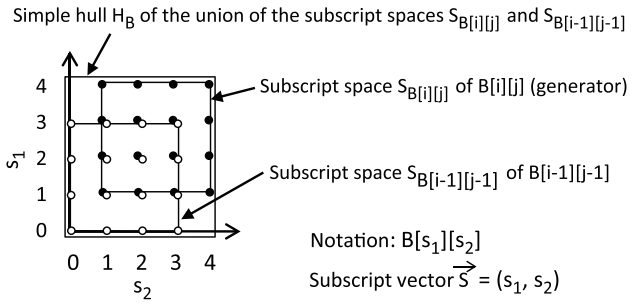

Fig. 4 Subscript spaces for B[i][j] and B[i-1][j-1] and the "simple hull" of the union of the subscript spaces for B[i][j], B[i-1][j-1].

4. The Proposed Shift Register Initialization

In this section, we propose a new method to reduce the code size of shift register initialization for scalar replacement. As in Refs.[7], [8], the input C programs for the proposed method are assumed to be stencil computations [5] in Static Control Part (SCoP) format [9]. We also assume that each target C program consists of a fully nested loop where all statements are contained in the innermost loop and no statement exists outside the innermost loop. We also assume that each target array is accessed only inside the loop body, that each target array has at most one write access and at least one read access in the loop body, and that the increment value of each loop induction variable is 1. In addition, we assume that each target array has only one generator and that loop bounds are represented not by parameters unknown at compile-time but by constant values.

Here, we explain the main idea of the proposal with **Fig. 4**. In Fig. 4, the set of black circles and the set of white circles represent the subscript spaces of the array accesses B[i][j] and B[i-1][j-1], respectively. The subscript spaces depict which data in the ar-

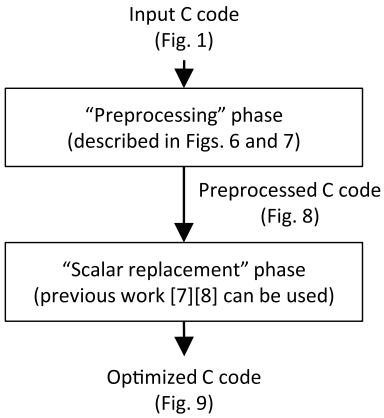

Fig. 5 Overall flow of the proposed method.

ray B are accessed by the generator B[i][j] and B[i-1][j-1]. As seen from Fig. 4, the generator B[i][j] does not access the data with $s_1 = 0$ or $s_2 = 0$ which are accessed by the reuse destination B[i-1][j-1]. To solve the problem, we propose to add a new generator B[i][j] at the top of the loop body as a dummy statement and to make the new generator access all the data accessed by all the reuse destinations by extending the domain of the loop from $D_L = \{(i, j) \mid 1 \leq i \leq 4 \wedge 1 \leq j \leq 4\}$ to $D'_L = \{(i, j) \mid 0 \leq i \leq 4 \wedge 0 \leq j \leq 4\}$ as shown in Fig. 8. To prevent the change in the program semantics, the original loop body is enclosed with a conditional statement, as shown in line 4 of Fig. 8, that guarantees that the original loop body is executed in the original domain of the loop D_L. Although the new generator temporarily increases the number of array accesses, scalar replacement which will be applied later will eliminate array accesses as shown in Fig. 9. Although the loop bounds are slightly increased, the proposed method does not incur additional array accesses for shift register initialization which have to be removed with loop peeling.

Figure 5 shows the overall flow of the proposed method. The overall flow consists of two phases, the "Preprocessing" phase and the "Scalar replacement" phase. The "Preprocessing" phase accepts input C code as shown in Fig. 1 and generates preprocessed C code as shown in Fig. 8. The "Scalar replacement" phase accepts the preprocessed C code and generates the optimized code after scalar replacement as shown in Fig. 9. In Fig. 5, we use the existing scalar replacement method [8] for the "Scalar replacement" phase. In the following, we will explain the "Preprocessing" phase.

The "Preprocessing" phase consists of the following steps and for Steps 1 to 4, we use the same techniques as explained in the previous work [7], [8], so please refer to Refs. [7], [8] for details.

Step 1. Build the polyhedral model for array accesses.
Step 2. Perform reuse analysis with the polyhedral model.
Step 3. Build a reuse graph for each array.
Step 4. Find the unique generator for each array.
Step 5. Compute extended domains of the loop.
Step 6. Generate the preprocessed C code.

In order to perform the C code generation in Step 6, we need to compute the extended domains of the loop in Step 5 with the polyhedral model, reuse analysis results and the generator information obtained from Steps 1 to 4. The detailed algorithms for Steps 5 and 6 are illustrated in **Figs. 6** and **7**, respectively.

Algorithm *ComputeExtendedDomainsOfLoop*

Input D_a: Domains for each array access a
Input F_a: Access functions for each array access a
Input $R_{g,a}$: Reuse relations from each generator g to each array access a
Input g_A: Generators for each array A
Input D_L: Domain of the original loop in input C code
Output D'_A: Extended domains of loop for new generator of each array A
Output D'_L: Extended domain of loop for all new generators

1: $D'_L \leftarrow D_L$
2: **foreach** array A :
3: $extendFlag \leftarrow$ false
4: **foreach** array access a to array A
5: **if** ($D_a \neq$ Range($R_{g,a}$)) :
6: $extendFlag \leftarrow$ true
7: **break**
8: **if** $extendFlag ==$ false :
9: $D'_A \leftarrow NULL$
10: **else**:
11: $U_A = \emptyset$
12: **foreach** array access a to array A :
13: $S_a \leftarrow F_a(D_a)$
14: $U_A \leftarrow U_A \cup S_a$
15: $H_A \leftarrow$ simple_hull(U_A)
16: $D'_A \leftarrow {F_{g_A}}^{-1}(H_A)$
17: **if** $D'_A \neq NULL$:
18: $D'_L \leftarrow D'_L \cup D'_A$
19: $D'_L \leftarrow$ simple_hull(D'_L)

Fig. 6 Procedure for computing extended domains of a loop for all new generators and for each new generator.

Algorithm *CodeGeneration*

Input Input C code
Input Text string "$genstr_A$" for each generator of each array A
Input D_L: Domain of the original loop in input C code
Input D'_A: Extended domains of loop for new generator of array A
Input D'_L: Extended domain of loop for all new generators
Output C code after preprocessing

1: **for** $d = 1$ to $d = n\_dim$:
2: Change the lower bound of d-th loop to min(project(D'_L, d))
3: Change the upper bound of d-th loop to max(project(D'_L, d))
4: **foreach** array A :
5: **if** $D'_A == NULL$:
6: **continue**
7: $cond_A \leftarrow$ gist(D'_A, D'_L)
8: Add the following code in the top of the loop body :
"if ($cond_A$) A_reg_0 = $genstr_A$;"
9: $cond_{body} \leftarrow$ gist(D_L, D'_L)
10: Add the following *if* statement enclosing the original loop body :
"if ($cond_{body}$) { *original loop body* }"

Fig. 7 Procedure for generating preprocessed C code.

The inputs to the algorithm in Fig. 6 is: (1) D_a: domains for each array access a (defined in Def. 2.6), (2) F_a: access functions for each array access a (defined in Def. 2.10), (3) $R_{g,a}$: reuse relations from the generator g to the array accesses a for each array (defined in Def. 2.11) and (4) g_A: generators for each array A. The outputs of the algorithm in in Fig. 6 are D'_A and D'_L that are used in the algorithm in Fig. 7. D'_A represents the extended domain of the loop for the newly added generator of the array A to access all the data accessed by all the reuse destinations of the array A. D'_A is used to generate the conditions of the *if* statements for genera-

```
1 for (i=0; i <5; i++) {
2    for (j=0; j <5; j++) {
3       B_reg_0 = B[i][j];
4       if (i >= 1 && j >= 1) {
5          A[i][j] = B[i][j]+B[i-1][j-1];
6       }
7    }
8 }
```

Fig. 8 Code after the proposed preprocessing for shift register initialization.

tors when multiple generators are added in the loop body. Since only one generator B[i][j] is newly added in the line 3 of **Fig. 8**, such a *if* statement is not added in the case of Fig. 8. D'_L represents the extended domain of the loop considering all the newly added generators and $D'_A \subseteq D'_L$ holds. D'_L is used to modify the loop bounds as shown in the lines 1 and 2 of Fig. 8. Please note that the domain of a loop is defined in Def. 2.7.

In the line 1 of Fig. 6, D'_L is initialized to the domain of the original loop D_L. The line 2 iterates over all the target arrays A in the input C code. In the line 3, *extendedFlag* is initialized to False. *extendedFlag* means weather it is necessary to extend the domain of the loop for the current array A. The lines 4 to 7 are checking if there exists an array access a to the array A that accesses data that are not accessed by the generator g_A. The main part of the checking is the line 5 where "Range" is the function that returns the destination of the reuse relation $R_{g,a}$. $D_a \neq \mathrm{Range}(R_{g,a})$ means that there exists an iteration in D_a where the array access a cannot reuse the data accessed by the generator g_A. In general, $\mathrm{Range}(R_{g,a}) \subseteq D_a$ holds. When every reuse destination of an array A reuses only the data accessed by the generator g_A, we do not need the preprocessing for the array A, so we set D'_A to $NULL$ in the lines 8 and 9. Otherwise, the lines 11 to 16 are executed to compute D'_A. In the line 11, U_A is initialized to the empty set. U_A represents the union set (space) of the subscript spaces S_a of all the array accesses a to an array A. The subscript space of an array access is defined in Def. 2.9. In the line 13, the subscript space S_a of an array access a is computed by applying the array access function F_a of a to the domain D_a of a. From the lines 12 to 14, the union set U_A is computed from all the S_a. For example, the union set U_B of the subscript spaces $S_{\mathrm{B[i][j]}}$ and $S_{\mathrm{B[i-1][j-1]}}$ of B[i][j] and B[i-1][j-1] in Fig. 4 is the set consisting of a black circle *or* a white circle. The line 15 in Fig. 6 computes the "simple hull" H_A of the union set U_A. "simple hull" operation [11] makes the generated code simple enough to apply scalar replacement.

For example, the "simple hull" H_B of the union set U_B of the subscript spaces of B[i][j] and B[i-1][j-1] in Fig. 4 is the square-shaped set including both of the subscript spaces $S_{\mathrm{B[i][j]}}$ and $S_{\mathrm{B[i-1][j-1]}}$. In Fig. 4, H_B includes the subscript vectors of $(s_1, s_2) = (4, 0), (0, 4)$ while U_B does not. In the line 16, ${F_{g_A}}^{-1}$ represents the inverse function of F_{g_A} where F_{g_A} is the access function of the generator g_A. By applying ${F_{g_A}}^{-1}$ to H_A, we obtain D'_A. In the line 18, D'_L is updated by unioning D'_A when D'_A is not $NULL$. Finally in the line 19, we compute the "simple hull" of the unioned set D'_L over all the target arrays in the input C code.

Figure 7 shows the algorithm that generates preprocessed C code. The parts of the inputs to the algorithm in Fig. 7, namely D'_A and D'_L, are the results of Fig. 6. Other than D'_A and D'_L, the inputs to the algorithm are (1) Input C code which will be modified by this algorithm, (2) Text strings $genstr_A$ for each generator and (3) D_L: The domain of the original loop in the input C code. The line 1 in Fig. 7 iterates over each loop dimension of the fully nested loop in the input C code from the outermost loop. $d = 1$ and $d = n\_dim$ represent the outermost loop (1-st loop) and the innermost loop (n_dim-th loop), respectively where n_dim denotes the depth of the nested loop. The lines 2 and 3 update the lower bound and upper bound of d-th loop, respectively. project(D'_L, d) is a function that performs the projection of the (multi-dimensional) set D'_L onto a given dimension d. The result of project(D'_L, d) is a one-dimensional set. The min and max functions in the lines 2 and 3 return the minimum and maximum values of the one-dimensional (integer) set. The lines 7 to 8 add new generators, such as the one in the line 3 of Fig. 8. The line 4 of Fig. 7 iterates over the target arrays and the arrays that do not require addition of the new generators are skipped as shown in the lines 5 to 6. When multiple new generators for different arrays $A_1, A_2, \ldots$ must be added, the extended domains of the loop $D'_{A_1}, D'_{A_2}, \ldots$ for these new generators are usually different, so *if* statements must be usually added to the statements of the new generator to reflect the different extended domains as shown in the line 8 of Fig. 7. The conditions $cond_A$ of the *if* statements are computed in the line 7. Instead of using D'_A directly, we perform "gist" operation [11] on D'_A with respect to D'_L in order to simplify $cond_A$. In case of the code in Fig. 8 where only one generator B[i][j] are newly added, D'_A and D'_L are equal, so gist(D'_A, D'_L) evaluates to True. As a result, the *if* statement is not necessary as shown in the line 3 of Fig. 8. $genstr_A$ in the line 8 of Fig. 7 is the text string for the generator of the array A. For example, in the case of Fig. 1, the text string for the generator of the array B is B[i][j]. The line 10 in Fig. 7 adds the *if* statement that encloses the loop body, as shown in the lines 4 and 6 in Fig. 8. After the proposed preprocessing, the loop bounds are extended as shown in the lines 1–2 in Fig. 8. To make the preprocessed code in Fig. 8 equivalent to the original input C code in Fig. 1, we have to enclose the original loop body (the line 5 in Fig. 8) with the *if* statement in the lines 4 and 6 in Fig. 8. The *if* statement constrains the original loop body to execute only in the original domain of the loop D_L. The line 9 of Fig. 7 generates the condition $cond_{body}$ of the *if* statement. Again, "gist" operation is applied to D_L with respect to D'_L to simplify the condition $cond_{body}$.

After applying the "Preprocessing" phase to the input C code, such as the one shown in Fig. 1, we obtain the preprocessed C code, such as the one shown in Fig. 8. The existing scalar replacement, such as Refs. [7], [8], can be applied to the preprocessed code to perform scalar replacement. The reuse information table [7], [8] for the code in Fig. 8 is shown in **Table 2**, and the resulting code after scalar replacement using the table is shown in **Fig. 9**.

Table 2 The reuse information table for the code in Fig. 8.

	Access type	Array access	Reuse vector	Reuse distance	Scalar variable
1	Generator	B[i][j]	N/A	N/A	B_reg_0
2	Reuse	B[i][j]	⟨0, 0⟩	0	B_reg_0
3	Reuse	B[i-1][j-1]	⟨1, 1⟩	6	B_reg_6

Table 3 Comparison between the previous method [6] and the proposed method for shift register initialization.

Benchmark programs	Code type	# lines	II	# of execution cycles [cycles]	Gate counts	Gate counts of I/O RAMs	Gate counts including I/O RAMs
Example	*previous* [6] *with loop peeling*	30	1	1,116 (1.00)	5,299 (1.00)	98,304	103,603 (1.00)
	previous [6] *without loop peeling*	15	2	1,986 (1.78)	5,580 (1.05)	98,304	103,884 (1.00)
	proposed	14	1	1,090 (0.98)	5,182 (0.98)	98,304	103,486 (1.00)
jacobi-2d	*previous* [6] *with loop peeling*	264	1	1,989 (1.00)	25,459 (1.00)	98,304	123,763 (1.00)
	previous [6] *without loop peeling*	71	7	6,542 (3.29)	5,995 (0.24)	98,304	104,299 (0.84)
	proposed	73	1	1,346 (0.68)	19,421 (0.76)	98,304	117,725 (0.95)
seidel-2d	*previous* [6] *with loop peeling*	287	2	2,361 (1.00)	24,150 (1.00)	49,152	73,302 (1.00)
	previous [6] *without loop peeling*	79	10	9,002 (3.81)	17,958 (0.74)	49,152	67,110 (0.92)
	proposed	76	2	2,050 (0.87)	17,602 (0.73)	49,152	66,754 (0.91)
heat-3d	*previous* [6] *with loop peeling*	262	1	10,626 (1.00)	63,072 (1.00)	196,608	259,680 (1.00)
	previous [6] *without loop peeling*	50	7	20,974 (1.97)	51,648 (0.82)	196,608	248,256 (0.96)
	proposed	50	1	6,403 (0.60)	64,836 (1.03)	196,608	261,444 (1.01)

```
 1 for (i=0; i <5; i++) {
 2   for (j=0; j <5; j++) {
 3     B_reg_0 = B[i][j];
 4     if (i >= 1 && j >= 1) {
 5       A[i][j] = B_reg_0+B_reg_6;
 6     }
 7     B_reg_6 = B_reg_5;
 8     ....
 9     B_reg_1 = B_reg_0;
10   }
11 }
```

Fig. 9 Code after the proposed preprocessing and scalar replacement.

5. Experimental Results

In this section, we show the impact of the proposed shift register initialization method presented in Section 4 on the numbers of execution cycles and the circuit area of the generated RTL code, compared to the previous shift register initialization method [6].

5.1 Experimental Setup

We implemented the preprocessing tool for the shift register initialization algorithm proposed in Section 4 based on Ref. [8] with ISL (Integer Set Library) [11]. 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. In HLS, we used the loop pipelining directives to all the innermost loops with the smallest initiation intervals (IIs). 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 were mapped to 1-port RAMs, and the arrays corresponding to circular buffers [8] were mapped to 2-port RAMs.

We used the benchmark programs shown in **Table 3** for the experiment. *Example* is the code shown in Fig. 1 except that the upper bounds of the *for* loops were increased from 5 to 30. *jacobi-2d* and *seidel-2d* are the code for a Jacobi like stencil computation with a 5-point stencil pattern and the code for a Gauss-Seidel like stencil computation with a 9-point stencil pattern, respectively. All of the programs are 2-dimensional loops except *heat-3d*. *jacobi-2d* and *seidel-2d* work on 2-dimensional arrays of 32×32. *heat-3d* is a 3-dimensional stencil computation for solving the heat equation and works on 3-dimensional arrays of $16 \times 16 \times 16$. We applied the proposed preprocessing tool to the benchmark programs, followed by the scalar replacement tool proposed in Ref. [8].

5.2 Results and Discussions

Table 3 shows the experimental results. In the table, *previous* [6] *with loop peeling*, *previous* [6] *without loop peeling* and *proposed* show the results of the previous shift register initialization that uses loop peeling [6], those of the previous shift register initialization that does not use loop peeling [6] and those of the proposed shift register initialization, respectively. All the generated RTL code satisfied the clock constraints of 500 MHz.

In Table 3, the column "# lines" shows the numbers of lines after scalar replacement, such as those in Figs. 2, 3 and 9, for the main computation part of the benchmark programs, and we see that the proposed method could reduce code size significantly compared to the code generated by the previous method with loop peeling. The code size by the previous method without loop peeling was almost the same as that by the proposed method. The previous shift register initialization [6] required the loop peeling of the iterations i=1 and j=1 for *Example*, the iterations i=1, j=1 and j=30 for *jacobi-2d*, the iterations i=1, j=1, j=29 and j=30 for *seidel-2d* and the iterations i=1, j=1, j=14, k=1 and k=14 for *heat-3d*, which resulted in the significant increase in code size. Instead of the increased code size due to the loop peeling, the proposed method increased the loop counts of the loops due to the extended domains of the loops. *Example* increased the loop count by 1 for each loop dimension and both *jacobi-2d*, *seidel-2d* and *heat-3d* increased the loop counts by 2 for each loop dimension.

In Table 3, "II" shows the results for the initiation intervals (IIs) after loop pipelining for the main loops (In case of Fig. 3, the main loop means the loop from the lines 20 to 28). The IIs for the previous method with loop peeling and those for the proposed method were the same. The IIs for the previous method without loop peeling, however, were significantly larger than those for the proposed method, since the array accesses in the main loop bodies were not reduced by the previous method without loop peeling. So, the numbers of execution cycles for the previous method without loop peeling were much larger than those by the proposed method.

Comparing the results in the column "# of execution cycles" in Table 3 for the previous method with loop peeling and the proposed method, we observe that the reduction in the number of

execution cycles was small for the simple benchmark program *Example*, however, the significant reductions of 32%, 13% and 40% were achieved for *jacobi-2d*, *seidel-2d* and *heat-3d*, respectively. In the previous method [6] with loop peeling, loop iterations are peeled out of the main loops, so that these peeled loops cannot be executed in parallel with the main loops. As a result, the numbers of the execution cycles by the previous method with loop peeling were increased compared to those by the proposed method. Although the proposed method lead to the slightly increased loop counts of the nested loops, the negative impacts of the increased loop counts was not large compared to the negative impacts by the loop peeling used in the previous method [6].

In Table 3, I/O RAMs mean RAMs that are mapped to input or output arrays that contain input or output data, and "gate counts" mean the gate counts in terms of NAND2 gates. We approximated the gate counts for 1-port RAMs by 1.5 NAND gates per 1 bit, since a 1-bit memory cell of 1-port RAMs consists of 6 transistors and a NAND gate consists of 4 transistors. In this approximated gate counts for 1-port RAMs, we only considered the gate counts for memory cells for the sake of simplicity, omitting the gate counts for decoders and sense amplifiers. According to Ref. [12], the density of the 1-port SRAM was twice the density of the 2-port SRAM. So, we approximated the gate counts for 2-port RAMs used in circular buffers by 3 NAND gates per 1 bit which is twice the 1.5 NAND gates per 1 bit. In Table 3, "Gate counts", the sixth column from the left, represent the gate counts of the synthesized hardware excluding the gate counts due to the I/O RAMs. In the table, "Gate counts of I/O RAMs" represents the gate counts of the I/O RAMs, and "Gate counts including I/O RAMs" is the sum of the "Gate counts" and "Gate counts of I/O RAMs". The column of "Gate counts" shows that the proposed shift register initialization method could reduce the gate counts by 2%, 24% and 27% for *Example*, *jacobi-2d* and *seidel-2d*, respectively, compared to the previous shift register initialization method with loop peeling. For *heat-3d*, the proposed shift register initialization method increased the gate counts by 3% compared to the previous shift register initialization method with loop peeling. This is due to the increased circular buffer size caused by the increased loop counts of the 3-dimensional loops. Even when the gate counts of the I/O RAMs were taken into account, as shown in the column "Gate counts including I/O RAMs", the proposed method could reduce the gate counts by 5% and 9% for *jacobi-2d* and *seidel-2d*, respectively, compared to the previous method with loop peeling. Although the proposed shift register initialization method resulted in the increased loop counts and hence the increased numbers of shift registers, the negative impacts of the increased shift registers were not significant compared to those of the increased code size by the loop peeling. The increased code size by the loop peeling resulted in significant increases in multiplexers, which brought significantly negative impacts on the numbers of the execution cycles and the circuit area of the synthesized circuits.

In summary, we found that the proposed shift register initialization method significantly reduced the code size after scalar replacement compared to the previous shift register initialization method [6] with loop peeling. The previous method without loop peeling generated the code with almost the same code size with the proposed method, however, the numbers of execution cycles were significantly worse than those by the proposed method. So the previous shift register initialization method requires loop peeling in order to design high-performance accelerators. Due to the code size reduction, the proposed method could significantly reduce the numbers of execution cycles compared to the previous method with loop peeling. The gate counts of the synthesized design for *jacobi-2d* and *seidel-2d* were reduced with the proposed method compared to the previous method with loop peeling. The gate count of the synthesized design for *heat-3d* was slightly increased with the proposed method compared to the previous method with loop peeling. Therefore, we claim that the proposed method is promising as a shift register initialization for scalar replacement when we perform scalar replacement to C programs where generators do not access all the data that are accessed by their reuse destinations.

6. Conclusions

Scalar replacement is a compiler optimization that is used to reduce array accesses in the input C programs for high-level synthesis, and shift registers are introduced after the application of scalar replacement. When reuse destinations access data that are not accessed by the generator, it is necessary to initialize the shift registers appropriately with extra array accesses. The previous method to initialize the shift registers [6] performs loop peeling to remove the extra array accesses from the main loop and to reduce the initiation intervals (IIs) of the pipelined loops. We discussed that the loop peeling increases the code size of the input C programs and negatively impacts the numbers of execution cycles and the gate counts of the synthesized hardware. In this paper, we proposed a new method for initializing the shift registers introduced by scalar replacement. The proposed method works as a preprocessing of the input C program prior to scalar replacement. The proposed method adds additional generators in the input C programs and these additional generators accesses all data that are accessed by their reuse destinations. Experimental results demonstrated that the proposed method for the shift register initialization successfully reduced the size of C programs that are synthesized by HLS without increasing the IIs. Compared to the previous method with loop peeling, the proposed method reduced the numbers of the execution cycles significantly and, for some cases, the proposed method reduced the gate counts of the synthesized hardware. We claim that the proposed shift register initialization method is promising when we perform scalar replacement to the input C programs where generators do not access all the data that are accessed by the reuse destinations.

References

[1] Gajski, D.D.: *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] So, B. and Hall, M.W.: Increasing the Applicability of Scalar Replacement, *Compiler Construction* (*CC*) (2004).
[7] Seto, K.: Scalar Replacement with Polyhedral Model, *IPSJ Trans. System LSI Design Methodology*, Vol.11 (2018).
[8] Seto, K.: Scalar Replacement with Circular Buffers, *IPSJ Trans. System LSI Design Methodology*, Vol.12 (2019).
[9] Bastou, C.: 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 2019-06).
[10] Bastoul, C., Cohen, A., Girbal, S., Sharma, S. and Temam, O.: Putting Polyhedral Loop Transformations to Work, *International Workshop on Languages and Compilers for Parallel Computers* (*LCPC*) (2003).
[11] Verdoolaege, S.: Integer Set Library: Manual (2019).
[12] Nii, K., Yabuuchi, M., Tsukamoto, Y., Ohbayashi, S., Oda, Y., Usui, K., Kawamura, T., Tsuboi, N., Iwasaki, T., Hashimoto, K., Makino, H. and Shinohara, H.: A 45-nm Single-port and Dual-port SRAM family with Robust Read/Write Stabilizing Circuitry under DVFS Environment, *Symposium on VLSI Circuits Digest of Technical Papers*, pp.212–213 (2008).

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: *Ittetsu Taniguchi*)