IPSJ Transactions on System LSI Design Methodology Vol.6 71–75 (Feb. 2013)

[DOI: 10.2197/ipsjtsldm.6.71]

Short Paper

Loop Fusion with Outer Loop Shifting for High-level Synthesis

Yuta Kato[1,†1] Kenshu Seto[1,a)]

Received: May 28, 2012, Revised: August 28, 2012,
Accepted: October 30, 2012, Released: February 15, 2013

Abstract: Loop fusion is often necessary before successful application of high-level synthesis (HLS). Although promising loop optimization tools based on the polyhedral model such as Pluto have been proposed, they sometimes cannot fuse loops into fully nested loops. This paper proposes an effective loop transformation called Outer Loop Shifting (OLS) that facilitates successful loop fusion. With HLS, we found that the OLS generates hardware with 25% less execution cycles on average than that only by Pluto for four benchmark programs.

Keywords: loop fusion, polyhedral model, high-level synthesis

1. Introduction

High-level synthesis (HLS) [1] offers significant benefits on improving design productivity of VLSIs. Current HLS tools, however, cannot generate efficient hardware from a sequence of nested loops, because they lack the ability to schedule operations from different loops in parallel. Loop fusion is one of the effective loop optimizations that address the problem. By fusing loops, operations from different loops can be scheduled in parallel. Classical loop fusion algorithms such as Refs. [2], [3] typically study loop fusion independently of other loop transformations. Since loop fusion often requires other loop transformations such as loop shifting and loop permutation, the applicability of these algorithms is limited.

The polyhedral model is a flexible and powerful representation of loops, in which a sequence of various loop transformations can be represented by matrices, so loop optimizations based on the polyhedral model are promising approach not only in compilers for parallel processors [4], [5], [6], [7] but also in optimizers for hardware synthesis [8], [9]. Schedule is a key concept in the polyhedral model and it is classified into two kinds; one is one-dimensional schedule and the other is multi-dimensional schedule. Multi-dimensional schedule [5] is more general and can represent a wider range of input code than one-dimensional one [4]. The pioneering work Refs. [4], [5] proposed methods to generate minimum latency schedules and minimum delay schedules, however, they do not take locality optimization into consideration, so large buffer memories are required. Subsequent work [6] successfully finds outer parallelism, however, it does not optimize locality when finding the parallelism, so large buffer memories may be required. In addition, the algorithm processes each loop one by one when it cannot find outer parallel loops, so maximally fused loops may not be obtained by it. MMAlpha [8] constructs the polyhedral model from the input declarative code called ALPHA and directly generates VHDL for hardware accelerators. Currently multi-dimensional schedules are not supported in MMAlpha, so the range of input code is limited. Recent work [9] proposes a loop optimization method for hardware accelerators based on branch-and-bound approach that minimizes on-chip memory size (or buffer memory size) under the constraints on the number of off-chip memory accesses. Our work is more closely related to Ref. [9] than Ref. [8] in that both our work and Ref. [9] propose loop optimizers that are effective before HLS, while Ref. [8] proposes a HLS tool itself from the polyhedral model. The work Ref. [9] does not include loop shifting in the formulation and does not consider the performance optimization by loop fusion, however, it can be combined by our work.

Pluto [7] is yet another loop optimization tool based on the polyhedral model. Although our basic idea can be combined with other loop optimization frameworks such as Ref. [9], our work is based on Pluto [7], because of the following three advantages of Pluto: (1) simultaneous optimization of outer parallelism (which leads to speedups by coarse-grained parallelism) and locality (which leads to small buffer memories), (2) support for multi-dimensional schedules and (3) support for a loop transformation called loop shifting which is often necessary for loop fusion. As far as we know, none of the related work has these advantages. Although Pluto successfully fuses loops, it fails to fuse loops in some important cases. In this paper, we analyze the problem and propose a post-processing after Pluto called Outer Loop Shifting. The idea of OLS itself can be incorporated in other optimizers based on the polyhedral model. We show that the proposed procedure successfully fuses loops for which Pluto fails, and provides significant performance improvements of the synthesized hardware.

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

2. Loop Fusion Algorithm in Pluto and Its Problem

2.1 Polyhedral Model and Loop Fusion Algorithm in Pluto

We briefly review the polyhedral model and the loop fusion algorithm in Pluto [7]. Polyhedral model consists of iteration domains, multi-dimensional schedules and dependence polyhedra. The iteration vector of a statement S is defined by $\overrightarrow{i_S} = (i_1, ..., i_{m_S})$ where $i_1, ..., i_{m_S}$ is the sequence of loop index variables of the loops surrounding S from the outermost loop (i_1) to the innermost loop (i_{m_S}). The iteration domain $\mathcal{D}_S$ of a statement S is the set of vectors of which $\overrightarrow{i_S}$ can take the values. A one-dimensional schedule of a statement S is:

$$\phi_S(\overrightarrow{i_S}) = c_{S,1} \cdot i_1 + c_{S,2} \cdot i_2 + \ ... \ + c_{S,m_S} \cdot i_{m_S} + c_{S,0} \qquad (1)$$

where $c_{S,1}, ..., c_{S,m_S}, c_{S,0} \in \mathbb{Z}$ and $\overrightarrow{i_S} \in \mathbb{Z}^{m_S}$. A multi-dimensional schedule of a statement S is a vector of the one-dimensional schedules given by:

$$\mathcal{T}_S(\overrightarrow{i_S}) = (\phi_S^1(\overrightarrow{i_S}), \phi_S^2(\overrightarrow{i_S}), ..., \phi_S^d(\overrightarrow{i_S}))^T \qquad (2)$$

A multi-dimensional schedule of a statement S specifies the execution order of S in lexicographic order. If $\phi_S^k(\overrightarrow{i_S})$ in Eq. (2) is $(c_{S,1}^k, c_{S,2}^k, ..., c_{S,m_S}^k) = \overrightarrow{0}$ for all statements S, the k-th dimension (or the k-th level) of $\mathcal{T}_S(\overrightarrow{i_S})$ is called scalar dimension, otherwise, it is called loop dimension. The statements S with the same value for $c_{S,0}$ are fused at the loop dimension below the scalar dimension, and unfused statements are placed in the increase order of the coefficients $c_{S,0}$. Data dependences between statements are represented by a data dependence graph (DDG) $G = (V, E)$. Each vertex $v \in V$ represents a statement and each edge $e^{S_i \to S_j} \in E$ represents a data dependence that means statement S_j must be executed after statement S_i. Each edge $e^{S_i \to S_j} \in E$ has the dependence polyhedron $\mathcal{P}_{e^{S_i \to S_j}} = \{(\overrightarrow{i_{S_i}}, \overrightarrow{i_{S_j}}) \mid \overrightarrow{i_{S_i}} \in \mathcal{D}_{S_i}, \overrightarrow{i_{S_j}} \in \mathcal{D}_{S_j}, S_j$ at $\overrightarrow{i_{S_j}}$ is dependent on S_i at $\overrightarrow{i_{S_i}}\}$.

Finding the best multi-dimensional schedule $c_{S,0}^k$, $c_{S,1}^k$, ..., c_{S,m_S}^k, $(k = 1, ..., d)$ for each statement S while observing the data dependence constraints is the key step in Pluto. Pluto heuristically does this and optimizes both locality and outer parallelism. The dependence vector $\overrightarrow{\delta}_{e^{S_i \to S_j}}(\overrightarrow{i_{S_i}}, \overrightarrow{i_{S_j}}) \in \mathbb{Z}^d$ is defined by:

$$\overrightarrow{\delta}_{e^{S_i \to S_j}}(\overrightarrow{i_{S_i}}, \overrightarrow{i_{S_j}}) = (\delta^1_{e^{S_i \to S_j}}(\overrightarrow{i_{S_i}}, \overrightarrow{i_{S_j}}), ..., \delta^d_{e^{S_i \to S_j}}(\overrightarrow{i_{S_i}}, \overrightarrow{i_{S_j}})) \qquad (3)$$

where $\delta^k_{e^{S_i \to S_j}}(\overrightarrow{i_{S_i}}, \overrightarrow{i_{S_j}})$ is the k-th element of the dependence vector defined by:

$$\phi^k_{S_j}(\overrightarrow{i_{S_j}}) - \phi^k_{S_i}(\overrightarrow{i_{S_i}}) \quad \forall(\overrightarrow{i_{S_i}}, \overrightarrow{i_{S_j}}) \in \mathcal{P}_{e^{S_i \to S_j}}, \ \forall e^{S_i \to S_j} \in E \qquad (4)$$

Pluto lexicographically minimizes $\overrightarrow{\delta}_{e^{S_i \to S_j}}$ in order to place S_i and S_j closer in the common iteration space. Thus, the locality is improved and loops are fused. Code generation tool [10] is used to generate C code from the polyhedral model.

2.2 Problem of Loop Fusion Algorithm in Pluto

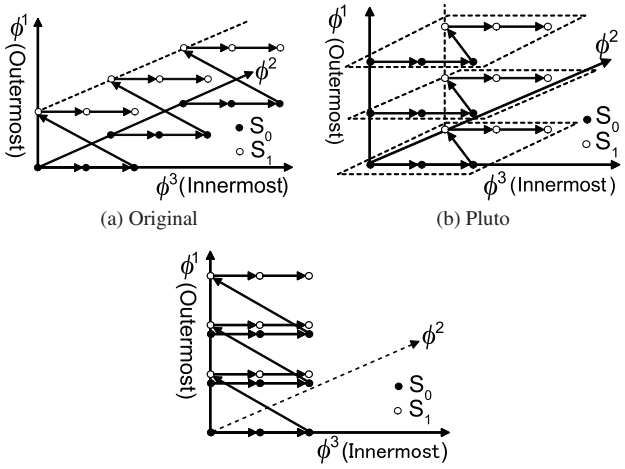
We describe the problem of Pluto with an example shown in **Fig. 1** where the original code and the original schedules $\mathcal{T}_{S_0}(\overrightarrow{i_{S_0}})$ and $\mathcal{T}_{S_1}(\overrightarrow{i_{S_1}})$ of the statements S_0 and S_1 are shown. Since a

```
for (i=0; i < N; i++)
  for (j=0; j < N; j++)
    B[i] = B[i] + A[i][j];              // S0   T_S0(i, j) = (0, i, j)^T
for (i=0; i < N; i++)
  for (j=0; j < N; j++)
    D[i] = D[i] + C[i][j] + B[i];       // S1   T_S1(i, j) = (1, i, j)^T
```

Fig. 1 Motivational example: Original code.

```
for ( φ1=0; φ1 < =N-1; φ1++ ){
  for ( φ3=0; φ3 < =N-1; φ3++ )
    B[φ1] = B[φ1] + A[φ1][φ3];          // S0   T_S0(i, j) = (i, 0, j)^T
  for ( φ3=0; φ3 < =N-1; φ3++ )
    D[φ1] = D[φ1] + C[φ1][φ3] + B[φ1]; } // S1   T_S1(i, j) = (i, 1, j)^T
```

Fig. 2 Partially fused loop after loop fusion by Pluto.



(a) Original (b) Pluto

(c) Outer Loop Shifting

Fig. 3 Schedules and data dependences of Original, Pluto and OLS code when $N = 3$.

scalar dimension appears in the outermost dimension (0 and 1), S_0 and S_1 share no common loop as shown in the code of Fig. 1.

Figure 2 shows the schedules and the code generated by Pluto. Since the schedules corresponding to the outermost dimension are $\phi^1_{S_0}(\overrightarrow{i_{S_0}}) = i$ and $\phi^1_{S_1}(\overrightarrow{i_{S_1}}) = i$, the dimension is not a scalar dimension. Therefore, the outermost dimension becomes a fused loop as shown in the code of Fig. 2. The inner loops in Fig. 2, however, are not fused, because of the second dimension ϕ^2 being a scalar dimension: $\phi^2_{S_0}(\overrightarrow{i_{S_0}}) = 0$ and $\phi^2_{S_1}(\overrightarrow{i_{S_1}}) = 1$. Since current HLS tools cannot schedule S_0 and S_1 that belong to different loops in parallel, the performance of hardware generated from the code in Fig. 2 is suboptimal.

To explain the reason why Pluto cannot fuse the loops to a fully nested form in an intuitive manner, we illustrate the multi-dimensional schedules of the original code and the code generated by Pluto in **Fig. 3** (a) and (b) where the filed circles and open circles represent the execution of S_0 and S_1, respectively, and the arrows represent the data dependencies where dependencies that can be inferred are removed from Fig. 3 for simplicity. All schedules are three dimensional: ϕ^1, ϕ^2, ϕ^3, and the statements are executed in the lexicographic order of (ϕ^1, ϕ^2, ϕ^3). In the following, we focus on the data dependence between S_0 and S_1 by the array B.

In the case of original code (Fig. 3 (a)), the dependence distance for $e^{S_0 \to S_1} \in E$ is $\overrightarrow{\delta}_{e^{S_0 \to S_1}} = (1, 0, -2)$. Since the outermost element $\delta^1_{e^{S_0 \to S_1}} = 1$, the different iterations of the outermost level

are dependent. Therefore, the parallel execution at the outermost level requires synchronization to satisfy the dependence.

Pluto preferentially minimizes the dependence distances in outer dimensions and as a result obtains the dependence distance $\overrightarrow{\delta}_{e^{S_0 \to S_1}} = (0, 1, -2)$ in Fig. 3 (b) where the outermost element $\delta^1_{e^{S_0 \to S_1}}$ is reduced to zero which means the outermost ϕ^1 loop can be executed in parallel without synchronization by multicore processors. Besides, the dependence distance is reduced from $(1, 0, -2)$ to $(0, 1, -2)$, so that locality of the array B accesses is optimized. The innermost loops, however, are not fused, because Pluto cannot find any one-dimensional schedule at the ϕ^2 level and a scalar dimension is inserted in the level. As mentioned above, S_0 and S_1 in each unfused loops are not executed in parallel, which is addressed in Section 3.

3. Loop Fusion with Outer Loop Shifting

In this section, we propose the loop transformation called Outer Loop Shifting (OLS) to enable loop fusion. The result of the loop fusion with OLS for the example in Fig. 1 is shown in **Fig. 4**. Fusion of inner loops in Fig. 2 is prevented because all iterations of the first inner loop must finish before the second inner loop. This sequential execution is caused by the data dependence from the write access to array B at S_0 to the read access from array B at S_1. By shifting the outer loop of the second loop by one in Fig. 1, the data dependence is satisfied by the outer loop, so the inner loops can be fused as explained below.

Loop fusion with OLS is represented by the schedules shown in Fig. 4. With OLS, the schedules produced by Pluto: $\mathcal{T}_{S_0}(i, j) = (i, 0, j)^T$, $\mathcal{T}_{S_1}(i, j) = (i, 1, j)^T$ in Fig. 2 are changed to the new

```
for ( φ1=0; φ1 <=N; φ1++ )
  for ( φ3=0; φ3 <=N-1; φ3++ ){
    if( φ1 <= N-1 )
      B[φ1] = B[φ1] + A[φ1][φ3];        // S0  T_S0(i,j) = (i,   0, j)^T
    if( φ1 >= 1 )
      D[φ1-1] = D[φ1-1] + C[φ1-1][φ3]
              + B[φ1-1];                // S1  T_S1(i,j) = (i+1, 0, j)^T
  }
```

Fig. 4 A fully nested loop after loop fusion with OLS.

Algorithm OuterLoopShift

OuterLoopShift(SS, sd, $\mathcal{T}_S$)
Input SS: a set of statements, sd: starting dimension
Input $\mathcal{T}_S = (\phi^1_S, ..., \phi^d_S)^T$: multi-dimensional schedules for all statements S
1: From the starting dimension sd to the innermost dimension d, Search loop dimensions L and L' ($L < L'$) enclosing a sequence of scalar dimensions
2: foreach statement S in SS
3: $S\,calar[S] \leftarrow$ A sequence of values in scalar dimensions from $L + 1$ to $L' - 1$
4: Make statements with the same $S\,calar[S]$ value a group
5: Sort groups in the increasing lexicographic order of $S\,calar[S]$ as $G_0, G_1, ..., G_q$
6: foreach statement $S \in SS$
7: Add the group value i of $S \in G_i$ to ϕ^L_S, namely $\phi^{L\prime}_S = \phi^L_S + i$
8: Reset the values of the scalar dimensions between L and L' to 0
9: foreach group G_i
10: OuterLoopShift($G_i, L', \mathcal{T}_S$)

Fig. 5 Procedure for Outer Loop Shifting.

schedules: $\mathcal{T}_{S_0}(i, j) = (i, 0, j)^T$, $\mathcal{T}_{S_1}(i, j) = (i + 1, 0, j)^T$ where the constant 1 in $\phi^2_{S_1}$ is reset to zero and the schedule of the outer dimension $\phi^1_{S_1}$ is instead added by the constant 1. After OLS, the elements in the scalar dimension are all set to zero, so the scalar dimension is removed. As evidenced by Fig. 3 (c), loop fusion with OLS shifts the execution of S_1 by one in ϕ^1 dimension, the scalar dimension ϕ_2 is removed and the execution of S_0 and S_1 is overlapped in the iteration space of $1 \le \phi^1 \le N - 1$. Because of the overlap, we expect the reduction of the execution cycles after HLS. Since the scalar dimension ϕ^2 is removed, the dependence vector in the case of Fig. 3 (c) is given by $\overrightarrow{\delta}_{e^{S_0 \to S_1}} = (1, -2)$ which has the same value as $\overrightarrow{\delta}_{e^{S_0 \to S_1}} = (0, 1, -2)$ in the case of Fig. 2. so that the locality is not deteriorated by OLS in this case compared to the result by Pluto. Since the loop in the outer dimension ϕ_1 of the scalar dimension ϕ_2 is shifted, we call the transformation Outer Loop Shifting.

In **Fig. 5**, we present a procedure for OLS that can be implemented as a post-processing step of Pluto. Initially, we call the procedure OuterLoopShift(SS, sd, $\mathcal{T}_S$) with SS: the set of all statements and sd: the outermost dimension ($sd = 1$). By the recursive call of the procedure, a sequence of scalar dimensions between loop dimensions L and L' are removed from the outermost dimension to the innermost dimension. After the call to OuterLoopShift, we check if the post-processed schedules do not violate data dependences and apply the OLS only when data dependencies are not violated.

4. Experiments

4.1 Experimental Setups

We demonstrate the effectiveness of the proposed post-processing with four benchmarks, MMs, TCE, Gemver and DCT in which the numbers of outermost loops are 2, 4, 4 and 2, and the numbers of statements in the innermost loops are 2, 4, 4 and 5, respectively. From the second column (Orig) of **Table 1**, we see that original code of MMs has two loops each of which is a 3-dimensional loop with loop index variables i, j and k and contains statements S_0 and S_1, respectively. Similarly, TCE has four 5-dimensional loops, Gemver has three 2-dimensional loops and one 1-dimensional loop, and DCT has two 3-dimensional loops each of which contains sets of statements $\{S_0, S_1\}$ and $\{S_2, S_3, S_4\}$, respectively. TCE, Gemver are included in the example directory of Pluto tool. MMs, DCT are the code for a sequence of matrix multiplication and discrete cosine transform, respectively. Pluto could not successfully fuse loops in any of the benchmarks into fully nested loops. We implemented the proposed procedure of Fig. 5 in Pluto (ver. 0.6.0) [11] and the total time of Pluto including OLS was around one second for all benchmarks. The proposed approach could successfully fuse all loops in each benchmark into a single fully nested loop. Table 1 shows the resulting multi-dimensional schedules after Pluto and after Pluto followed by OLS.

To evaluate the impact of OLS, we synthesized original code, optimized code by Pluto, and optimized code by OLS with a commercial HLS tool. The clock constraint for HLS was set to 200 MHz and the target cell library was a 0.13 μm library. For each distinct array, we allocated a single-port local memory. As

Table 1 Multi-dimensional scheduling results after Pluto and Outer Loop Shifting.

	Orig	After Pluto	After Pluto followed by OLS
MMs	$\mathcal{T}_{S_0}$ (0, i, j, k) $\mathcal{T}_{S_1}$ (1, i, j, k)	$\mathcal{T}_{S_0}$ (j, i, 0, k) $\mathcal{T}_{S_1}$ (j, k, 1, i)	$\mathcal{T}_{S_0}$ (j, i, 0, k) $\mathcal{T}_{S_1}$ (j, k+1, 0, i)
TCE	$\mathcal{T}_{S_0}$ (0, a, q, r, s, p) $\mathcal{T}_{S_1}$ (1, a, b, r, s, q) $\mathcal{T}_{S_2}$ (2, a, b, c, s, r) $\mathcal{T}_{S_3}$ (3, a, b, c, d, s)	$\mathcal{T}_{S_0}$ (a, s, 0, r, q, 1, 0, p) $\mathcal{T}_{S_1}$ (a, s, 0, r, q, 1, 1, b) $\mathcal{T}_{S_2}$ (a, s, 1, b, c, 0, 0, r) $\mathcal{T}_{S_3}$ (a, s, 1, b, c, 1, 0, d)	$\mathcal{T}_{S_0}$ (a, s, 0, r, q, 0, 0, p) $\mathcal{T}_{S_1}$ (a, s, 0, r, q+1, 0, 0, b) $\mathcal{T}_{S_2}$ (a, s+1, 0, b, c, 0, 0, r) $\mathcal{T}_{S_3}$ (a, s+1, 0, b, c+1, 0, 0, d)
Gemver	$\mathcal{T}_{S_0}$ (0, i, j) $\mathcal{T}_{S_1}$ (1, i, j) $\mathcal{T}_{S_2}$ (2, i, 0) $\mathcal{T}_{S_3}$ (3, i, j)	$\mathcal{T}_{S_0}$ (j, 0, i, 2, 0) $\mathcal{T}_{S_1}$ (i, 0, j, 2, 1) $\mathcal{T}_{S_2}$ (i, 1, 0, 1, 0) $\mathcal{T}_{S_3}$ (j, 2, i, 0, 0)	$\mathcal{T}_{S_0}$ (j, 0, i, 2, 0) $\mathcal{T}_{S_1}$ (i, 0, j, 2, 1) $\mathcal{T}_{S_2}$ (i+1, 0, 0, 1, 0) $\mathcal{T}_{S_3}$ (j+2, 0, i, 0, 0)
DCT	$\mathcal{T}_{S_0}$ (0, i, j, k, 0) $\mathcal{T}_{S_1}$ (0, i, j, k, 1) $\mathcal{T}_{S_2}$ (1, i, j, k, 0) $\mathcal{T}_{S_3}$ (1, i, j, k, 1) $\mathcal{T}_{S_4}$ (1, i, j, k, 2)	$\mathcal{T}_{S_0}$ (i, 0, j, k, 1, 0) $\mathcal{T}_{S_1}$ (i, 0, j, k, 1, 1) $\mathcal{T}_{S_2}$ (k, 0, i, j, 1, 0) $\mathcal{T}_{S_3}$ (k, 1, i, j, 0, 0) $\mathcal{T}_{S_4}$ (k, 1, i, j, 1, 0)	$\mathcal{T}_{S_0}$ (i, 0, j, k, 1, 0) $\mathcal{T}_{S_1}$ (i, 0, j, k, 1, 1) $\mathcal{T}_{S_2}$ (k, 0, i, j, 1, 0) $\mathcal{T}_{S_3}$ (k+1, 0, i, j, 0, 0) $\mathcal{T}_{S_4}$ (k+1, 0, i, j, 1, 0)

Table 2 The achieved minimum initiation intervals (IIs) after HLS.

	MMs	TCE	Gemver	DCT
Orig	4 (100%)	8 (100%)	6 (100%)	7 (100%)
Pluto	4 (100%)	8 (100%)	7 (116%)	6 (86%)
OLS	3 (75%)	6 (75%)	5 (83%)	4 (57%)

Table 3 The number of execution cycles (top) and area (bottom) after HLS.

	MMs	TCE	Gemver	DCT
Orig	124 K	204 M	61 K	897 K
Pluto	121 K	203 M	70 K	772 K
OLS	96 K	161 M	51 K	540 K

	MMs	TCE	Gemver	DCT
Orig	5.5 K	13.8 K	11.6 K	6.8 K
Pluto	5.2 K	10.8 K	10.7 K	7.6 K
OLS	5.0 K	17.0 K	15.0 K	8.1 K

behavioral-level transformations, the commercial tool performs standard compiler optimizations such as common sub-expression elimination (CSE) and constant propagation. Loop fusion by Pluto increases the chances for CSE and enhances the advantage of CSE. Loop unrolling was disabled in the experiment, however, unrolling the fused loops may further improve the operation-level parallelism. To each innermost loop of the code, we performed loop pipelining with HLS.

4.2 Experimental Results

Each entry in **Table 2** shows the sum of the achieved minimum initiation intervals (IIs) for innermost loops in all the deepest loop nests. In the table, Orig, Pluto and OLS are the HLS results of the original code, the code after Pluto and the code after OLS. The values in the parentheses show the IIs that are normalized with respect to those from original code. From Table 2, we see that loop fusion enhances operation-level parallelism and successfully reduces the sums of the achieved minimum IIs.

Table 3 shows the number of execution cycles (top) and area (bottom) after HLS. The area results are shown in terms of the number of AND gates. From the table, we see that OLS provides significant speedups with a moderate increase in area compared to Orig. **Figure 6** shows the numbers of execution cycles (top) and area (bottom) after HLS that are normalized with respect to those from original code. As expected, the reduction rate of the number of execution cycles were almost the same as the reduction

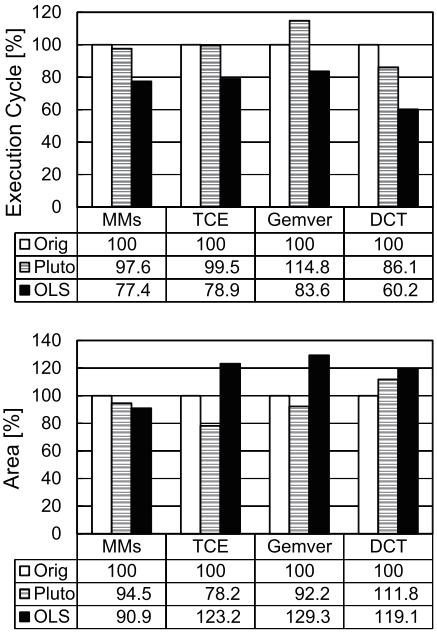


Fig. 6 The number of execution cycles and area after HLS (relative improvements in %).

rate of the minimum IIs shown in Table 2. On average, OLS could reduce the numbers of execution cycles by 25%, while Pluto reduced those only by 0.5% compared to Orig. This is because OLS could overlap the execution of statements from different loops and, as a result, the parallel execution of statements became possible. In addition, OLS could fuse loops into a single fully nested loop, so that the controllers are simplified and redundant states are eliminated. In Gemver, the II (and hence, the number of execution cycles) after Pluto were increased, although the loops in Gemver were partially fused after Pluto and the improvement of the operational-level parallelism was expected. This is because it happened the loop pipelining algorithm could not find a pipelined schedule with a reduced II under the tight clock constraint (5 ns). By relaxing the clock constraint, the II were successfully reduced. From Fig. 6, the area results after OLS increased by 15.6% on average, while the area results after Pluto was reduced by 5.8%. The area increase after OLS is due to the increasing resource requirements by the parallel execution.

5. Conclusion

This paper presented the loop transformation called Outer Loop Shifting (OLS). The state-of-the-art loop fusion algorithm, Pluto, optimizes locality and maximizes outer parallelism, however, fails to fuse loops in some important cases. OLS performs the post-processing to the schedules generated by Pluto and removes the scalar dimensions that prevent loop fusion by shifting the scheduling in the outer loop dimensions. Experimental results with high-level synthesis demonstrated that the synthesized hardware from the code after OLS took 25% less execution cycles on average than those of the synthesized hardware from the code after Pluto for the four benchmarks.

References

[1] Gajski, D.D. et al.: *High Level Synthesis: An Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).
[2] Kennedy, K. and Allen, J.R.: *Optimizing compilers for modern architectures: A dependence-based approach*, Morgan Kaufmann Publishers Inc. (2002).
[3] Singhai, S.K. and McKinley, K.S.: A Parametrized Loop Fusion Algorithm for Improving Parallelism and Cache Locality, *The Computer Journal*, Vol.40, No.6, pp.340–355 (1997).
[4] Feautrier, P.: Some efficient solutions to the affine scheduling problem. Part I. one-dimensional time, *Intl. Journal of Parallel Programming*, Vol.21, No.5, pp.313–347 (1992).
[5] Feautrier, P.: Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time, *Intl. Journal of Parallel Programming*, Vol.21, No.6, pp.389–420 (1992).
[6] Lim, A.W., Cheong, G.I. and Lam, M.S.: An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication, *Intl. Conf. Supercomputing*, pp.228–237 (1999).
[7] Bondhugula, U. et al.: Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model, *Intl. Conf. Compiler Construction* (*ETAPS CC*) (2008).
[8] Derrien, S., Rajopadhye, S., Quinton, P. and Risset, T.: High-Level Synthesis of Loops Using the Polyhedral Model, *High-Level Synthesis: From Algorithm to Digital Circuit*, Spinger (2008).
[9] Cong, J., Zhang, P. and Zou, Y.: Optimizing memory hierarchy allocation with loop transformations for high-level synthesis, *Design Automation Conference* (*DAC*), pp.1229–1234 (2012).
[10] Bastoul, C.: Code generation in the polyhedral model is easier than you think, *Intl. Conf. Parallel Architectures and Compilation Techniques* (*PACT*), pp.7–16 (2004).
[11] Pluto, available from ⟨http://pluto-compiler.sourceforge.net/⟩.

Yuta Kato received his B.S. and M.S. in electrical and electronic engineering from Tokyo City University in 2010, 2012, respectively. In 2012, he joined Mitsubishi Electric Corporation, Japan.

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

(Recommended by Associate Editor: *Takashi Takenaka*)