Scalar replacement in the presence of multiple write accesses for high-level synthesis

Kenshu Seto
Department of Electrical, Electronic and Communication Engineering
Tokyo City University, Tokyo, Japan

***Abstract*—High-level synthesis (HLS) reduces design time of domain-specific accelerators from loop nests. Usually, naive usage of HLS leads to accelerators with insufficient performance, so very time-consuming manual optimizations of input programs are necessary in such cases. Scalar replacement is a promising automatic memory access optimization that removes redundant memory accesses. However, it cannot handle loops with multiple write accesses to the same array, which poses a severe limitation of its applicability. In this paper, we propose a new memory access optimization technique that breaks the limitation. Experimental results show that the proposed method achieves 2.1x performance gain on average for the benchmark programs which the state-of-the-art memory optimization techniques cannot optimize.**

***Index Terms*—high-level synthesis, memory access optimization, scalar replacement**

I. Introduction

HLS (High-level synthesis) [1] has been leveraged to design hardware rapidly from algorithmic descriptions such as C programs for image processing, deep neural networks, etc. Typically, we use the loop pipelining technique to synthesize high-performance and area-efficient accelerators from loops in C programs, where we try to reduce the initiation intervals (IIs) as much as possible for the highest performance.

Applying HLS to loops which are coded without considering hardware implementation often leads to hardware whose performance is far below satisfaction. In such cases, hardware designers have to devote significant efforts to read tool manuals carefully and perform time-wasting trial and error in order to rewrite C programs into ones that are suitable for HLS and that are expected to achieve reduced IIs. Such rewriting includes loop transformations for increasing parallelism and memory-related optimizations that optimizes memory accesses. In this paper, we focus on memory access optimizations.

In HLS, RAMs are represented by arrays. Typically, the numbers of ports for RAMs are limited up to 2. So, the application of HLS to loop programs that contain many accesses to the same array generates hardware with frequent memory access conflicts, and as a result, IIs are not reduced and the performance gains by the generated accelerators are limited. To address the access conflicts to RAMs in HLS, two types of memory access optimizations have been proposed called memory partitioning [2] [3] and scalar replacement [4] [5] [6].

Memory partitioning partitions an array (RAM) into multiple smaller arrays so that the numbers of available memory ports, and hence, memory bandwidths increase. Memory partitioning was originally proposed for array accesses whose subscripts are affine expressions [2] and was further extended to handle array accesses whose subscripts are non-affine expressions [3]. As the number of partitions increases, the number of inputs to multiplexers that select partitioned RAMs increases, which has negative impact on circuit performance and area. Furthermore, we cannot apply the memory partitioning to the array accesses whose subscripts are the same which limits the applicability of memory partitioning.

Scalar replacement flows a sequence of data accessed to or from an array element to shift registers. When the same array element is accessed, the optimized program accesses the data not from the array but from the shift registers, so that the access to the array (RAM) is removed. Scalar replacement requires shift registers which have negative impact on circuit area. Differently from memory partitioning, we can apply scalar replacement even to the array accesses whose subscripts are the same. Scalar replacement was originally proposed in [4] as an optimization for RISC processors and focused on data reuse carried within the innermost loops. Then, scalar replacement was extended to handle data reuses carried not only by innermost loops but also by outer loops for FPGAs [5] which can afford more registers than RISC processors. Scalar replacement had been further extended to handle array accesses with constant subscripts [6], to reduce the chip area for shift registers and handle loop bounds with compile time parameters [7] and to realize efficient initialization of shift registers [8].

Unfortunately, the applications of existing scalar replacement are restricted to loops with at most one write access to each array, and this restriction severely limits the effectiveness of scalar replacement. In this paper, we extend scalar replacement to break the limitation of the previous techniques and to generate high-performance accelerators with HLS. The contributions of this paper are the following:

- We propose an extended scalar replacement algorithm for loops with more than one write accesses to each array.
- We propose an algorithm to exhaustively enumerate sets of array accesses that can be simultaneously removed and an algorithm to select the best sets of array accesses to be removed in order to satisfy the constraint on the initiation interval (II) and minimize the length of shift registers.
- We implement the tool to automate the algorithms and present the evaluation results on benchmark programs in terms of performance and gate counts.

978-3-9819263-5-4/DATE21/©2021 EDAA

```
1.  for (i = 0; i < 42; i++)
2.    y[i] = 0;
3.  for (i = 0; i < 38; i++) {
4.    T[i] = 0;
5.    for (j = 0; j < 42; j++)
6.      T[i] = T[i] + A[i][j] * x[j];
7.    for (j = 0; j < 42; j++)
8.      y[j] = y[j] + A[i][j] * T[i]; }
```

(a) Original program

```
1.  for (i=0;i<=38;i++)
2.    for (j=0;j<=41;j++) {
3.      if (i >= 1)
4.        y[j]=y[j] + A[i-1][j] * T[i-1](T_0_R);
5.      if (i <= 37 && j == 0)
6.        T[i](T_1_W)=0;
7.      if (i <= 37)
8.        T[i](T_3_W)=T[i](T_2_R) + A[i][j] * x[j];
9.      if (i == 0)
10.       y[j]=0; }
```

(b) Program after loop fusion

Fig. 1. Example program (atax): (a) is an original program and (b) is a program after applying loop fusion to (a). In (b), accesses to array "T" are annotated with access names in green such as T_0_R or T_1_W where 0 and 1 are ID numbers and R and W represent read and write, respectively.

II. Problem of existing memory access optimizations

In this section, we explain the limitations of the existing memory partitioning [2] [3] and scalar replacement [5] [6] using the example program shown in Fig. 1. Fig. 1 (a) is a program (atax) in PolyBench [11] and Fig. 1 (b) is a program where nested loops are fused into a single fully nested loop with loop fusion [9]. Fusing loops possibly leads to enhancing parallelism and reduced buffer size due to improved locality [10]. So, in the following, we focus on optimizing the code after loop fusion shown in Fig. 1 (b).

In this paper, we assume arrays are mapped to dual-port memories when the arrays are accessed more than once in a loop iteration. Other memories with different numbers of ports can be handled with the proposed method. We also assume that loop pipelining is applied to the innermost loops with the target initiation intervals (IIs) of 1. When we apply loop pipelining in HLS to the program is Fig. 1 (b), the lower bound of II is 2. The bottleneck array that limits the reduction of II is "T" which is accessed 4 times for each loop iteration (In Fig. 1 (b), the four accesses to "T" are referred to as T_0_R, T_1_W, T_2_R and T_3_W). Since the number of memory ports is 2, the lower bound of II due to array "T" is given by $II_{\mathrm{T}} = ceil(4/2) = 2$ [6]. Since the target II of 1 is not satisfied, we need to apply memory access optimizations to array "T".

If we use memory partitioning [2] [3], array accesses with the same subscripts cannot be assigned to different memory banks. In Fig. 1 (b), read accesses T[i-1] in line 4 and T[i] in line 8 have different subscripts so that these two accesses can be accessed in parallel by assigning the accesses to different banks. Unfortunately, the program in Fig. 1 (b) has 3 accesses to T[i] with the same subscript, and these 3 accesses cannot be assigned to different banks. So, the lower bound of II remains to be 2 because of $II_{\mathrm{T}} = ceil(3/2) = 2$, which is the same as that without the memory partitioning.

In addition, we cannot apply existing scalar replacement techniques to array "T" in Fig. 1 (b), because the read access T[i] in line 8 (T_2_R) reuses the data accessed not only by the write access T[i] in line 6 (T_1_W) but also by the write access T[i] in line 8 (T_3_W) and existing scalar replacement techniques permit at most one write access for each array. So, we cannot achieve II of 1 for the program shown in Fig. 1 (b) with the existing scalar replacement techniques.

III. Preliminaries

In this section, we briefly summarize the preliminaries to describe the proposed method. For more details on the definitions, please refer to [6] [7]. We assume target fused loops are fully nested and written in Static Control Part (SCoP) format [12] by which polyhedral analysis is possible. Array accesses in C programs are categorized into two types: static and dynamic. A static array access is an array access that appears in a program text. A dynamic array access is an array access instance that is actually executed when running a program. Reducing static array accesses directly affects the IIs of loop pipelining, so array accesses in this paper mean static array accesses.

Definition 3.1 (Iteration vector): A vector $\vec{i}$ whose elements are values of loop indices from the outermost loop to the innermost loop in a nested loop is called an **iteration vector**. For the code in Fig. 1 (b), $(i, j) = (0, 0)$ is an iteration vector.

Definition 3.2 (Domain of array access): A set of iteration vectors, D_a, that contains all iterations in which an array access a in a loop is executed is called the **domain** of the array access a. For example, the domain $D_{\mathrm{T\_2\_R}}$ for the array access T_2_R in Fig. 1 is $D_{\mathrm{T\_2\_R}} = \{(i, j) \mid 0 \leq i \leq 37 \wedge 0 \leq j \leq 41\}$.

Definition 3.3 (Reuse relation between array accesses): The **reuse relation** $R_{s,d}$ from an array access s to an array access 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 $\vec{i_s} \in D_s$. For example in Fig. 1 (b), the reuse relation $R_{\mathrm{T\_3\_W,T\_2\_R}}$ from T_3_W to T_2_R is $R_{\mathrm{T\_3\_W,T\_2\_R}} = \{(i, j) \rightarrow (i', j') \mid 0 \leq i \leq 37 \wedge 0 \leq j \leq 41 \wedge i' = i \wedge j' = j + 1\}$

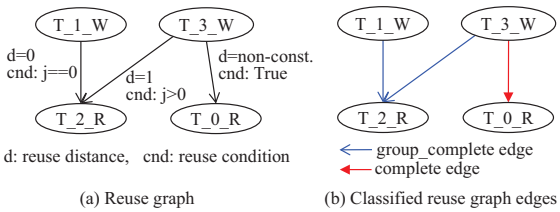
IV. Scalar replacement for multiple write accesses

In this section, we present the proposed scalar replacement extended to handle multiple write accesses to the same array.

A. Overall flow

The overall flow of the proposed scalar replacement method consists of following steps: (Step. 1) Parse an input C program, (Step. 2) Build reuse graphs, (Step. 3) Exaustively enumerate removable subsets of array accesses, (Step. 4) Select the best subset of array accesses to remove, and (Step. 5) Transform the program in order to remove array accesses.

Step. 1 parses an input C program in SCoP format and construct the polyhedral model [12]. Step. 2 builds reuse graphs which are the key data structure of the proposed method based on the polyhedral model. Using the reuse graphs, Step. 3 enumerates removable subsets of array accesses for each array. Step 4. selects the best subset of array accesses to remove in terms of the shift register length, satisfying the target II constraint. Based on the information on which array accesses should be removed from Step. 4, actual program transformation is performed to remove the selected array accesses in Step 5. We used ISL (Integer Set Library) [13], a library to handle sets and mappings of integer lattice points and automated all the steps from steps 1 to 5. In the following, we describe the details of steps 2 to 4.



(a) Reuse graph (b) Classified reuse graph edges

Fig. 2. Reuse graph and classified edges for the program in Fig. 1(b)

Algorithm *GenerateReuseGraphEdges*

Input Arr: set of arrays in target loop
Input $accesses(A)$: Accesses for each array $A \in Arr$
Input $writes(A)$: Write accesses for each array $A \in Arr$
Input $R_{s,d}$: Reuse relations for each pair (s, d) of accesses
Input D_d: Domain of each access d
Output $edges(A)$: Edges of reuse graph for array $A \in Arr$

```
1: foreach array A ∈ Arr :
2:   edges(A) = None
3:   foreach s in accesses(A)
4:     foreach d in accesses(A)
5:       if s == d : continue
6:       if d is not Read : continue
7:       intervening_writes = None
8:       foreach w in writes(A)
9:         if w == d : continue
10:        intervening_writes += w
11:      foreach iw in intervening_writes :
12:        R_{s,d} = R_{s,d} − R_{s,iw} ∘ R_{iw,d}
13:      if R_{s,d} ≠ ∅ : edges(A) += edge (s, d)
14:      ReuseDist_{s,d} = ReuseDistance(R_{s,d})
15:      ReuseCond_{s,d} = gist(Range(R_{s,d}), D_d)
```

Fig. 3. Algorithm for generating reuse graph edges

B. Building reuse graphs

Figure 2 (a) shows the reuse graph build with the algorithm in Fig. 3. 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 value accessed by the access s will be accessed later by the access d. We call s a reuse source and d a reuse destination. We build a reuse graph for each array.

Each node of a reuse graph is simply generated from each access of an array. Figure 3 shows an algorithm to generate edges of a reuse graph. In line 5 of Fig. 3, we skip the case of self-reuse where the reuse source s and destination d are the same, since we cannot exploit self-reuse to remove static array access. This is because array accesses regarding to shift register initialization remain in the loop body when removing self-reuse. The reuse by a reuse source s from a reuse destination d occurs only when the dependence from s to d is Read-After-Read (RAR) or Read-After-Write (RAW), so we impose the constraint on reuse graph edges as in line 6. Differently from the previous work where the number of write accesses for each array is limited up to one, when multiple write accesses exist in a loop body for an array, one of the write accesses may overwrite the value accessed by other write accesses. We subtract reuse relations that are invalidated by such intervening write accesses from lines 7 to 12 in Fig. 3 where the symbol

Algorithm *ClassifyReuseGraphEdges*

Input Arr: set of arrays in target loop
Input $edges(A)$: Reuse graph edges for each array $A \in Arr$
Input $R_{s,d}$: Reuse relations for each pair (s, d) of accesses
Input D_d: Domain of each access d
Output $type(s, d)$: Types of each edge $e = (s, d)$

```
1: foreach array A ∈ Arr :
2:   foreach destination node d of edges(A) :
2:     edges(d) = all edges to the destination node d
3:     foreach edge (s, d) ∈ edges(d) :
4:       if Range(R_{s,d}) == D_d : type(s, d) = complete
5:       else :                     type(s, d) = partial
6:     Range_d = None
7:     foreach edge (s, d) ∈ edges(d) :
8:       Range_d = Range_d + Range(R_{s,d})
9:     if Range_d == D_d :
10:      foreach edge (s, d) ∈ edges(d) :
11:        if type(s, d) ≠ complete :
12:          type(s, d) = group_complete
```

Fig. 4. Algorithm for classifying edges of reuse graphs

"∘" in line 12 represents the composition of the reuse relations $R_{s,iw}$ and $R_{iw,d}$. We generate an edge from s to d in a reuse graph only when the reuse relation $R_{s,d}$ without the intervened write accesses is not empty in line 13. In line 14, the reuse distance is computed for each edge with the similar method as [5]. In line 15, the reuse condition is computed for each edge with the gist operation [13] that simplifies the condition. The reuse condition represents the condition under which the reuse occurs in terms of the iteration vectors at reuse destination. The algorithm in Fig. 3 generates the reuse graph shown in Fig. 2 (a) for the program in Fig. 1(b).

In order to handle multiple write accesses for an array, we classify edges in the reuse graph into 3 types: *complete*, *group_complete* and *partial* using the algorithm shown in Fig. 4. An edge $e = (s, d)$ is *complete* when only one source node s accesses in advance all the values accessed by the destination node d. An edge $e = (s, d)$ is *group_complete* when the destination node d reuse data from more than one source node $s_1, s_2, ...$ When an edge e is neither *complete* nor *group_complete*, the edge e is *partial*. The proposed method in this paper executes scalar replacement not for *partial* edges but for *complete* and *group_complete* edges. Fig. 2 (b) shows the result of the edge classification. In Fig. 2 (b), for example, the edge to "T_0_R" is *complete* because the source node "T_3_W" accesses all the data that are accessed later by "T_0_R". On the other hand, the edges to "T_2_R" is *group_complete*, since a part of the data accessed by "T_2_R" is only accessed by "T_1_W" and the other part of data is only accessed "T_3_W".

C. Enumerating removable combinations of array accesses

After we build reuse graphs, we exaustively enumerate sets of removable array accesses for each array using the algorithm in Fig. 5. Applying the algorithm in Fig. 5 to the reuse graph in Fig. 2 generates the result shown in Fig. 6.

The algorithm proceeds in two phases: phase 1 which enumerates removable read accesses, followed by phase 2 which enumerates removable write accesses enabled by phase 1. More specifically, we enumerate sets of removable read accesses and

Algorithm *EnumerateSetsOfRemovableAccesses*

Input Arr: set of arrays in target loop
Input $reads(A)$: read accesses for each array $A \in Arr$
Input $writes(A)$: write accesses for each array $A \in Arr$
Input $edges(A)$: Edges of reuse graph for array $A \in Arr$
Output $removeSets(A)$: sets of removable accesses for each array $A \in Arr$

```
1: foreach array A ∈ Arr :
2:   RemoveReadSets = None
3:   readSets = combinations(reads(A))
4:   foreach readSet ∈ readSets :
5:     canRemove = True
6:     foreach read access r ∈ readSet :
7:       inedges_r = input_edges(r, edges(A))
8:       foreach input edge ie ∈ inedges_r :
9:         if ReuseDistance_ie is not const.: canRemove = False
10:      if size(inedges_r) == 0: canRemove = False
11:      if all_partial(inedges_r) == True: canRemove = False
12:      if canRemove == False: break
13:    if canRemove == True: removeReadSets += readSet
14:  foreach removeReadSet ∈ removeReadSets :
15:    removeWriteSet = None
16:    foreach write access w ∈ writes(A)
17:      if array A is output array : break
18:      canRemove = True
19:      foreach edge e ∈ output_edges(w) :
20:        if destination(e) ∉ removeReadSet :
21:          canRemove = False
22:      if canRemove == True : removeWriteSet += w
23:    removeSet = removeReadSet ∪ removeWriteSet
24:    removeSets(A) += removeSet
```

Fig. 5. Algorithm for enumerating sets of removable accesses

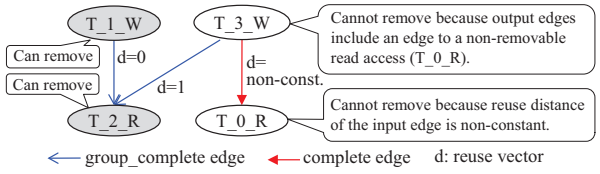


Fig. 6. A set of removable accesses (filled in gray) for Fig. 2 by Fig. 5

store them in $removeReadSets$ from lines 4 to 13 in Fig. 5, then we compute a set $removeWriteSet$ of removable write accesses from lines 14 to 24 for each set $removeReadSet$ of removable read accesses. To check if a read access r in a reuse graph can be removed in the algorithm shown in Fig. 5, the algorithm checks the following 3 conditions: (1) the reuse vectors of the input edges of r should be all constant vectors, (2) the number of the input edges of r is not zero and (3) at least one input edge of r is not partial.

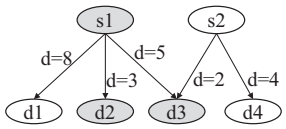
Given a set of removable read accesses, we compute a set of removable write accesses based on the information on the set of removable read accesses. Removing the write accesses, if possible, is always beneficial, since it does not incur any overhead. As shown in line 17 in Fig. 5, we do not remove write accesses to output arrays. In addition, a write access w cannot be removed when any output edge from w in the reuse graph is connected to a read access that is not removed (lines 19-21), since removing such a write access makes the read access incorrect. The set of removable read accesses and the set of removable write accesses are unioned in line 23 to make a set of removable accesses. Finally, $removeSets(A)$ in line 24 contains sets of removable accesses for each array A.

Algorithm *SelectBestRemoveSet*

Input Arr: set of arrays in target loop
Input $edges(A)$: Reuse graph edges for each array $A \in Arr$
Input $accesses(A)$: Accesses for each array $A \in Arr$
Input $removeSets(A)$: sets of removable accesses
Input $targetII$: target initiation interval (II)
Output $bestRemoveSet(A)$: the best set of removable accesses for each array $A \in Arr$

```
1: foreach arrayA ∈ Arr :
2:   bestRemoveSet(A) = None
3:   minArea = ∞
4:   foreach removeSet ∈ removeSets(A) :
5:     shiftRegLen = 0
6:     foreach source node s of edges(A) :
7:       srcShiftRegLen = 0
8:       foreach edge (s, d) ∈ edges(A) from source node s :
9:         if type(s, d) == partial : continue
10:        if d ∈ removeSet :
11:          if ReuseDist_s,d > srcShiftRegLen :
12:            srcShiftRegLen = ReuseDist_s,d
13:      shiftRegLen += srcShiftRegLen
14:    unremoveSet = accesses(A) − removeSet
15:    II = ceil(size(unremoveSet)/2)
16:    if II ≤ targetII :
17:      if shiftRegLen < minShiftRegLen :
18:        minShiftRegLen = shiftRegLen
19:        bestRemoveSet(A) = removeSet
20:  if bestRemoveSet(A) == None : exit
```

Fig. 7. Algorithm for selecting the set of accesses that tries to achieve the target II and minimize the shift register length



Shift register length for s1: max{3, 5} = 5
Shift register length for s2: max{2} = 2
Total shift register length for {s1, d2, d3}: 5+2=7

II= ceil(size(unremoveSet)/2) = ceil(3/2) = 2

Fig. 8. Computing shift register length and lower bound of II for a set of removable access by the algorithm in Fig. 7 for a synthetic example

In Fig. 6, there are two read accesses: "T_0_R" and "T_2_R". Since the input edge of "T_0_R" has a non-constant reuse distance, "T_0_R" cannot be removed. As a result, the sets of removable read accesses for Fig. 2 consist of just one set { "T_2_R" } with one element "T_2_R". In Fig. 6, there are two write accesses: "T_1_W" and "T_3_W". The output edge of "T_1_W" is connected only to the read access "T_2_R" that is removed, so "T_1_W" is added to the removable write set. On the other hand, one of the output edges of "T_3_W" is connected to "T_0_R" which remains unremoved, so "T_3_W" is not added to the removable write set. As a result, the sets of removable accesses for array "T" consist of one set { "T_2_R", "T_1_W"} as filled in gray in Fig. 6.

D. Selecting the combination of array accesses to remove

After exhaustively enumerating sets of removable accesses with the algorithm shown in Fig. 5, we select the best set of removable accesses under the target initiation interval (II) constraint with the algorithm shown in Fig. 7. Removing array accesses with scalar replacement incurs shift registers and the

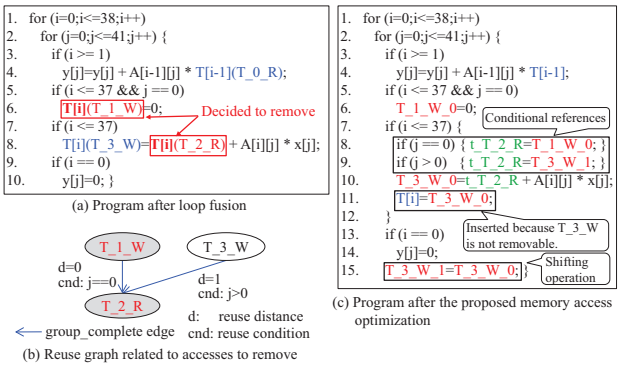


Fig. 9. Program transformation to remove the selected set of accesses

length of the shift registers generated by scalar replacement depends on which array accesses are removed. So, the best combination of array accesses means it minimizes the length of the shift registers while trying to satisfy the target II. In the algorithm in Fig. 7, we only consider resource-constrained II. After removing the selected set of array accesses, the target II may not be achieved due to loop-carried dependencies. Our work can be directly applied to many useful applications that do not contain loop carried dependencies. If a target application has loop-carried dependencies, we can still run the proposed method. In such a case, we can obtain the minimum II under the loop-carried dependences after HLS run. Then, we can use the resulting minimum II under the loop-carried dependences after HLS as the targetII in Fig. 7 in order to reduce the shift registers. In Fig. 7, we compute the length of shift registers (lines 6 to 13) and II (lines 14 to 15) for each set of removable accesses. If the target II is not achievable for an array, the algorithm ends without a valid solution (line 20). For the example in Fig. 6, the sets of removable accesses for array "T" happen to be just one set { "T_2_R", "T_1_W"}. Removing the two accesses in this set requires the shift register with the length of 1 and the lower bound of II for "T" is $ceil(2/2) = 1$, which satisfies the target II of 1. So the set { "T_2_R", "T_1_W"} is selected by the algorithm in Fig. 7. To illustrate the algorithm in Fig. 7, we show another synthetic example in Fig. 8.

E. Program transformation

After the set of accesses to remove is determined for each array by the algorithm in Fig. 7, actual program transformation generates the final optimized program as shown in Fig. 9(c). For the example in Fig. 9(a)(b), the accesses "T_2_R" and "T_1_W" are removed.

When parsing the input C code, we build a simplified abstract syntax tree (AST). With the AST, we can identify the statement that a given array access belongs to. With the AST, we can replace an array accesses with a scalar variable with any names, insert statements before and after any statement.

In the reuse graph related to removing accesses, each source node is replaced with a scalar variable. In this work, we name the scalar variable "access name"+"_0". So, the array accesses "T_1_W" and "T_3_W" in Fig. 9(a) are replaced with the scalar variable "T_1_W_0" and "T_3_W_0", respectively. Since we do not remove the array access "T_3_W", we insert the statement that "T[i] = T_3_W_0" at line 11 in Fig. 9(c) to recover the array access "T_3_W" just below the statement in which "T_3_W" originally belonged to.

In the reuse graph related to removing accesses, each destination node is replaced with a scalar variable. The scalar variable is named "access name of source node"+"reuse distance". The destination node "T_2_R" has group complete edges from "T_1_W" and "T_3_W" with reuse distances of 0 and 1 and reuse conditions of $j == 0$ and $j > 0$, respectively, as shown in Fig. 9(b). So, "T_2_R" should be replaced with different scalar variables "T_1_W_0" and "T_3_W_1" depending on the conditions $j == 0$ and $j > 0$, respectively. To implement such conditional references to different scalar variables, we use the temporary variable "t_T_2_R" and the assignments to the variable are inserted before the statement that "T_2_R" originally belonged to as shown at lines 8-10 in Fig. 9(c). Shifting operations of shift registers that correspond to register move operations are inserted at the bottom of the innermost loop body at lines 15 in Fig. 9(c).

V. Experimental results

In this section, we show the impacts of the proposed memory access optimization on performance and gate counts of synthesized accelerators with HLS.

A. Experimental setups

We implemented an automatic tool that executes the proposed memory access optimization explained in Section IV. In the implementation, we utilized ISL (Integer Set Library) [13]. We set the targetII in Fig. 7 to be 1. We mapped arrays that are accessed more than once in one iteration of a loop body to dual-port RAMs, and mapped arrays that are accessed only once in one iteration of a loop body to single-port RAMs.

In Table I, we show 4 benchmark programs which are computations on matrices and vectors. *atax*, *2mm*, *3mm* are programs from PolyBench [11]. *dct* is a program for discrete cosine transform. We applied the proposed memory access optimization to the 4 benchmark programs with our tool, followed by the executions of a commercial high-level synthesis (HLS) tool and logic synthesis tool with a $45nm$ technology library and clock period constraint of $2ns$.

B. Results and discussions

Table I shows the experimental results. For all the benchmark programs, our tool generated the optimized programs within a few seconds. In the code type, *original* means the programs without loop fusion and the proposed optimization, *fused* means the programs with loop fusion but without the proposed optimization, and *proposed* means the programs with loop fusion and the proposed optimization. For each code type, we applied loop pipelining to the innermost loops. We verified the equivalence between *original*, *fused* and *proposed* by simulation. We could not apply the previous scalar replacement algorithms [5] [6] [7] [8] to the *fused* programs, since all the programs have more than one write accesses to the target arrays. In addition, memory partitioning algorithms [2] [3] could not achieve the

TABLE I
IMPACT OF THE PROPOSED MEMORY OPTIMIZATION TECHNIQUE ON PERFORMANCE AND GATE COUNTS

Benchmark programs	Code type	II	# of execution cycles	Speedup	Min. clock period [ns]	Gate counts	# of total RAM bits	Total gate counts
atax	*original*	-	3,565	1.00	1.61	6,742 (1.00)	54,976	169,654 (1.00)
	fused	3	5,070	0.72	1.81	6,765 (1.00)	54,976	169,677 (1.00)
	proposed	1	1,794	2.04	1.73	12,477 (1.85)	54,976	175,389 (1.03)
2mm	*original*	-	17,600	1.00	1.60	6,625 (1.00)	59,264	127,777 (1.00)
	fused	3	23,424	0.75	1.81	6,954 (1.05)	59,264	128,106 (1.00)
	proposed	1	9,136	1.93	1.84	22,051 (3.33)	60,800	145,507 (1.14)
3mm	*original*	-	28,902	1.00	1.71	7,245 (1.00)	85,632	185,421 (1.00)
	fused	3	32,205	0.90	1.83	8,538 (1.18)	85,632	186,714 (1.01)
	proposed	1	12,559	2.30	1.79	26,202 (3.07)	87,168	206,682 (1.11)
dct	*original*	-	1,744	1.00	1.60	6,007 (1.00)	8,192	30,583 (1.00)
	fused	3	2,169	0.80	1.71	6,620 (1.10)	8,192	31,196 (1.02)
	proposed	1	873	2.00	1.67	11,260 (1.88)	8,192	35,836 (1.17)

target II of 1 for the benchmark programs as explained in Section II, since all the programs have target arrays with more than 2 accesses with the same subscripts.

In Table I, II represents the minimum initiation intervals (IIs) that were achieved when the innermost loops were loop pipelined. The proposed optimization could achieve the target II of 1 for all benchmark programs, because it could successfully remove array accesses. On the other hand, the minimum IIs of *fused* were 3, because of unremoved array accesses. For *original*, each non-fused innermost loop achieved the minimum II of 1. In Table I, Speedup is computed as the ratio of the number of execution cycles for *original* to the number of execution cycles for *fused* and *proposed* and the average speedup was 0.79 and 2.1, respectively. *fused* degraded the performance compared to *original* because array accesses got together in fused loop bodies. On the other hand, *proposed* achieved speedup compared to *original* since the concentrated array accesses were removed. *fused* and *proposed* satisfied the target clock period of $2ns$, and the minimum clock periods of *original* were slightly smaller than those of *fused* and *proposed*. From the results, we see that the proposed method achieves significant speedups for the benchmark programs.

In Table I, Gate counts represent the gate counts after logic synthesis in terms of NAND2. Gate counts exclude the gate counts for RAMs. The numbers in parenthesis show the ratios of *fused* and *proposed* to *original*. The average ratios of gate counts of *fused* and *proposed* to those of *original* was 1.1 and 2.7, respectively. The gate counts for *original* and *fused* were almost the same. On the other hand, the parallel execution achieved in *proposed* required increased numbers of functional units compared to *original*, which lead to the increased gate counts. # of total RAM bits show the total number of RAM bits for input, output and temporal arrays. Total gate counts (the rightmost column) show the gate counts including all RAMs. Since the gate counts for RAMs dominated the total gate counts, the increases in the total gate counts for *proposed* were moderate compared to those for *original*.

In summary, the proposed memory optimization method provided significant speedup while the increases of gate counts by the proposed method were reasonable. Hence, the proposed memory optimization technique is promising for designing high-performance accelerators with HLS from loops with multiple writes to the same array.

VI. CONCLUSION

Designing domain-specific accelerators with HLS from loop programs often involves memory access optimizations such as scalar replacement. Unfortunately, existing scalar replacement techniques have a restriction that only one write access is permitted to each array. This restriction poses a severe limitation to the applicability of the scalar replacement and significantly limits the performance gains of the synthesized accelerators. This paper generalized the scalar replacement and proposed a technique that successfully handles loops with multiple write accesses to the same array. Experimental results show that the proposed optimization effectively boosts the performance of synthesized accelerators with moderate increases in gate counts.

REFERENCES

[1] Razvan Nane, et al., "A Survey and Evaluation of FPGA High-Level Synthesis Tools," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, 2016

[2] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou, "Theory and algorithm for generalized memory partitioning in high-level synthesis, " ACM/SIGDA International Symposium on FPGAs, 2014

[3] Yuan Zhou, Khalid Al-Hawaj, and Zhiru Zhang, "A New Approach to Automatic Memory Banking using Trace-Based Address Mining, " ACM/SIGDA International Symposium on FPGAs, 2017

[4] David Callahan, Steve Carr and Ken Kennedy, "Improving Register Allocation for Subscripted Variables, " ACM/SIGPLAN Conference on Programming Language Design and Implementation, 1990

[5] Byoungro So and Mary W. Hall, "Increasing the Applicability of Scalar Replacement", Compiler Construction (CC), 2004

[6] Kenshu Seto, "Scalar Replacement with Polyhedral Model," IPSJ Transactions on System LSI Design Methodology, vol. 11, 2018

[7] Kenshu Seto, "Scalar Replacement with Circular Buffers," IPSJ Transactions on System LSI Design Methodology, vol. 12, 2019

[8] Kenshu Seto, "Shift Register Initialization in Scalar Replacement for Reducing Code Size," IPSJ Transactions on System LSI Design Methodology, vol. 13, 2020

[9] Yuta Kato and Kenshu Seto, "Loop Fusion with Outer Loop Shifting for High-level Synthesis," IPSJ Transactions on System LSI Design Methodology, vol. 6, 2013

[10] Antoine Fraboulet, Karen Kodary and Anne Mignotte, "Loop Fusion for Memory Space Optimization," International Symposium on Systems Synthesis (ISSS), 2001

[11] Tomofumi Yuki, Louis-Noël Pouchet, "Polybench", 2016, http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[12] Cédric Bastoul, "A Polyhedral Representation Extractor for High Level Programs", http://icps.u-strasbg.fr/~bastoul/development/clan/docs/clan.pdf

[13] Sven Verdoolaege, "Integer Set Library", 2020, http://isl.gforge.inria.fr