IPSJ Transactions on System LSI Design Methodology Vol.11 46–56 (Aug. 2018)

[DOI: 10.2197/ipsjtsldm.11.46]

Regular Paper

Scalar Replacement with Polyhedral Model

Kenshu Seto[1,a)]

Received: December 16, 2017, Revised: March 9, 2018,
Accepted: April 16, 2018

Abstract: High-level synthesis (HLS) significantly reduces hardware design time. Unfortunately, the users of HLS usually have to manually rewrite algorithm C code for satisfactory synthesis results. These manual tunings of C code often cause extra design time and decrease the advantage of HLS. One of such manual tunings is array access optimization. Large arrays are implemented as RAMs in HLS, so reducing array accesses in C code can increase performance of synthesized hardware since access conflicts to the RAMs are reduced. Furthermore, the removal of all accesses to arrays leads to the complete removal of the RAMs corresponding to the arrays. By successful application of scalar replacement to C code, data read from RAMs or written to RAMs are stored in shift registers, and these shift registers are accessed instead of the RAMs when reusing the accessed data, thus array accesses are completely removed. Unfortunately, the most advanced scalar replacement method for nested loops cannot appropriately handle array accesses with constant subscripts. This paper proposes a scalar replacement method to solve the problem. In particular, we target a subset of C code called Static Control Part (SCoP) for which we can build the mathematical representation called the polyhedral model. The proposed method builds elaborate reuse information tables with the polyhedral model. Differently from the previous method, the proposed method replaces each reuse destination that has multiple reuse vectors with scalar variables. These scalar variables are referenced conditionally according to the conditions in the reuse information tables. With the experimental results, we demonstrate that the proposed method decreases the area of synthesized hardware significantly and improves circuit performance compared to the most advanced scalar replacement method for nested loops in the case of C code which contain array accesses with constant subscripts.

Keywords: high level synthesis, memory access optimization, data reuse, scalar replacement, loop pipelining

1. Introduction

High-level synthesis (HLS) [1], [2] significantly reduces hardware design time compared to manual RTL design, since HLS automatically generates RTL descriptions from C/SystemC code. HLS is useful not only for hardware engineers who have to reduce design time, but also for software engineers who try to accelerate their software with FPGAs. One of the most important steps for hardware design with HLS is the creation of input C/SystemC code. Unfortunately, HLS from software C code that is written without considering hardware implementation often leads to poor hardware. So, designing efficient hardware with current HLS requires sufficient understanding of high-level synthesis algorithms, interface options and source-level optimizations such as various loop transformations and array (memory) access optimizations and appropriate settings of synthesis directives. In particular, this paper focuses on array access optimizations. Large arrays are mapped to RAMs with limited number of ports in HLS. It is, therefore, necessary to increase the bandwidth of these RAMs by optimizing array accesses to generate high-performance hardware from C code with rich array accesses.

The main objectives of array access optimizations are increasing the bandwidth of corresponding RAMs or removing access conflicts to RAMs with limited numbers of ports. Examples of array access optimizations include array reshaping [2], memory partitioning [3], [4] and scalar replacement [6], [7], [8], [9]. Array reshaping [2] increases the bandwidth of successive accesses of array elements by increasing the word widths of RAMs and can be applied along with memory partitioning or scalar replacement. Memory partitioning [3], [4] partitions a single array into multiple smaller arrays which are accessed in parallel and can even partition arrays with non-affine indices [10]. Memory partitioning requires address generation units and multiplexers to select the partitioned arrays, which increase clock period and hardware area. Recent memory partitioning [5] reduces the amount of multiplexers by limiting the target code to stencil computations. By reducing memory access conflicts with memory partitioning, the initiation intervals (IIs) of the loop kernels after loop pipelining [2] is minimized.

Scalar replacement [6], [7], [8], [9] copies data accessed from an array into a set of scalar variables that forms a shift register. By replacing repeated accesses to the array with accesses to the scalar variables, scalar replacement reduces access conflicts to the RAMs corresponding to the array. Since scalar replacement does not require multiplexers, scalar replacement results in synthesized hardware with less area and faster clock frequency compared to memory partitioning. Furthermore, if we can replace all accesses to the array storing temporal data with the shift register accesses in C code, we can completely remove the array, or the corresponding RAM, from the code, which leads to significant chip area reduction.

It is often necessary to apply scalar replacement to nested

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

loops, such as two-dimensional image processing loops, since real-life loops commonly have nested levels of more than one. A classical scalar replacement method is proposed by Carr et al. [6]. Unfortunately, the method is applicable to the data reuse carried by innermost loops, so the chances of applying scalar replacement is limited. To enhance the chances of applying scalar replacement for nested loops, the method [6] requires a loop transformation called unroll-and-jam, which increases the code size and, hence, most likely degrades the clock frequencies of synthesized hardware. In addition, unroll-and-jam is not always applicable to C code. To resolve the problem, So et al. proposed a scalar replacement method [7] that exploits data reuse carried by multiple levels of loops, so that their scalar replacement method can be applied to arbitrarily nested loops without unroll-and-jam. Budiu et al. proposed the scalar replacement method that minimizes the numbers of dynamic memory accesses in the presence of conditional control flow [8]. Unfortunately, their method cannot always minimize the number of static memory accesses or the numbers of memory accesses in program texts, so that the initiation intervals (IIs) of the pipelined loop kernels cannot always be minimized. Recently, Surendran et al. proposed a scalar replacement method using Array SSA form [9] that handles loop kernels with conditional control flows and implemented their algorithm in the LLVM compiler infrastructure. Similar to the classical method [6], the approach [9] is applicable only to the data reuse carried by innermost loops, so that the applicability of the scalar replacement is limited. In summary, the most advanced and effective scalar replacement algorithm that handles nested loops is Ref. [7]. Unfortunately, the previous method [7] cannot apply to array accesses with constant values in array indices, so that the chances of the access conflict reduction and the RAM removal by scalar replacement [7] are limited. To resolve the limitation, this paper presents a scalar replacement method based on the polyhedral model [15] in order to handle array accesses with constant subscripts.

The organization of this paper is as follows. In Section 2, we describe the impact of scalar replacement on the hardware synthesized by HLS. In Section 3, we explain the most advanced scalar replacement method [7] and its problem. In Section 4, we present our proposed method for scalar replacement, followed by experimental results in Section 5. Finally, we conclude this paper in Section 6.

2. The Impact of Scalar Replacement on the Hardware Synthesized by HLS

In terms of hardware area, scalar replacement adds shift registers to the synthesized hardware. The addition of shift registers increases the area of synthesized hardware. Successful scalar replacement, however, removes all the array accesses to arrays, so that we can completely remove the RAMs corresponding to the arrays. If the area reduction by the removal of the RAMs is larger than the area increase by the shift registers, the hardware area is reduced after scalar replacement.

HLS infers RAMs from large arrays and these RAMs typically allow very limited numbers of accesses in each clock cycle because the number of ports in such RAMs are limited to 1 or 2. As a result, rich array accesses in C code usually degrade the performance of the synthesized hardware due to memory access conflicts. Scalar replacement is effective in reducing the execution cycles of the synthesized hardware since it removes array accesses in C code.

In this paper, we focus on fully nested loops with rich array accesses, such as image processing code, as target C code, and aim to generate hardware with highest performance with loop pipelining [2]. We perform loop pipelining to the innermost loops. One of the key parameters in loop pipelining is the initiation interval (II), which represents the fixed number of clock cycles between the successive iterations of a loop. So, II=1 is the best achievable II in loop pipelining and provides highest performance. In this paper, we minimize IIs to achieve the best performance for given C code.

For an innermost loop, it is known that the MII (Minimum Initiation Interval), which is the lower-bound of achievable IIs, is given by the following formula: MII = Max(ResMII, RecMII) [11]. RecMII is the lower-bound of IIs due to loop-carried data dependencies and it is typically zero for image processing loops because of the absence of loop-carried data dependences. On the other hand, ResMII is the lower-bound of IIs due to resource usage conflicts and it is given by the following formula:

$$\mathrm{ResMII} = \max_{r \in R} \mathrm{ResMII_r} = \max_{r \in R} \left\lceil \frac{O_r}{N_r} \right\rceil$$

where R is a set of allocated resources, such as adders, multipliers and memory ports of RAMs that implement arrays. O_r is the number of the operations in the loop body that are mapped to the resource r. N_r is the number of the allocated resource instances of the resource r. If the resource r is a functional unit, we can increase the number of the resource N_r to O_r, in order to make the value of O_r/N_r to be the minimum value 1. Unfortunately, we cannot typically increase the numbers of memory ports of large RAMs as desired, because the numbers of the memory ports are usually limited to 1 or 2. Since increasing the numbers of ports of RAMs significantly increases both the area and the power consumption of the RAMs, it is better to use single port RAMs in HLS when possible. In the case of the example code in **Fig. 1**, there are four array accesses to the array A. If we map the array A to a single port memory M and allocate sufficient resource instances for other types of resources r, $\mathrm{ResMII_M} = O_\mathrm{M}/N_\mathrm{M} = 4/1 = 4$ and $\mathrm{ResMII_r} = O_\mathrm{r}/O_\mathrm{r} = 1$, so that II equals to 4 even in the best case. On the other hand, if we can remove the array accesses as shown in the code in **Fig. 5**,

```
 1 for(y=0; y<=9; y++){
 2   for(x=0; x<=9; 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.

$\mathrm{ResMII_M} = O_\mathrm{M}/N_\mathrm{M} = 0/1 = 0$. As a result, the MII becomes 1, so that we can expect the loop pipelining with II=1 after scalar replacement and significant performance improvement.

3. The Previous Scalar Replacement Method [7] and Its Problem

We describe the most advanced scalar replacement method that effectively handles nested loops [7] in this section, followed by its problem using the example code in Fig. 1. Before the detailed description, we define some terminologies.

3.1 Preliminaries

Definition 3.1 (Reuse graph) A **reuse graph** (V, E) illustrates the possibility of data reuse between array accesses and is a directed graph where each node $v \in V$ represents an array access, and each edge $e = (s, d) \in E$ from the source node s to the destination node d means that an array element accessed by the access s will be accessed later by the access d. We call s a reuse source and d a reuse destination. **Figure 2** shows the reuse graph for the array A in the code shown in Fig. 1.

Definition 3.2 (Reuse chain) A **reuse chain** is a connected component in a reuse graph. So, Fig. 2 is not only a reuse graph but also a reuse chain.

Definition 3.3 (Generator) In a reuse graph, a special reuse source s without any incoming edge is called a **generator**. The double circle in Fig. 2 is 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 3.4 (Reuse vector) In reuse graphs, each edge $e = (s, d) \in E$ is assigned a **reuse vector** $\langle d_1, d_2, \ldots \rangle$ which means the number of loop iterations (or simply, iterations) between the access by s and the succeeding access by d. In the reuse vector, d_1, d_2, ... are the numbers of iterations for outermost loop, 2nd outermost loop, ..., respectively. In Fig. 2, the reuse vector from the access A[y][x] to the access A[y-3][x] is $\langle d_1, d_2 \rangle = \langle 3, 0 \rangle$. If a component of a reuse vector is not a single constant value but a set of multiple positive values, the component is represented by '+'. Figure 2 contains three reuse vectors with the '+' in components. In the previous method [7], reuses whose reuse vectors contain the '+' are not the target for scalar replacement.

Definition 3.5 (Reuse distance) A **reuse distance** for an edge $e = (s, d)$ is calculated from the 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.

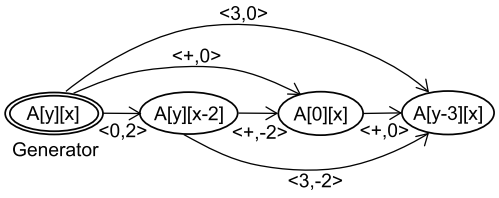

Fig. 2 Reuse graph (or reuse chain) for Fig. 1.

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

In Fig. 2, the reuse distance from the access A[y][x] to the access A[y-3][x] is $I_2 \times d_1 + d_2 = 10 \times 3 + 0 = 30$.

3.2 The Previous Scalar Replacement Method [7]

Below, we show the algorithm of the previous scalar replacement method [7].

Step 1. Perform reuse analysis.
Step 2. Build a reuse graph and partition the reuse graph to reuse chains.
Step 3. Classify the reuse chains into four types $C^{\emptyset}$, C^{C}, $C^{i\emptyset}$, C^{iC}.
Step 4. Find the unique generator for each reuse chain.
Step 5. Build the reuse information table for each reuse chain.
Step 6. Perform code transformations.
- (a) Insert the declaration of the shift registers and their shifting behavior at the bottom of the loop body.
- (b) Replace each array access with the corresponding scalar variable.

In the following, we explain the above algorithm with the example code in Fig. 1. The code in Fig. 1 has four memory accesses to the array A. One of the accesses, A[0][x], contains a constant 0 in the subscript. In **Step 1**, the method performs reuse analysis to collect the information for building a reuse graph. The previous method [7] does not analyze conditions of if conditionals that enclose reuse destinations, so that the reuse analysis results are approximate ones. As the results of the reuse analysis, a reuse graph is built in **Step 2**. Figure 2 shows the reuse graph for the array A in the code in Fig. 1. The reuse graph is a connected graph, therefore, it is a reuse chain. In **Step 3**, the reuse chain is classified as one of the four categories, namely, $C^{\emptyset}$, C^{C}, $C^{i\emptyset}$, C^{iC} [7]. Unfortunately, reuse chains with array accesses whose subscripts contain constant values deviate from the classification, so that scalar replacement of array accesses whose subscripts contain constant values, such as A[0][x] in Fig. 1, is outside the scope of the method [7]. In order to perform scalar replacement to the code in Fig. 1 with the method [7], we need to remove the array access A[0][x] from the reuse chain in Fig. 2 as shown in **Fig. 3**. The modified reuse chain in Fig. 3 is classified as C^{C}, and can be handled by the method [7]. In **Step 4**, the unique generator for the reuse chain is identified. If more than one generators exist in a reuse chain, the reuse chain is out of scope for the scalar replace by the previous method [7] and also by the proposed method. The double circle in Fig. 3 illustrates the identified generator which corresponds to the array access A[y][x]. The edge e = (A[y][x],A[y-3][x]) in Fig. 3 means that array elements that are accessed by the reuse source A[y][x] will be accessed by the reuse destination A[y-3][x] three iterations later of the outer loop (y loop). In **Step 5**, the method builds the

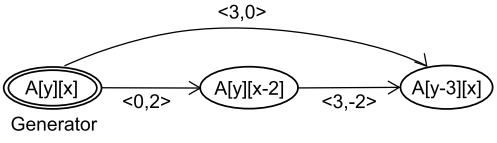

Fig. 3 Reuse chain of C^{C} type after the removal of the array access A[0][x].

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

	Access type	Array access	Reuse vector	Reuse distance	Scalar variable
1	Generator	A[y][x]	N/A	N/A	s0
2	Reuse	A[y][x-2]	⟨0, 2⟩	2	s2
3	Reuse	A[y-3][x]	⟨3, 0⟩	30	s30

reuse information table which consists of reuse vectors and reuse distances from the generator to all reuse destinations in the reuse chain. **Table 1** shows the reuse information table for the reuse chain in Fig. 3. Each reuse vector is computed as the difference of the corresponding array subscripts between the generator and each reuse, and each reuse distance is computed by the formula (1) from the corresponding reuse vector. Finally in **Step 6**, we perform code transformations to the input C code based on the reuse information table. First, in **Step 6**(a), the declarations of scalar variables corresponding to shift registers are inserted, and in addition, the behaviors of the shift registers are added at the bottom of the loop body. The number of the scalar variables for each reuse chain is determined by the maximum reuse distance in the reuse information table. For the reuse information table in Table 1, the maximum reuse distance is 30, so the length of the shift register, or the number of scalar variables, is determined as 30. In this paper, we use a prefix 's' followed by an integer representing a reuse distance to denote a scalar variable corresponding to an element in a shift register. The data accessed by the generator is stored in the scalar variable $s0$ and the data in $s0$ reaches, for example, the scalar variable $s2$ after two iterations of the innermost loop. In other words, it takes x iterations of the innermost loop before the reuse destination with the reuse distance x reuses the data $s0$ accessed by the generator. The scalar variables that constitute shift resisters are shifted at every iteration of the innermost loop. Such shifting operations are placed at the bottom of the innermost loop body, and can be done in one clock cycle in hardware. Since the length of shift register is 30, the shifting operations are generated as $s29 \rightarrow s30, s28 \rightarrow s29, \ldots, s0 \rightarrow s1$. In **Step 6**(b), each reuse destination is replaced by the scalar variable with the corresponding reuse distance. For example, reuse destinations A[y][x-2] and A[y-3][x] whose reuse distances are 2 and 30 are replaced by the scalar variables $s2$ and $s30$, respectively. The final optimized code by the previous scalar replacement [7] is shown in **Fig. 4**.

3.3 The Problem of the Previous Scalar Replacement [7]

Array accesses with constant subscripts such as A[0][x], A[15][x-1], A[y][0], A[y][14], etc., are common in image filtering algorithms in order to fill up the pixel values outside image edges where the filter kernels run off. As explained in Section 3.2, the previous method [7] cannot replace array accesses that have constant subscripts with scalar variables, since such accesses make reuse chains out of the scope for their method. Therefore, performance improvement and area reduction of synthesized hardware for such image filtering algorithms by the previous method [7] are limited. In Section 4, we propose a method to address this problem.

By applying loop distribution to the example code in Fig. 1, and splitting the fully nested loop into a sequence of four fully

```
 1 for(y=0; y<=9; y++){
 2   for(x=0; x<=9; x++){
 3     s0 =……;
 4     A[y][x] = s0;
 5     if (x>=2)   t+= s2;
 5     if (y>=1) {
 6       if (y<=2) t+= A[0][x];
 7       else      t+= s30;
 9     }
10     s30=s29;
11     s29=s28;
12     s28=s27;
       …..
38     s2=s1;
39     s1=s0;
40   }
41 }
```

Shift register operations

Fig. 4 Code after applying previous scalar replacement for example code.

nested loops with the range of y being $y == 0$, $y == 1$, $y == 2$ and $3 <= y <= 9$, respectively, we can successfully apply the previous method [7] to each of these four loops. Unfortunately, high-level synthesis of a sequence of the loops typically results in hardware which sequentially executes each loop, so the number of execution cycles will increase compared to the hardware synthesized with the proposed method in Section 4 which handles the scalar replacement of the original fully nested loop. In addition, the application of loop distribution will increase the code size, which is likely to degrade the maximum clock frequency of the synthesized hardware.

4. Proposed Scalar Replacement Method with Polyhedral Model

A careful observation on the code in Fig. 1 reveals that the array access A[0][x] in line 6 occurs only when $y == 1$ or $y == 2$. In addition, the data accessed by A[0][x] is accessed in advance by the generator A[y][x] in line 3 when $y == 0$. So, we can expect that A[0][x] can reuse the data accessed by the generator. In this section, we propose a scalar replacement method with the polyhedral model that can handle such array accesses with constant subscripts.

4.1 Target Code for the Proposed Method

The Static Control Part (SCoP) is a subset of C code that translates into the polyhedral model and is the target for polyhedral-model based loop optimizations [15]. Differently from the previous method [7], we assume that the input C code is a stencil code [5] in the form of a static control part (SCoP) [13] represented by a fully nested loop where all statements are included only in the innermost loop. A sequence of loops with the same nested levels can be fused into a fully nested loop by using loop fusion [12].

A SCoP consists of for-loops or if-conditionals or both in arbitrary nested form, where the following expressions are limited to affine expressions in terms of loop induction variables of the enclosing for-loops.

- condition expressions of for-loops and if-conditionals
- lower-bound and upper-bound expressions of for-loops
- array subscripts

C code in the form of a SCoP is translated into the polyhedral model and benefits from exact array dataflow analysis [18], [19] which is important for the proposed scalar replacement. We assume that lower-bound and upper-bound expressions of for-loops are constant values. C code that deviates from the above limitations may be handled as a SCoP by approximation [20] or compiler transformations [21]

Array accesses in loops are classified as one of the two categories: static array accesses and dynamic array accesses. Static array accesses in C code is array accesses that appear in program texts (or C code). Dynamic array accesses in C code is array access instances that are actually executed while the execution of the C code. A static array access typically generates multiple dynamic array accesses. For reducing initiation intervals (IIs) of loop pipelining, it is important to reduce the number of static array accesses in the loop body as explained in Section 2. In this paper, array accesses mean static array accesses.

Data reuses between array accesses are classified as one of the two types: group-temporal reuse and self-temporal reuse [7]. Group-temporal reuses are data reuses between different array accesses. On the other hand, self-temporal reuses are data reuses by the same array access. Exploiting self-temporal reuses in scalar replacement does not remove the corresponding RAMs, so we only focus on scalar replacement for array accesses with group-temporal reuses.

4.2 The Polyhedral Model for Scalar Replacement

In this subsection, we explain the definitions of the polyhedral model which is used in the proposed method. Typically, the polyhedral model defines three key concepts: *domains*, *access functions* and *multi-dimensional schedules* for statements in C code and is used for statement reordering, or equivalently, loop transformation [13], [14]. Differently from the previous definitions of the polyhedral model for statements [13], [14], we define the polyhedral model especially for array accesses in order to use the model for the proposed scalar replacement. The polyhedral model for scalar replacement defined below is represented or computed with ISL (Integer Set Library) [18], [19].

Definition 4.1 (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**. For the 2-dimensional loop in Fig. 1, the iteration vectors are $\vec{i} = (y, x) = (0, 0), \ldots, (9, 9)$.

Definition 4.2 (Domain of array access) A set of iteration vectors, D_a, that contains all iterations at which an array access a in a loop is executed is called the **domain** of the array access a. For Fig. 1, the domain $D_{A[0][x]}$ of the access A[0][x] is $\{(y, x) \mid 1 \leq y \leq 2 \wedge 0 \leq x \leq 9\}$ where $\wedge$ represents logical AND.

Definition 4.3 (Subscript vector of array) We define a vector $\vec{s}$ whose elements are subscript values of an array as a **subscript vector**. The number of dimensions of subscript vectors 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 specific subscript values of an array access. For the 2-dimensional array A in Fig. 1, the array subscript vectors are $\vec{s} = (s_1, s_2) = (0, 0), \ldots, (9, 9)$.

Definition 4.4 (Subscript space of array) We call a set of all subscript vectors of an array A a **subscript space** of the array A and denote it by S_A. For the 2-dimensional array A in Fig. 1, the subscript space S_A of the array A is $\{(s_1, s_2) \mid 0 \leq s_1 \leq 9 \wedge 0 \leq s_2 \leq 9\}$.

Definition 4.5 (Access function of array access) The **access function** F_a of an array access a to an m-dimensional array 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 A. For Fig. 1, the access function of the access A[0][x] is $F_{A[0][x]} : \{(y, x) \rightarrow (s_1, s_2) \mid 1 \leq y \leq 2 \wedge 0 \leq x \leq 9 \wedge s_1 == 0 \wedge s_2 == x\}$.

Definition 4.6 (Multi-dimensional schedule of array access) The **multi-dimensional schedule** [15] ϕ_a of an array access a is a function $\phi_a : \vec{i} \in D_a \rightarrow \vec{t} \in T_a$ where T_a represents the set of multi-dimensional discrete time when the array access a at the iteration $\vec{i}$ is executed and $\vec{t}$ is represented by a vector which has alternating components of constants and loop induction variables from the outermost loop. For Fig. 1, the multi-dimensional schedule of the access A[0][x] is $\phi_{A[0][x]} : \{(y, x) \rightarrow (0, y, 0, x, 3) \mid 1 \leq y \leq 2 \wedge 0 \leq x \leq 9\}$. For each iteration vector $(y, x) \in D_{A[0][x]}$, the multi-dimensional schedule $\phi_{A[0][x]}$ gives the multi-dimensional time $(0, y, 0, x, 3)$ which represents the time when the array access A[0][x] is executed at the iteration vector $(y, x) \in D_{A[0][x]}$.

The polyhedral model typically defines the dependence relation (Read-after-Write, Read-after-Read, Write-after-Read, and Write-after-Write) which represents the condition under which a statement S_1 depends on a statement S_2 and is used in loop transformations. In the proposed method, we define a similar concept called **reuse relations** (Read-after-Write, Read-after-Read) which represents the condition under which a reuse destination d reuses the accessed data by a reuse source s.

Definition 4.7 (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 $\vec{i_s} = (i_{s1}, i_{s2}, \ldots, i_{sn}) \in D_s$ and $\vec{i_d} = (i_{d1}, i_{d2}, \ldots, i_{dn}) \in D_d$ 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 $\vec{i_s} \in D_s$. The reuse relation is formally defined as $R_{s,d} = \{(\vec{i_s}, \vec{i_d}) \mid \vec{i_s} \in D_s \wedge \vec{i_d} \in D_d \wedge \phi_s(\vec{i_s}) \prec \phi_d(\vec{i_d}) \wedge \exists \vec{s} \in S_A, (F_s(\vec{i_s}) = \vec{s} \wedge F_d(\vec{i_d}) = \vec{s})\}$ where $\phi_s(\vec{i_s}) \prec \phi_d(\vec{i_d})$ means that the schedule $\phi_d(\vec{i_d})$ of the reuse destination d at the iteration $\vec{i_d}$ is lexicographically later than the schedule $\phi_s(\vec{i_s})$ of the reuse source s at the iteration $\vec{i_s}$. Differently from the previous method [7], our method analyze exact reuse relations with the polyhedral model. For Fig. 1, the reuse relation $R_{A[y][x],A[0][x]}$ from the reuse source A[y][x] to the reuse destination A[0][x] is $\{(y_s, x_s), (y_d, x_d) \mid (y_s = 0) \wedge (x_d == x_s) \wedge (0 \leq x_s \leq 9) \wedge (1 \leq y_d \leq 2)\}$.

Definition 4.8 (Reuse vector between array accesses) For a pair of iteration vectors $(\vec{i_s}, \vec{i_d})$ in a reuse relation $R_{s,d}$, the difference $\vec{i_d} - \vec{i_s}$ is called a **reuse vector** from the access s to the access d. A set of **reuse vectors**, or simply, **reuse vectors** from a reuse source s to a reuse destination d is represented by $V_{s,d} = \{\vec{i_d} - \vec{i_s} \mid (\vec{i_s}, \vec{i_d}) \in R_{s,d}\}$. For Fig. 1, the set of reuse vectors from the reuse source A[y][x] to the reuse destination A[0][x] is

$V_{A[y][x],A[0][x]} = \{(y_d - y_s, 0) : 1 \leq (y_d - y_s) \leq 2\}$ which consists of two reuse vectors $(1, 0)$ and $(2, 0)$. On the other hand, the set of reuse vectors from the array access A[y][x] to the array access A[y][x-2] is $V_{A[y][x],A[y][x-2]} = \{(0, 2)\}$ which consists of a single reuse vector $(0, 2)$.

4.3 The Proposed Method

In this section, we present our proposed algorithm for scalar replacement. The input to the algorithm is C code in the form of a SCoP, and the output is the optimized C code after scalar replacement.

The major differences of the proposed scalar replacement algorithm from the previous algorithm [7] in Section 3.2 are in Step 1, Step 5 and Step 6(b) in the following algorithm. By the differences, the proposed scalar replacement method applies not only to reuse destinations with a single reuse vector but also to reuse destinations, such as A[0][x], with multiple reuse vectors. Array accesses with constant subscripts, therefore, are handled by the proposed method.

Step 1. Build the polyhedral model for array accesses.
Step 2. Perform reuse analysis with the polyhedral model.
Step 3. Build a reuse graph and partition the reuse graph to reuse chains.
Step 4. Find the unique generator for each reuse chain.
Step 5. Build the reuse information table for each reuse chain with the polyhedral model.
Step 6. Perform code transformations.
(a) Insert the declaration of the shift registers and their shifting behavior at the bottom of the loop body.
(b) Replace each array access with the corresponding scalar variables with the conditions.

Our tool parses an input C code in the form of a SCoP and builds the polyhedral model for array accesses (**Step 1**) explained in Section 4.2. The polyhedral model is the key to **Step 5** which is one of the important steps in the proposed method. In **Step 2**, we perform reuse analysis. With the polyhedral model, the proposed method takes if conditionals that enclose reuse destinations into consideration, so that exact reuse graphs without false reuse edges can be built in **Step 3**. Except the differences in the resulting reuse graphs and reuse chains, the **Step 3** is the same as the **Step 2** in the previous method [7]. For the example code in Fig. 1, our method generates the same reuse graph as the one shown in Fig. 2 except that all edges have reuse relations defined in Definition 4.7. The proposed method does not have the classification step (**Step 3** in Section 3.2), since the proposed method does not use it. **Step 4** is the same as the previous method. In **Step 5**, we compute the reuse information table for each reuse chain with the polyhedral model. The proposed polyhedral-model based algorithm for **Step 5** is shown in **Fig. 7**, which will be explained later in Section 4.4. The reuse information table contains not only reuse vectors and reuse distances but also reuse conditions in terms of the domains of reuse destinations. For the code in Fig. 1, the reuse information table generated by the **Step 5** is shown in **Table 2**. In the table, a single access A[0][x] with the constant subscript 0 has the two reuses with reuse vectors $\langle 1, 0\rangle$ and $\langle 2, 0\rangle$ and the corresponding reuse conditions of $y == 1$ and $y == 2$, respectively. Finally, in **Step 6**, we perform code transformations and obtain the optimized C code. **Step 6**(a) is the same as the previous method. Differently from the previous method [7], our method replaces a reuse destination that has multiple reuse vectors with the corresponding multiple scalar variables in **Step 6**(b). The final optimized code is shown in Fig. 5. In the lines 7 and 8 of the code, the array access A[0][x] with the constant subscript 0 is conditionally replaced by the two scalar variables s10 and s20 with reuse conditions $y == 1$ and $y == 2$, respectively. In the code, all array accesses to array A, and consequently the RAM corresponding to the array A, are completely removed, so that the proposed scalar replacement achieves significant area reduction as well as performance improvement in synthesized hardware as will be demonstrated by the experimental results in Section 5.

Table 2 The reuse information table for Fig. 1 by the proposed method.

	Access type	Array access	Reuse vector	Reuse distance	Scalar variable	Reuse condition
1	Generator	A[y][x]	N/A	N/A	s0	N/A
2	Reuse	A[y][x-2]	$\langle 0, 2\rangle$	2	s2	always
3	Reuse	A[0][x]	$\langle 1, 0\rangle$	10	s10	y==1
4	Reuse	A[0][x]	$\langle 2, 0\rangle$	20	s20	y==2
5	Reuse	A[y-3][x]	$\langle 3, 0\rangle$	30	s30	always

```
 1 for(y=0; y<=9; y++){
 2   for(x=0; x<=9; x++){
 3     s0=……;
 4     if (x>=2)  t+= s2;
 5     if (y>=1) {
 6       if (y<=2) {
 7         if (y==1)       s10_s20=s10;
 8         else if (y==2) s10_s20=s20;
 9         t += s10_s20;
10       }
11       else  t+=s30;
12     }
13     s30=s29;
14     s29=s28;
15     s28=s27;
       …..
41     s2 = s1;
42     s1 = s0;
43   }
44 }
```

Shift registers operations

Fig. 5 Code after applying proposed scalar replacement for example code.

Algorithm *ReuseRelation*

ComputeReuseRelation(s, d)
Input D_s, F_s, ϕ_s: Polyhedral model for an access s
Input D_d, F_d, ϕ_d: Polyhedral model for an access d
Output $R_{s,d}$: Reuse relation from the access s to the access d
1: $temp_{s,d} \leftarrow F_s \circ {F_d}^{-1}$
2: $Order \leftarrow \{(\vec{i_s}, \vec{i_d}) \mid \vec{i_s} \in D_s \wedge \vec{i_d} \in D_d \wedge \phi_s(\vec{i_s}) \prec \phi_d(\vec{i_d})\}$
3: $R_{s,d} \leftarrow temp_{s,d} \cap Order$

Fig. 6 Procedure for computing the reuse relation from an array access s to an array access d.

4.4 The Algorithm for Building Reuse Information Tables

In this subsection, we explain **Step 5** in Section 4.3 in detail. Before computing reuse information tables such as the one shown in Table 2, we compute reuse relations by the algorithm shown in **Fig. 6**, according to Definition 4.7. In Fig. 6, ${F_d}^{-1}$ in the line 1 is the inverse function of the array access function F_d. The

Algorithm *BuildReuseInformationTable*

BuildReuseInformationTable(A_C, $R_{g,a}$)
Input A_C: A set of array accesses in a reuse chain C
Input $R_{g,a}$: Reuse relations from the generator g in C to each array access a in C
Output The reuse information table for the reuse chain C
1: **for each** array access $a \in A_C$
2: $ReuseVectors \leftarrow \{\vec{i_a} - \vec{i_g} : (\vec{i_g}, \vec{i_a}) \in R_{g,a}\}$
3: **while** ($ReuseVectors \neq \emptyset$)
4: $\vec{v} \leftarrow$ SamplePoint($ReuseVectors$)
5: Compute reuse relation $R_{\vec{v}} \subseteq R_{g,a}$ whose reuse vector is $\vec{v}$
6: $Ran \leftarrow$ Range($R_{\vec{v}}$)
7: $ReuseCond \leftarrow$ gist(Ran, D_a)
8: $ReuseDistance \leftarrow$ ComputeReuseDistance($\vec{v}$)
9: AddReuse(a, $\vec{v}$, $ReuseDistance$, $ReuseCond$)
10: $ReuseVectors = ReuseVectors - \vec{v}$
11: **end while**
12: **end for**

Fig. 7 Procedure for computing reuse vectors, reuse distances and reuse conditions of array accesses in reuse information tables.

line 1 in Fig. 6 computes the composition of the functions F_s and ${F_d}^{-1}$. The line 2 in Fig. 6 computes the set $Order$ that consists of all pairs $(\vec{i_s}, \vec{i_d})$ of iteration vectors $\vec{i_s} \in D_s$ and $\vec{i_d} \in D_d$ such that the multi-dimensional schedule $\phi_s(\vec{i_s})$ is lexicographically smaller than the multi-dimensional schedule $\phi_d(\vec{i_d})$. By the line 3 in Fig. 6, all the pairs $(\vec{i_s}, \vec{i_d})$ that are not only in $temp_{s,d}$ but also in $Order$ are stored in the resulting reuse relation $R_{s,d}$.

Figure 7 shows the proposed algorithm for building reuse information tables. We implemented the algorithm with Integer Set Library (ISL) [18], [19]. A reuse information table is built for each reuse chain C. The inputs to the algorithm are (1) A_C : a set of all array accesses except the generator in a reuse chain C and (2) $R_{g,a}$: the reuse relations between the generator g in C and each array access a in C. The algorithm in Fig. 7 uses reuse relations computed by the algorithm in Fig. 6 and processes each access a in A_C one by one. The line 2 in Fig. 7 computes the set of all reuse vectors, $ReuseVectors$, from the generator g to the access a. While $ReuseVectors$ is not empty, the algorithm picks up a reuse vector $\vec{v}$ in $ReuseVectors$ one by one in the lines 3 to 4. The algorithm processes the reuse vector according to the lines 5 to 9 in Fig. 7. The line 5 in Fig. 7 computes the reuse relation $R_{\vec{v}}$ which is a subset of the reuse relation $R_{g,a}$. $R_{\vec{v}}$ has all the elements in $R_{g,a}$ that have only one reuse vector $\vec{v}$ in common. The line 6 in Fig. 7 computes the range Ran of the reuse relation $R_{\vec{v}}$, which is a subset of D_a. The computed range Ran represents the reuse condition $ReuseCond$ under which the reuse from the generator g to the access a occurs with the reuse vector $\vec{v}$. The line 7 in Fig. 7 simplifies the reuse condition $ReuseCond$ in terms of the iteration domain D_a of the access a by the gist operation [22]. The line 8 in Fig. 7 computes the reuse distance corresponding to the reuse vector $\vec{v}$ with the formula (1) in Section 3.1. The line 9 in Fig. 7 adds the computed 3-tuple of (a reuse vector $\vec{v}$, $ReuseDistance$, $ReuseCond$) for the access a to the reuse information table. The line 10 in Fig. 7 subtracts the reuse vector $\vec{v}$ from $ReuseVectors$.

5. Experimental Results

In this section, we show the impacts of the proposed scalar replacement method on hardware performance and area. In particular, we compare these design metrics by the proposed method with those by the most advanced scalar replacement method for nested loops [7].

```
for (y = 0; y <= 9; y++) {
 for (x = 0; x <= 9; x++) {
  A[y][x] = in[y][x];
  if (y>=6) t = t + A[y-6][x];
  if (y>=1) {
   if (y<=2) t = t + A[0][x];
   else t = t + A[y-3][x];
  }
 }
}
```

Fig. 8 Benchmark code (ex_d60).

```
for (y = 0; y <= 29; y++) {
 for (x = 0; x <= 29; x++) {
  A[y][x] = in[y][x];
  if (y>=1) t = t + A[y-1][x];
  if (x>=1) {
   if (x<=2) t = t + A[y][0];
   else t = t + A[y][x-3];
  }
 }
}
```

Fig. 9 Benchmark code (ex1).

```
for (y = 0; y <= 29; y++) {
 for (x = 0; x <= 29; x++) {
  A[y][x] = in[y][x];
  if (y>=1) t = t + A[y-1][x];
  if (x>=1) {
   if (x<=5) t = t + A[y][0];
   else t = t + A[y][x-6];
  }
 }
}
```

Fig. 10 Benchmark code (ex1_x5).

```
for (y = 0; y <= 29; y++) {
 for (x = 0; x <= 29; x++) {
  A[y][x] = in[y][x];
  if (y>=1) t = t + A[y-1][x];
  if (x>=1) {
   if (x<=10) t = t + A[y][0];
   else t = t + A[y][x-11];
  }
 }
}
```

Fig. 11 Benchmark code (ex1_x10).

5.1 Experimental Setups

We implemented the proposed method in Section 4 in C with the libraries Clan [15], CLooG [17] and ISL [18]. We applied the proposed method and the previous method [7] to eight image processing code shown in Fig. 1, **Figs. 8** to **14**. The reuse information tables for each code are shown in **Tables 3** to **9**. Each of the code consists of a fully nested loop with the nested level of two, although the proposed method is applicable to fully nested

```
for(y=0 ; y<13 ; y++) {
 for(x=0 ; x<16 ; x++) {
  if(y <= 11)
   tmp0[y][x] = ...
  if(y >= 1){
   if(y-1 < 1)//y=1
    itmp0 = tmp0[0][x];
   else
    itmp0 = tmp0[y-1-1][x];
   itmp1 = tmp0[y-1][x];
   if(y-1 >= 11)
    itmp2 = tmp0[11][x];
   else
    itmp2 = tmp0[y-1+1][x];
   tmp1[y-1][x] = ...
  }
 }
}
```

Fig. 12 Benchmark code (ex2).

```
for (y=0;y<=16;y++) {
 for (x=0;x<=16;x++) {
  if ((y <= 15) && (x <= 15))
   tmp0[y][x]=in[y][x];
  if ((y <= 15) && (x >= 1) && (x <= 15))
   itmp2=tmp0[y][x-1 +1];
  if ((y <= 15) && (x >= 1))
   tmp1[y][x-1]=...
  if ((y >= 1) && (y <= 15) && (x >= 1))
   ttmp2=tmp1[y-1 +1][x-1];
  if ((y >= 1) && (x >= 1))
   out[y-1][x-1]=...
  if ((y == 16) && (x <= 15))
   ttmp2=tmp1[15][x];
  if ((y >= 1) && (x <= 15))
   ttmp1=tmp1[y-1][x];
  if ((y >= 2) && (x <= 15))
   ttmp0=tmp1[y-2][x];
  if ((y == 1) && (x <= 15))
   ttmp0=tmp1[0][x];
  if ((y <= 15) && (x == 15))
   itmp2=tmp0[y][15];
  if ((y <= 15) && (x >= 1) && (x <= 15))
   itmp0=tmp0[y][x-1];
  if ((y <= 15) && (x <= 15))
   itmp1=tmp0[y][x];
  if ((y <= 15) && (x == 0))
   itmp0=tmp0[y][0];
 }
}
```

Fig. 13 Benchmark code (filter).

loops with any nested levels. We did not use the benchmark programs in Ref. [7], since the programs do not contain array accesses with constant subscripts. *ex* is the example code shown in Fig. 1. *ex_d60* is a modified version of *ex* and the maximum reuse distance is 60 as shown in Table 3, which is larger than that of *ex* which is 30. *ex1*, *ex1_x5* and *ex_x10* are also modified versions of *ex* and their maximum reuse distances are the same. The reuse conditions of these three programs, however, are different as shown in Tables 4 to 6. *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 [12]. *loop4* is an image processing code originally consisting of four loops,

```
for (y=0;y<=13;y++) {
 for (x=0;x<=18;x++) {
  if ((y >= 2) && (x >= 3) && (x <= 16))
   itm25=tmp1[y-2][x-1];
  if ((y >= 2) && (x >= 3) && (x <= 17))
   itm24=tmp1[y-2][x-2];
  if ((y >= 2) && (x >= 3))
   itm23=tmp1[y-2][x-3];
  if ((y >= 2) && (x == 3))
   itm22=tmp1[y-2][0];
  if ((y >= 2) && (x >= 3) && (x <= 4))
   itm21=tmp1[y-2][0];
  if ((y >= 2) && (x >= 3) && (x <= 5))
   itm20=tmp1[y-2][0];
  if ((y >= 12) && (x <= 15))
   itm14=tmp0[11][x];
  if ((y >= 2) && (y <= 12) && (x <= 15))
   itm13=tmp0[y-1][x];
  if ((y == 13) && (x <= 15))
   itm13=tmp0[11][x];
  if ((y >= 2) && (x <= 15))
   itm12=tmp0[y-2][x];
  if ((y >= 3) && (x <= 15))
   itm11=tmp0[y-3][x];
  if ((y == 2) && (x <= 15))
   itm11=tmp0[0][x];
  if ((y >= 4) && (x <= 15))
   itm10=tmp0[y-4][x];
  if ((y >= 2) && (y <= 3) && (x <= 15))
   itm10=tmp0[0][x];
  if ((y <= 11) && (x <= 15))
   tmp0[y][x]=src0[y][x]-src1[y][x];
  if ((y >= 2) && (y <= 11) && (x <= 15))
   itm14=tmp0[y][x];
  if ((y >= 2) && (x <= 15))
   tmp1[y-2][x]=...
  if ((y >= 2) && (x >= 3) && (x <= 15))
   itm26=tmp1[y-2][x];
  if ((y >= 2) && (x >= 3))
   tmp2[y-2][x-3]=...
  if ((y >= 2) && (x >= 3))
   tmp3[y-2][x-3]=src1[y-2][x-3]+tmp2[y-2][x-3];
  if ((y >= 2) && (x >= 15) && (x <= 17))
   itm26=tmp1[y-2][15];
  if ((y >= 2) && (x >= 16) && (x <= 17))
   itm25=tmp1[y-2][15];
  if ((y >= 2) && (x == 17))
   itm24=tmp1[y-2][15];
  if ((y >= 2) && (x >= 3) && (x <= 17))
   itm22=tmp1[y-2][x-3];
  if ((y >= 2) && (x >= 4) && (x <= 17))
   itm21=tmp1[y-2][x-4];
  if ((y >= 2) && (x >= 5) && (x <= 17))
   itm20=tmp1[y-2][x-5];
 }
}
```

Fig. 14 Benchmark code (loop4).

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 [12].

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

Table 3 The reuse information table for benchmark *ex_d60*.

	Access type	Array access	Reuse vector	Reuse distance	Reuse condition
ex_d60	Generator	A[y][x]	N/A	N/A	N/A
	Reuse	A[y-6][x]	⟨6, 0⟩	60	always
	Reuse	A[0][x]	⟨1, 0⟩	10	y==1
	Reuse	A[0][x]	⟨2, 0⟩	20	y==2
	Reuse	A[y-3][x]	⟨3, 0⟩	30	always

Table 4 The reuse information table for benchmark *ex1*.

	Access type	Array access	Reuse vector	Reuse distance	Reuse condition
ex1	Generator	A[y][x]	N/A	N/A	N/A
	Reuse	A[y-1][x]	⟨1, 0⟩	30	always
	Reuse	A[y][0]	⟨0, 1⟩	1	x==1
	Reuse	A[y][0]	⟨0, 2⟩	2	x==2
	Reuse	A[y][x-3]	⟨0, 3⟩	3	always

Table 5 The reuse information table for benchmark *ex1_x5*.

	Access type	Array access	Reuse vector	Reuse distance	Reuse condition
ex1_x5	Generator	A[y][x]	N/A	N/A	N/A
	Reuse	A[y-1][x]	⟨1, 0⟩	30	always
	Reuse	A[y][0]	⟨0, 1⟩	1	x==1
	Reuse	A[y][0]	⟨0, 2⟩	2	x==2
	Reuse	A[y][0]	⟨0, 3⟩	3	x==3
	Reuse	A[y][0]	⟨0, 4⟩	4	x==4
	Reuse	A[y][0]	⟨0, 5⟩	5	x==5
	Reuse	A[y][x-6]	⟨0, 6⟩	6	always

Table 6 The reuse information table for benchmark *ex1_x10*.

	Access type	Array access	Reuse vector	Reuse distance	Reuse condition
ex1_x10	Generator	A[y][x]	N/A	N/A	N/A
	Reuse	A[y-1][x]	⟨1, 0⟩	30	always
	Reuse	A[y][0]	⟨0, 1⟩	1	x==1
	Reuse	A[y][0]	⟨0, 2⟩	2	x==2
	Reuse	A[y][0]	⟨0, 3⟩	3	x==3
	Reuse	A[y][0]	⟨0, 4⟩	4	x==4
	Reuse	A[y][0]	⟨0, 5⟩	5	x==5
	Reuse	A[y][0]	⟨0, 6⟩	6	x==6
	Reuse	A[y][0]	⟨0, 7⟩	7	x==7
	Reuse	A[y][0]	⟨0, 8⟩	8	x==8
	Reuse	A[y][0]	⟨0, 9⟩	9	x==9
	Reuse	A[y][0]	⟨0, 10⟩	10	x==10
	Reuse	A[y][x-11]	⟨0, 11⟩	11	always

Table 7 The reuse information table for benchmark *ex2*.

	Access type	Array access	Reuse vector	Reuse distance	Reuse condition
ex2	Generator	tmp0[y][x]	N/A	N/A	N/A
	Reuse	tmp0[0][x]	⟨1, 0⟩	16	always
	Reuse	tmp0[y-2][x]	⟨2, 0⟩	32	always
	Reuse	tmp0[y-1][x]	⟨1, 0⟩	16	always
	Reuse	tmp0[11][x]	⟨1, 0⟩	16	always
	Reuse	tmp0[y][x]	⟨0, 0⟩	0	always

Table 8 The reuse information table for benchmark *filter*.

	Access type	Array access	Reuse vector	Reuse distance	Reuse condition
filter	Generator	tmp0[y][x]	N/A	N/A	N/A
	Reuse	tmp0[y][x]	⟨0, 0⟩	0	always
	Reuse	tmp0[y][15]	⟨0, 0⟩	0	always
	Reuse	tmp0[y][x-1]	⟨0, 1⟩	1	always
	Reuse	tmp0[y][x]	⟨0, 0⟩	0	always
	Reuse	tmp0[y][0]	⟨0, 0⟩	0	always
	Generator	tmp1[y][x-1]	N/A	N/A	N/A
	Reuse	tmp1[y][x-1]	⟨0, 0⟩	0	always
	Reuse	tmp1[15][x]	⟨1, -1⟩	16	always
	Reuse	tmp1[y-1][x]	⟨1, -1⟩	16	always
	Reuse	tmp1[y-2][x]	⟨2, -1⟩	33	always
	Reuse	tmp1[0][x]	⟨1, -1⟩	16	always

Table 9 The reuse information table for benchmark *loop4*.

	Access type	Array access	Reuse vector	Reuse distance	Reuse condition
loop4	Generator	src1[y][x]	N/A	N/A	N/A
	Reuse	src1[y-2][x-3]	⟨2, 3⟩	41	always
	Generator	tmp0[y][x]	N/A	N/A	N/A
	Reuse	tmp0[11][x]	⟨1, 0⟩	19	y==12
	Reuse	tmp0[11][x]	⟨2, 0⟩	38	y==13
	Reuse	tmp0[y-1][x]	⟨1, 0⟩	19	always
	Reuse	tmp0[11][x]	⟨2, 0⟩	38	always
	Reuse	tmp0[y-2][x]	⟨2, 0⟩	38	always
	Reuse	tmp0[y-3][x]	⟨3, 0⟩	57	always
	Reuse	tmp0[0][x]	⟨2, 0⟩	38	always
	Reuse	tmp0[y-4][x]	⟨4, 0⟩	76	always
	Reuse	tmp0[0][x]	⟨2, 0⟩	38	always
	Reuse	tmp0[0][x]	⟨3, 0⟩	57	always
	Reuse	tmp0[y][x]	⟨0, 0⟩	0	always
	Generator	tmp1[y-2][x]	N/A	N/A	N/A
	Reuse	tmp1[y-2][x-1]	⟨0, 1⟩	1	always
	Reuse	tmp1[y-2][x-2]	⟨0, 2⟩	2	always
	Reuse	tmp1[y-2][x-3]	⟨0, 3⟩	3	always
	Reuse	tmp1[y-2][0]	⟨0, 3⟩	3	always
	Reuse	tmp1[y-2][0]	⟨0, 3⟩	3	x==3
	Reuse	tmp1[y-2][0]	⟨0, 3⟩	3	x==3
	Reuse	tmp1[y-2][0]	⟨0, 4⟩	4	x==4
	Reuse	tmp1[y-2][0]	⟨0, 3⟩	3	x==3
	Reuse	tmp1[y-2][0]	⟨0, 4⟩	4	x==4
	Reuse	tmp1[y-2][0]	⟨0, 5⟩	5	x==5
	Reuse	tmp1[y-2][x]	⟨0, 0⟩	0	always
	Reuse	tmp1[y-2][15]	⟨0, 0⟩	0	x==15
	Reuse	tmp1[y-2][15]	⟨0, 1⟩	1	x==16
	Reuse	tmp1[y-2][15]	⟨0, 2⟩	2	x==17
	Reuse	tmp1[y-2][15]	⟨0, 1⟩	1	x==16
	Reuse	tmp1[y-2][15]	⟨0, 2⟩	2	x==17
	Reuse	tmp1[y-2][15]	⟨0, 2⟩	2	always
	Reuse	tmp1[y-2][x-3]	⟨0, 3⟩	3	always
	Reuse	tmp1[y-2][x-4]	⟨0, 4⟩	4	always
	Reuse	tmp1[y-2][x-5]	⟨0, 5⟩	5	always
	Generator	tmp2[y-2][x-3]	N/A	N/A	N/A
	Reuse	tmp2[y-2][x-3]	⟨0, 0⟩	0	always

original code, 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 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 were mapped to single-port RAMs.

5.2 Results and Discussions

Table 10 shows the experimental results for the eight benchmark code including array accesses with constant subscripts. In the table, *no SR*, *previous*, *proposed* show the synthesis results for the original benchmark code without scalar replacement (SR), the code optimized by the previous SR method [7] and the code optimized by the proposed SR method, respectively.

All the designs satisfied the clock constraints of 500 MHz. From the table, we see that scalar replacement reduces initiation intervals (IIs) in loop pipelining, and hence, the numbers of execution cycles by removing memory accesses. Compared to the previous method [7], the proposed method enhances the performance improvement, since the proposed method removes array accesses with constant subscripts. For example, for *loop4*, the proposed method achieves 2.81x speedup compared to the previous method [7]. Both *ex* and *ex_d60* had the same numbers of execution cycles and satisfied the clock constraint of 500 MHz. So,

Table 10 Comparison between the previous method [7] and the proposed method.

Benchmark code	Code type	II [cycles]	# of execution cycles [cycles]	Speedup	Total gate counts [gates]	# of shift register bits [bits]	# of RAM bits [bits]
ex	*no SR*	4	401	1	10,829.8 (1.00)	0	6,400
	previous	2	221	1.81	18,527.9 (1.71)	960	6,400
	proposed	1	101	3.97	13,342.1 (1.23)	960	3,200
ex_d60	*no SR*	4	401	1	10,809.0 (1.00)	0	6,400
	previous	2	221	1.81	25,154.8 (2.33)	1,920	6,400
	proposed	1	101	3.97	19,795.7 (1.83)	1,920	3,200
ex1	*no SR*	4	3,601	1	87,768.0 (1.00)	0	57,600
	previous	2	1,861	1.93	95,624.2 (1.09)	960	57,600
	proposed	1	901	4.00	51,775.3 (0.59)	960	28,800
ex1_x5	*no SR*	4	3,601	1	87,776.6 (1.00)	0	57,600
	previous	2	1,861	1.93	95,640.5 (1.09)	960	57,600
	proposed	1	901	4.00	51,954.9 (0.59)	960	28,800
ex1_x10	*no SR*	4	3,601	1	87,773.5 (1.00)	0	57,600
	previous	2	1,861	1.93	95,631.0 (1.09)	960	57,600
	proposed	1	901	4.00	52,131.0 (0.59)	960	28,800
ex2	*no SR*	4	899	1	12,206.8 (1.00)	0	6,144
	previous	2	457	1.97	20,322.5 (1.67)	1,024	6,144
	proposed	1	236	3.81	14,506.2 (1.19)	1,024	3,072
filter	*no SR*	6	1,788	1	53,567.2 (1.00)	0	32,768
	previous	2	614	2.91	63,112.9 (1.18)	1,088	32,768
	proposed	1	342	5.23	38,388.4 (0.72)	1,088	16,384
loop4	*no SR*	14	3,880	1	87,832.9 (1.00)	0	51,072
	previous	3	912	4.26	119,388.1 (1.36)	3,904	51,072
	proposed	1	324	11.98	77,143.6 (0.88)	3,904	25,536

the increase in the maximum reuse distance did not degrade the performance in these two benchmark code. Although the high-level synthesis results of *ex1*, *ex1_x5* and *ex_x10* contained multiplexers with 2 inputs, 5 inputs and 10 inputs due to the different reuse conditions, the numbers of the execution cycles were exactly the same. In addition, *ex1*, *ex1_x5* and *ex_x10* satisfied the clock constraint of 500 MHz. So the changes in the reuse conditions, or equivalently, the numbers of multiplexer inputs, did not degrade the performance in these three benchmark code.

Table 10 also shows the total gate counts including all RAMs in terms of NAND2 gates. We approximated the gate counts for RAMs by 1.5 NAND gates per 1 bit of a RAM. Scalar replacement introduced shift registers as shown in Table 10. The previous [7] and the proposed scalar replacement require the same numbers of shift register bits because the maximum reuse distances for the corresponding reuse chains were the same. Compared to the synthesis results for the original code without scalar replacement, the synthesis results for the code with the previous scalar replacement [7] showed increased total gate counts because of the added shift registers. In addition, the total gate counts after the previous method increased because of the increases in the numbers of functional units that were necessary for the parallel execution with the reduced IIs. Differently from the previous method [7], the proposed method completely removed the internal RAMs storing temporary results, so that the numbers of the RAM bits in the synthesized hardware by the proposed method were significantly less than those by the previous method [7]. Furthermore, in the proposed method, the decreases in gate counts by the removal of the RAMs exceeded the increases in gate counts by the addition of shift registers, so that the total gate counts for the code after the proposed scalar replacement method were significantly less than those for the original code without scalar replacement in case of *ex1*, *ex1_x5*, *ex1_x10*, *filter* and *loop4* benchmark code. For *loop4*, the proposed method achieves 35.4% gate counts reduction compared to the previous method [7].

Since the maximum reuse distance, or equivalently, the number of shift register bits of *ex_d60* doubled compared to those of *ex*, the total gate count of *ex_d60* in *proposed* was significantly larger than that of *ex*. So, the scalar replacement for array accesses with large reuse distances results in large numbers of shift registers and significant area increase, which also holds for the previous work [7].

Compared to that of *ex1*, the total gate counts of *ex1_x5* and *ex_x10* in *proposed* increased by 179.6 and 355.7 gates, respectively. The increases were mainly due to the increases in the numbers of inputs of the multiplexers (5 inputs and 10 inputs, respectively) caused by the increases in the numbers of reuse conditions. Although we consider that the numbers of reuse conditions are not large in typical image processing applications, code with significantly large numbers of reuse conditions will result in the synthesized hardware with large multiplexers, which will negatively impact the gate counts and the maximum clock frequency of the hardware.

6. Conclusion

For successful application of high-level synthesis, manual code tunings that optimize memory accesses are often necessary. In this paper, we focused on one of the most effective memory access optimizations called scalar replacement, and proposed the improved method. Differently from the most advanced scalar replacement method for nested loops [7], our scalar replacement method handles array accesses with constant subscripts which often appear in image filtering code. Our method focuses on input C code in the form of a Static Control Part (SCoP) and builds elaborate reuse information tables with the polyhedral model. Our method replaces each reuse destination that has multiple reuse vectors with the corresponding scalar variables. These scalar variables are referenced conditionally according to the corre-

sponding conditions in the reuse information tables. We implemented the proposed method to carry out experiments to see the impacts of our method. From the experimental results, we conclude that the proposed method is a promising approach for reducing area and improving performance of hardware generated by HLS from array-intensive C code which contain array accesses with constant subscripts.

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] Carr, S. and Kennedy, K.: Scalar Replacement in the Presence of Conditional Control Flow, *Software-Practice and Experience*, Vol.24, No.1, pp.51–77 (1994).
[7] So, B. and Hall, M.W.: Increasing the Applicability of Scalar Replacement, *Compiler Construction* (*CC*) (2004).
[8] Budiu, M. and Goldstein, S.C.: Inter-iteration Scalar Replacement in the Presence of Conditional Control Flow, *Workshop on Optimizations for DSP and Embedded Systems* (*ODES*) (2005).
[9] Surendran, R., Barik, R., Zhao, J. and Sarkar, V.: Inter-iteration Scalar Replacement Using Array SSA Form, *International Conference on Compiler Construction* (2014).
[10] Zhou, Y., Al-Hawaj, K. and Zhang, Z.: A New Approach to Automatic Memory Banking using Trace-Based Address Mining, *FPGA'17* (2017).
[11] Rau, B.: Iterative Modulo Scheduling, HP Labs Technical Report HPL-94-115 (1994).
[12] Kato, Y. and Seto, K.: Loop Fusion with Outer Loop Shifting for High-level Synthesis, *IPSJ Trans. System LSI Design Methodology*, Vol.6 (2013).
[13] 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).
[14] Bondhugula, U., Hartono, A., Ramanujam, J. and Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer, *Proc. 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI*) (2008).
[15] 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 2018-03).
[16] Bastoul, C.: A Specification and a Library for Data Exchange in Polyhedral Compilation Tools, available from ⟨http://icps.u-strasbg.fr/people/bastoul/public_html/development/openscop/docs/openscop.pdf⟩ (accessed 2018-03).
[17] Bastoul, C.: CLooG: A Loop Generator For Scanning Polyhedra, available from ⟨http://www.bastoul.net/cloog/pages/download/cloog.pdf⟩ (accessed 2018-03).
[18] Verdoolaege, S.: Integer Set Library: Manual, available from ⟨http://isl.gforge.inria.fr/manual.pdf⟩ (accessed 2018-03).
[19] Verdoolaege, S.: isl: An Integer Set Library for the Polyhedral Model, *International Congress on Mathematical Software* (*ICMS*) (2010).
[20] Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A. and Bastoul, C.: The Polyhedral Model is More Widely Applicable Than You Think, *Compiler Construction* (*CC*) (2010).
[21] Grosser, T., Groesslinger, A. and Lengauer, C.: Polly - Performing polyhedral optimizations on a low-level intermediate representation, *Parallel Processing Letters* (2012).
[22] Kelly, W. and Pugh, W.: A unifying framework for iteration reordering transformations, *IEEE 1st International Conference on Algorithms and Architectures for Parallel Processing* (1995).

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: *Hiroyuki Tomiyama*)