An Integrated and Automated Memory Optimization Flow for FPGA Behavioral Synthesis

Yuxin Wang,[1,3] Peng Zhang,[2] Xu Cheng,[1] Jason Cong[2,3]
[1]Computer Science Department, Peking University, China
[2]Computer Science Department, University of California, Los Angeles, USA
[3]UCLA/PKU Joint Research Institute in Science and Engineering and Center for Energy-Efficient Computing and Applications
ayerwang@pku.edu.cn, pengzh@cs.ucla.edu, chengxu@mprc.pku.edu.cn, cong@cs.ucla.edu

ABSTRACT

Behavioral synthesis tools have made significant progress in compiling high-level programs into register-transfer level (RTL) specifications. But manually rewriting code is still necessary in order to obtain better quality of results in memory system optimization. In recent years different automated memory optimization techniques have been proposed and implemented, such as data reuse and memory partitioning, but the problem of integrating these techniques into an applicable flow to obtain a better performance has become a challenge. In this paper we integrate data reuse, loop pipelining, memory partitioning, and memory merging into an automated optimization flow (AMO) for FPGA behavioral synthesis. We develop memory padding to help in the memory partitioning of indices with modulo operations. Experimental results on Xilinx Virtex-6 FPGAs show that our integrated approach can gain an average 5.8x throughput and 4.55x latency improvement compared to the approach without memory partitioning. Moreover, memory merging saves up to 44.32% of block RAM (BRAM).

Categories and Subject Descriptors

B.5.2 [**Hardware**]: Design Aids–*automatic synthesis*

General Terms

Algorithms, Performance, Design

Keywords

Behavioral Synthesis, Memory Partitioning, Memory Merging

1. INTRODUCTION

Automated synthesis flow from high-level specification (such as C or C++) to RTL implementation (such as Verilog HDL or VHDL) has been an active research goal for two decades. The advantages and productivity gained from behavior-level synthesis have been demonstrated by a number of state-of-the-art commercial tools such as AutoPilot from AutoESL/Xilinx [1], C-to-Silicon from Cadence [2], Catapult from Mentor Graphics [3], and Synphony from Synopsys [4]. However the traditional scheduling and binding algorithms are mainly designed for scalar operations. Most memory optimizations are carried out manually by experienced designers. In recent years, different automated flows for memory optimization have been developed, including data reuse and memory partitioning for pipelining. But research is required to integrate them into automated behavioral synthesis flows properly.

Data reuse has been extensively researched for bandwidth and power optimization during this decade (e.g., see [5, 6, 7, 8, 9, and 28]). Banakar [5] proposes the data reuse buffer as a power-efficient alternative for cache. Panda [6] partitions the scalar and array variables into off-chip DRAM with cache and on-chip reuse buffer according to the variables' size, lifetime, and conflicts. Kandemir [7] manages reuse for arrays in loops mainly through loop tiling. Issenin [8] builds a powerful dependence distance-based approach to cover reused data in sets. Cong [9] establishes a unified heuristic algorithm to manage reuse buffer allocation. Cong [28] combines loop transformation and memory hierarchy allocation to generate an on-chip reuse buffer. In all of this work, the reuse buffer is shared among all the arrays without access conflict because it is designed for a sequential execution model. However in pipelined loops, concurrent data requests may cause access conflict under port limitations of the physical RAMs. So directly combining data reuse and loop pipeline may not achieve the anticipated throughput improvement.

A large amount of previous work uses scheduling guided memory allocation and binding to avoid access conflict. Kim [10] chooses the best allocation and binding strategy by instruction-level macro-rescheduling and memory access operation-level micro-rescheduling. Wuytack [11] generates conflict graphs depending on loop ordering and scheduling, and uses the graphs to allocate conflict accesses into different physical memories. Although these scheduling guided methodologies can minimize access conflicts and reduce latency, the performance improvement is limited without some sort of data layout optimization—like data reuse.

A straightforward solution to reducing access conflicts is to increase the number of memory ports. However to implement a multi-port memory with quadratic growth in complexity and area (e.g., see [12]) is inefficient and unrealistic. As an alternative, partitioning memory into several banks can successfully manage port constraint with an acceptable overhead. Ho [13] designs a logical-to-physical mapping algorithm to break and pack memories into dual-port RAM. Benini [14] partitions the on-chip SRAM using an application-driven approach. Frequently accessed data is mapped into a small power-efficient memory, guided by application profiling. In reconfigurable architectures, Baradaran [15] attempts to map data arrays to the heterogeneous storage resources through memory distribution, replication, and scalar replacement. The approach combines the high-level specification with scheduling. For behavior-level synthesis, Ben-Asher [27] provides a profiling-based approach for increasing memory parallelism by data partitioning. That approach only considers the partitioning of elements in the data structures. Cong [16] automates memory partitioning in order to achieve the maximal throughput in loop pipelining. All of these previous methods require the indices to be affine. However the data reuse buffers are always updated circularly to save buffer size. This brings modulo operations into the indices. Under these circumstances, none of the above strategies work.

In addition to data reuse, pipelining, and memory partitioning, memory merging can also be taken into consideration in order to reduce the huge increase of banks. In fact, some integration efforts already exist. For example, Panda [26] combines array partitioning and merging in logical to physical mapping for low power. The approach splits the array into partitions according to access patterns

and merges the partitions into multiple memories. The partitioning is tile based. In contrast to that work, we integrate data reuse and loop pipelining in the flow. And the memory partitioning algorithm is based on indices analysis and can support modulo operations. Liu [17, 18] implements an integer non-linear programming model for data reuse and loop-level parallelization. This resolves the access conflict problem by using memory and data duplication. The integrated solution leads to a better performance, but memory duplication results in on-chip RAM increase and the redundant movement of data. Our automated optimization flow can solve the access conflict problem in a more efficient way, with the following contributions.

1) We develop an integrated method for FPGA behavioral synthesis by combining data reuse, loop pipelining, memory partitioning, and memory merging to optimize the throughput during pipelining and the area after memory partitioning.

2) For efficient partitioning of the reuse buffer, we propose a buffer padding algorithm to handle array indices with modulo operations.

3) We merge partitioned reuse banks into several larger reorganized reuse buffers, with consideration of the area overhead of address and control generation.

The remainder of this paper is organized as follows: Section 2 gives a motivational example for our integrated optimization; Section 3 describes the overall design flow and the detailed algorithm of memory partitioning and memory merging; Section 4 analyzes the experimental results, and is followed by conclusions in Section 5.

2. A MOTIVATIONAL EXAMPLE

Our example is from the motion compensation procedure in the H.264 official video decoder JM14.0 [19]. A loop belonging to the decoding procedure is given in Fig. 1(a). In the inner loop iteration, six pixels (*pp0 ~ pp5*) are accessed from an off-chip buffer (*lumabuffer*) as the inputs of a filter to interpolate the current pixel. In the next loop iteration, another six pixels are accessed, while five of them can reuse the previously fetched data (*pp1 ~ pp5*). Fig. 2(a) shows the five reused pixels (the solid circles) between iteration *{i=0, j=0}* and iteration {*i=1, j=0}*. We use the on-chip buffer *RUB* to temporarily store the reusable data until it will no longer be used and replace it with other reusable data; the rectangle in Fig. 2(b) indicates *RUB*. Based on the reuse pattern, different contents fill the buffer in different iterations. The dotted circle represents new reusable data fetched in the iteration *{i=1, j=0}* from *lumabuffer* to *RUB*. Meanwhile the *RUB[0]* is no longer reusable, so it is replaced by the new data. According to our reuse algorithm [9], the reuse buffer's size is six depending on the reuse distance in this example. We update the reuse buffer by replacing the non-reusable data with new reusable data cyclically. The transformed code for data reuse buffering is shown in Fig. 1 (b). It shows the cyclic updating feature which introduces modulo operations in the references.

Assuming that the on-chip memory has single port, the throughput improvement achieved by loop pipelining will be limited by the port constraint. Six cycles are required to read all the reuse data and update the new data in the buffer, as shown in Fig. 1(b). Partitioning the reuse buffer into six banks is a straightforward solution to reduce the access conflicts and improve the throughput of loop pipelining.

For complex applications in practice, such as the H.264 decoder, several reuse buffers are generated in different modules. Pre-obtained scheduling results provide information for access conflict analysis among reuse buffers. This gives us the opportunity to merge partitioned banks, which are scheduled in different time intervals, into less organized reuse buffers. For example, Fig. 3(a) shows a result of two loops after data reuse and memory partitioning. We assume that each loop is constructed in the same shape as *loop1* in Fig. 1(b). Based on our scheduling strategy, *loop1* and *loop2* are in different time intervals. Thus there is no conflict access between *RUB0_i* and *RUB1_i*. Our merging scheme associates *RUB0_i* with *RUB1_i* in the reorganized *RUBi*, as shown in Fig. 3(b), where *RUBi+offset* in *loop2* represents the address of *RUB1_i* in the reorganized buffer *RUBi*. After logical synthesis, the reuse buffers will be mapped into physical memories, and the area is saved due to the reduction of BRAMs and associated interconnections.

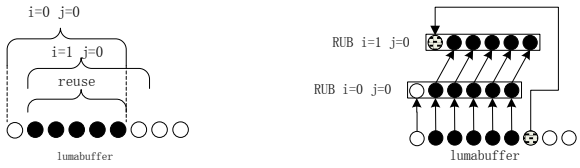
```
//loop1
for (j = 0; j < 16 j++)
   for (i = 0; i < 16; i++) {
   pp0 = lumabuffer[j][i];
   pp1 = lumabuffer[j][i+1];
   pp2 = lumabuffer[j][i+2];
   pp3 = lumabuffer[j][i+3];
   pp4 = lumabuffer[j][i+4];
   pp5 = lumabuffer[j][i+5];
   filter(pp0,pp1,pp2,pp3,pp4,pp5,&out,…); }
```

(a)

```
//loop1
imgpel RUB[6];
 for (j = 0; j < 16; j++)
 {  //Reuse buffer pre-fetch from lumabuffer
    RUB[0] = lumabuffer[j][0]; RUB[1]……
     for (i = 0; i < 16 i++) {
          pp0 = RUB[i%6];
          pp1 = RUB[(i+1)%6];
          pp2 = RUB[(i+2)%6];
          pp3 = RUB[(i+3)%6];
          pp4 = RUB[(i+4)%6];
          pp5 = lumabuffer[j][i +5];
          filter(pp0,pp1,pp2,pp3,pp4,pp5,&out,…);
          RUB[(i+5)%6]=pp5;   } }
```

(b)

Fig. 1. MC in H.264: (a) original program, (b) modified program after data reuse



(a) **(b)**

Fig. 2. (a) data reuse between iterations, (b) reuse data buffering

```
if(j==0)
loop1 (RUB0_0, RUB0_1, RUB0_2, RUB0_3, RUB0_4, RUB0_5)
else if(i==0)
loop2 (RUB1_0, RUB1_1, RUB1_2, RUB1_3, RUB1_4, RUB1_5)
```

(a)

```
// area(RUB0)=area(RUB0_0)+ area(RUB1_0)
 if(j==0)
loop1 (RUB0, RUB1, RUB2, RUB3, RUB4, RUB5)
else if(i==0)
loop2 (RUB0+offset, RUB1+offset, RUB2+offset, RUB3+offset,
RUB4+ offset, RUB5+offset)
```

(b)

Fig. 3. Example of optimization process: (a) two similar structures in the program after partitioning, (b) structure after merging

3. AMO FLOW DESCRIPTION

3.1 Design Flow Overview

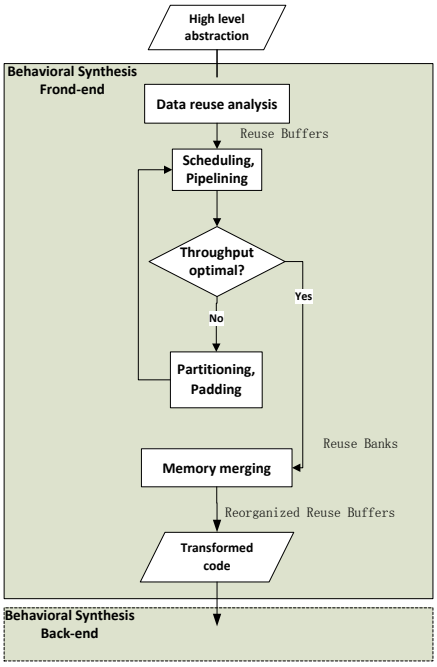


Fig. 4. AMO design flow

We will provide an overview of our AMO design flow in this section (as shown in Fig. 4). The high-level application specification is parsed into the behavioral synthesis front-end and followed by data reuse analysis [9] for the arrays. In this step reuse buffers are generated in different modules. Then scheduling and pipelining iterations are carried out to optimize the loops for the maximum throughput under certain resource constraints. A lightweight scheduling and pipelining algorithm is applied here to predict the result of these two steps [29]. Then, memory partitioning (and padding, if needed) is performed on the reuse buffers to reduce the access conflict. After reuse banks are generated, memory merging analysis searches for the optimal reuse banks merging scheme. The goal of this step is to reduce the area overhead caused by memory partitioning. It merges the reuse banks without access conflict into reorganized reuse buffers according to scheduling and area analysis. The transformed code will be synthesized into RTL implementation through the back-end of behavioral synthesis.

3.2 Reuse Buffer Partitioning and Padding

Loop pipelining is widely used for throughput optimization in data-intensive applications. But it is highly limited by resource constraints, such as the number of function units and memory ports. In Section 2 we illustrated that simultaneous accesses to the reuse buffer are often desired in a loop. This section presents a method to find an optimal memory partitioning scheme to achieve the target throughput in the loop pipeline. When the references in the buffer are affine, we formulate the problem as,

PROBLEM 1. Given k affine references on the same array, the target throughput requirement II, and memory port constraint pc, find a partition n such that target throughput is satisfied.

The approach in [16] partitions arrays with affine references. Assuming that the target throughput requirement *II=1* and the memory port constraint *pc=1* (the same assumption for the following problems)**,** we define the affine references to the array as $R_1=a_1*i+b_1$ and $R_2=a_2*i+b_2$, where *i* is the induction variable. The array is partitioned based on analyzing access conflict using the following theorem.

THEOREM 1. [16]

Let n_1 represent the number of banks, then

$$\forall i \quad a_1*i+b_1 \neq a_2*i+b_2 \mod n_1$$
$$\Leftrightarrow \gcd(a_1-a_2, n_1) \nmid (b_2-b_1)$$

However in the reuse buffers, the references have modulo operations. In some cases, THEOREM 1 is unable to attain the right partitioning, as shown in the following example.

EXAMPLE 1.

*For array RUB (the size is 99), the accesses RUB[i%99] and RUB [(7*i+1)% 99], n_1=2, s.t. gcd(6, 2) ∤ (1-0). The array is partitioned into two banks for conflict-free access. But when i=20, the two accesses, RUB[20] and RUB[42], are still in the same bank.*

Our approach is an improvement of previous work [16] that dealt with references to modulo operations. The partitioning problem is formulated as,

PROBLEM 2. Given k modulo references on the same array, the target throughput requirement II, memory port constraint pc, and the reuse buffer size m, find a partition n such that target throughput is satisfied.

To clarify our discussion, we unified the references in the form of $index_k=R_k\%m$. R_k denotes an affine function of induction variables. Then two array accesses can be represented as $index_1=R_1\%m$ and $index_2=R_2\%m$. We derive the following result to attain partitioning n_2 considering both the buffer size and the boundary effect due to modulo operation.

THEOREM 2.

$$\forall i \quad (a_1*i+b_1)\%m \neq (a_2*i+b_2)\%m \mod n_2$$
$$\Leftrightarrow \gcd(a_1-a_2, m, n_2) \nmid (b_2-b_1)$$

PROOF.

We prove the converse-negative proposition of THEOREM 2.

$$\forall i \ (a_1*i+b_1)\%m \equiv (a_2*i+b_2)\%m \mod n_2$$
$$\Leftrightarrow a_1*i+b_1 \equiv a_2*i+b_2 \mod m \mod n_2$$
$$\Leftrightarrow \exists i,h \ (a_1-a_2)*i+h*n_2 = b_2-b_1 \mod m$$
$$\Leftrightarrow \exists i,h,l \ (a_1-a_2)*i+h*n_2+l*m = b_2-b_1$$
$$\Leftrightarrow \gcd(a_1-a_2, m, n_2) \mid (b_2-b_1)$$ *(Bézout's lemma [25])*

EXAMPLE 2.

*For array RUB (the size is 99), and the accesses RUB[i%99] and RUB [(7*i+1)% 99], to achieve gcd(6, 99, n2) ∤ 1, the smallest partition is 3. But if we increase the buffer size by 1 (the size increased from 99 to 100), the smallest partition could be 2, s.t. gcd(6, 100, 2) ∤ 1. This illustrates the benefit of memory padding.*

It's trivial to prove that n_1 from THEOREM 1 is always smaller than or equal to n_2 from THEOREM 2. Considering n_1 is the optimal partitioning for PROBLEM 1 [16], we define n_1 as the lower bound of the partitioning for modulo references and n_2 as the upper bound. Then we need to solve the problem as,

PROBLEM 3. Given k modulo references on the same array, the partitioning lower bound n_1, the partitioning upper bound n_2, and the reuse buffer size m, find a partition candidate pairs set S, in which each pair consists of a partition n and a padding mp such that target throughput is satisfied.

If $n_1=n_2$, then n_1 is a partition candidate for the problem. Otherwise our AMO flow starts padding the reuse buffer to search for an available partitioning. A searching process is performed to find partition candidate pairs (*n, mp*) in which the partition *n* is in the range of [n_1, n_2], satisfying formula (1).

$$\gcd(a_1-a_2, m+mp, n) \nmid (b_2-b_1) \quad (1)$$

Because the searching space is not large, we enumerate all the possible partition candidate pairs. The rest of our extended

partitioning approach for *k* modulo references is similar to that in [16]. Scheduling is combined with partitioning for the throughput optimization in the loop pipeline. Our approach is unique in that, in contrast to previous work, we consider the trade-off between the buffer size and the number of memory banks. On one hand, padding decreases the partitions but increases the buffer size. On the other hand, the solution without padding keeps the size of the reuse buffer unchanged, but maybe largely increase the partitions, control logic and interconnection. As a result, we consider a combination of the partition *n* and the padding *mp* to minimize the reuse banks area. The area overhead minimizing problem is formulated as,

PROBLEM 4. Given partition candidate pairs set S on the same array for k modulo array references, and the reuse buffer size m, find a solution including the partition number n and a padding number mp that minimizes the area overhead.

We define the array area after partitioning as *area_partition* in formula (2) below.

$$\forall s = (n, mp) \in S,$$
$$area\_partition = memarea(m + mp) + \alpha * Inter\_p(n) \quad (2),$$

where $S = \{(n, mp)\}$ is the partition candidate pair set, in which each candidate pair is a solution to PROBLEM 1. Function *memarea()* is mapping from logical size to the physical area. *Inter_p(n)* is the control logic and interconnection due to the partition *n*. It is highly affected by whether the partition is a power of two. Coefficient α reflects the trade-off between the buffer size and the interconnection.

Our algorithm finds the optimal partitioning for the target throughput optimization and minimizes the area overhead. The complexity for Problem 3 applied on *k* references is $O(nk^2)$, where *n* is for the enumeration of partition number and k^2 is for checking formula (1) for each reference pair of the array.

3.3 Reuse Banks Merging

Memory partitioning can increase the throughput of loop pipelining. However, the size, amount and placement of BRAM are fixed in FPGA, so straightforward partitioning sometimes results in a large amount of small banks, which may occupy BRAMs inefficiently. When the BRAM resource is relatively insufficient, some arrays fail to be implemented on-chip, which leads to more accesses to the external memory with long latency and power consumption. To address the BRAM utilization problem, we design a merging scheme to efficiently merge partitioned banks into reorganized reuse buffers without the loss of throughput speedup gained by memory partitioning.

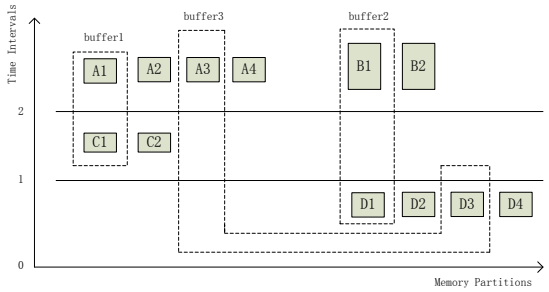


Fig. 5. Merging depends on time conflict and area

Automatic memory merging has been well studied in previous work and implemented in tools. Xilinx ISE can automatically merge two single-port memories into a dual-port BRAM without considering port sharing among the merged memories [24]. Conflict graph-based methods were proposed for the general memory mapping problem; they achieve minimal overhead according to scheduling [2, 7]. Compared to general memory mapping, our approach merges reuse banks considering address translation logic sharing.

As Fig.5 shows, the reuse banks are scheduled into the different time intervals. Shaded blocks represent reuse banks partitioned from the reuse buffers labeled by the capital letters in the names. The banks in the same time interval conflict with each other, which means they cannot be merged into one bank, such as *{A1, A2}* and *{A1, B1}*. Our approach analyzes reuse banks in the different intervals to seek the optimal merging strategy. Merging results are represented by dashed blocks. *A1* and *C1* are the first partitioning banks of reuse buffers *A* and *C*, respectively. So they could share part of the address translation logic. Our approach merges them into one bank, as *buffer1*. We formulate our merging problem as,

PROBLEM 5. Given reuse bank set V on r arrays, and the scheduling T for all the reuse banks, our goal is to find a merging scheme that merges reuse banks into reorganized reuse buffers set W to minimize the area without changing the throughput.

The optimal merging scheme is determined by minimizing the area overhead shown in formula (3) as below,

$$area\_merge = \sum_{0 \le i < g} memarea(M(i)) + \sum_{0 \le k < g} Inter\_m(V_{Wk}) \quad (3),$$

where *g* represents the number of reorganized reuse buffers, *M(i)* is the size of *ith* reorganized reuse buffer, and V_{Wk} is the subset of reuse banks merged into the *kth* reorganized reuse buffers. The function $Inter\_m(V_{Wk})$ is the area cost of the control logic and interconnection for reuse banks set V_{Wk}. We propose a heuristic algorithm, which merges the reuse banks into the minimum number of reorganized buffers. The area minimization problem in the approach is resolved as a sub-problem as following.

PROBLEM 6. Given a reuse bank v, and the reorganized reuse buffers set W, find a merging scheme that keeps the throughput unchanged and minimizes the area.

The area cost for this problem is shown in formula (4).

$$area_i(v, W_i) = memarea(M(i)) + Inter\_m(v, V_{Wi}), 0 \le i < g \quad (4)$$

To keep the throughput unchanged, the conflicting banks that are scheduled into the same time interval cannot be merged into the same reorganized buffer. So the minimum number of reorganized reuse buffers is the maximum number of conflict banks among all the time intervals. For example, the maximum number of conflict banks in Fig. 5 is six in time interval two. The following heuristic flow describes how we merge the reuse banks into the minimum reorganized reuse buffers.

Step 1: Assuming *g* as the maximum number of conflict banks for all the time intervals, create *g* empty reorganized reuse buffers.

Step 2: According to the scheduling *T*, arrange all the reuse banks in a bank queue in the increasing order of time intervals.

Step 3: Pick one reuse bank from the front of the bank queue and remove it from the queue. Find every reorganized reuse buffer which does not conflict with the current bank as a candidate merging buffer.

Step 4: For each candidate merging buffer, compute the area cost of merging with the current bank as shown in formula (4). Merge the current bank into the reorganized reuse buffer which minimizes the area cost.

Step 5: Repeat the approach from Step 3 until the bank queue is empty.

Our algorithm guarantees a solution with the minimum number of reorganized reuse buffers to allocate conflict banks at each time interval. From the algorithm, we can see the complexity of our merging approach is $O(g \times |V|)$, where $|V|$ (the number of reuse banks) is for the loop in Step 5, and *g* for the area minimization in Step 4.

4. EXPERIMENTAL RESULTS

We implement the AMO flow in C++ and evaluate it on five benchmarks. The optimized benchmarks are fed into the behavioral synthesis platform AutoPilot [1] to compile into RTL, and then implemented by Xilinx ISE 11.5 [20] on Xilinx Virtex-6. Among all of the five benchmarks, MYTEST is an integrated benchmark from several real-life applications, which we designed as controlled experiments. The other four benchmarks are data-intensive applications from medical imaging processing. DENOISE [21] removes noise from an image based on Rician-denoise. SEGMENTATION [22] detects objects in 2D/3D images. REGISTRATION [23] is a fluid registration algorithm. CONVOLUTION [30] is a TV-based deconvolution for medical imaging.

Our experimental data and comparison results are shown in Table 1. We report selected comparisons for every benchmark. Some of the results show the improvement after partitioning and others present the benefits from the whole approach. Columns three and four show the *throughput speedup* and *latency reduction* after data reuse, loop pipelining, and memory partitioning. The *throughput speedup* depends on the initiation interval (II) of the pipeline. The next two columns, which are from the Xilinx FPGA implementation reports, present *the number of lookup tables (LUT*) and *clock period (CP).* The column *BRAM w_pad* shows the amount of BRAMs after partitioning with padding, after memory merging, and the reduction of BRAMs after merging. The *BRAM wo_pad* column shows the results using the algorithm directly extended from [16] without padding. The last two columns are the *padding size* of reuse buffers and the padding area percentage of each benchmark.

Throughput speedup is shown in the first two columns. The columns are throughput and latency improvement after memory partitioning, compared to the approach that only uses data reuse and loop pipelining. There is a 5.8x throughput speedup on average, which is very similar to the speedup of 5.67x in [16]. The latency improvement is 4.55x on average. Memory partitioning increases the parallelism of the reuse buffer accesses and improves the throughput and latency.

Area optimization is presented in the rest of the columns. In the column *BRAM w_pad*, the average area decrease after memory merging is 44.32%. Among all the benchmarks, the best improvement (69.23%) is in MYTEST. Because the number of arrays and loops in MYTEST are more than in other benchmarks, there are more possibilities for reuse banks without merging conflicts. Moreover, each partition is small enough to store in a BRAM. According to our analysis, the number of banks after partitioning is directly related to the number of accesses in the loops.

The column *BRAM wo_pad* is the partitioning result without padding. The algorithm is a direct extension from [16]. The reuse buffer size m, the bank number n, the index $R_1=a_1*i+b_1$, and $R_2=a_2*i+b_2$ satisfy $gcd(a_1-a_2, m, n)\nmid(b_2-b_1)$. There is no available partition for benchmarks REGISTRATION and SEGMENTATION because a_1-a_2 and m are relatively prime numbers. Also the number of partitioning in CONVOLUTION is too high to be acceptable. Because of unsatisfactory results compared to the padding scheme, we ceased further experimentation on memory merging based on this scheme. Buffer padding can solve the problem easily with a slight overhead. The last two columns *padding (byte)* and *padding (%)* show the amount and percentage of the padding portion in the reuse buffer. Due to the partitioning lower boundn_1 and $gcd(a_1-a_2, n_1)$, the searching space of padding size k won't be too large.

In general, memory partitioning and merging increases the interconnection and control logic. But as shown in the *LUT* report, the increase is acceptable (around 10%). Also given the slightly changed *CP*, we believe that the critical path is almost unchanged after optimization.

Although padding can help the designers largely decrease the partitions, the subscript expressions of all the access references need to be revised. This modification is easy to perform on the automatically generated reuse buffers. Instead of statically padding the reuse buffer for modulo partitioning, extending the partitioning approach in [16] with a dynamic scheme that stalls the conflict accesses will also help solve the modulo-subscript problem. But the dynamic scheme needs to detect the access conflicts during execution and stall the whole pipeline when serializing the accesses. Compared to the small padding size, the dynamic control introduces a much larger hardware overhead, and the throughput will be impacted by the stall.

Our integrated flow AMO can also be used in ASIC behavioral synthesis. But we notice that the results for ASIC might be different than FPGA in some cases. Memory partitioning could still improve the throughput by reducing the access conflicts. But for on-chip SRAMs in ASIC, we may not gain much area savings after merging, because merging organizes small partitions into one BRAM and reduces the insufficient use of BRAM. Compared to a fixed-size BRAM, SRAM is available for various shapes and sizes. Although the port sharing and the interconnection reduction will optimize the area, the improvement is not as much as the FPGA design shows. As a result bank merging is more important for the FPGA design with fixed-size BRAMs.

5. CONCLUSIONS

In this paper we present an integrated automatic approach for combining memory partitioning and merging with data reuse and pipelining to generate a memory optimization flow for FPGA behavioral synthesis. To our knowledge, this is the first work to combine these four techniques in an automatic optimization flow and the first to solve the problem of memory partitioning for indices with modulo operations.

6. ACKNOWLEDGMENTS

This work was supported in part by the Semiconductor Research Corporation (SRC) under Contract 2009-TJ-1879, and in part by the National Science Foundation (NSF) under the Expeditions in Computing Program CCF-0926127. Yuxin Wang gratefully acknowledges the support from the China Scholarship Council (CSC). We would like to thank Xilinx for supporting the UCLA/PKU Joint Research Institute in Science and Engineering (JRI). Also we give special thanks to Janice Martin-Wheeler from UCLA for proof-reading.

7. REFERENCES

[1] AutoPilot, http://www.autoesl.com
[2] C-to-Silicon Compiler, Cadence, http://www.cadence.com/products/sd/silicon_compiler/
[3] Catapult C Synthesis, Mentor Graphics, http://www.mentor.com/esl/catapult/overview
[4] Synphony C Compiler, Synopsys, http://www.synopsys.com/Systems/BlockDesign/HLS/
[5] R. Banakar, S. Steinke, B. Lee, "Scratchpad memory design alternative for cache on-chip memory in embedded systems," in *Proc. of the 10th Int. Symp. on Hardware/Software Codesign (CODES)*, 2002, pp. 73 - 78.
[6] P.R. Panda, N.D. Dutt, A. Nicolau, "Efficient utilization of scratch-pad memory in embedded processor applications," in *IEEE Trans. European Design and Test Conference (ED&TC)*, 1997, pp. 7.
[7] M. Kandemir, J. Ramanujam, M.J. Irwin, et al, "A Compiler-Based Approach for Dynamically Managing Scratch-Pad Memories in Embedded Systems," in *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2004, pp. 243 - 260.
[8] I. Issenin, E. Brockmeyer, M. Miranda, et al, "DRDU: A Data Reuse Analysis Technique for Efficient Scratch-Pad Memory Management," in *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 2007, Vol. 12, No. 2, Article 15.

Table 1. Comparison of Different Benchmarks

		throughput speedup	*latency improve*	*LUT*	*CP(ns)*	*BRAM w_pad*	*BRAM wo_pad*	*padding(byte)*	*padding(%)*
MYTEST	partition	**3x**	**1.825x**	3289	4.860	26	38	12	1.7%
	merging			3684	4.929	8			
	improve			**-12%**	**-1.42%**	**69.23%**			
DENOISE	partition	**10x**	**2.675x**	11849	4.983	40	40	0	0
	merging			14715	4.997	30			
	improve			**-24%**	**-0.28%**	**25%**			
REGISTRATION	partition	**6x**	**7.02x**	32990	4.996	72	no solution	64	0.7%
	merging			33341	4.992	54			
	improve			**-1%**	**0.09%**	**25%**			
SEGMENTATION	partition	**7x**	**7.16x**	30938	5.000	36	no solution	32	0.3%
	merging			31723	4.998	18			
	improve			**-2.5%**	**0.04%**	**50%**			
CONVOLUTION	partition	**3x**	**4.11x**	6164	4.963	42	4096	32	0.3%
	merging			6938	4.893	20			
	improve			**-12.6%**	**1.41%**	**52.38%**			
AVERAGE	**improve**	**5.8x**	**4.55x**	**-10.4%**	**0.00%**	**44.32%**			

[9] J. Cong, H. Huang, C. Liu, Y. Zou, "A Reuse-Aware Prefetching Scheme for Scratchpad Memory," in *Proc. of the 48th Annual Design Automation Conference (DAC)*, 2011, pp. 960-965.

[10] T. Kim, J. Kim, "Integration of Code Scheduling, Memory Allocation, and Array Binding for Memory-Access Optimization," in *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2007, pp. 142 - 151.

[11] S. Wuytack, F. Catthoor, G. de Jong, et al, "Minimizing the Required Memory Bandwidth in VLSI System Realizations," in *IEEE Trans. on Very Large Scale Integration Systems (TVLSI)*, 1999, pp. 433 - 441.

[12] Y. Tatsumi, H. Mattausch, "Fast quadratic increase of multiport-storage-cell area with port number," in *Electronics Letters*, 1999.

[13] W.K.C. Ho, S.J.E. Wilton, "Logical-to-Physical Memory Mapping for FPGAs with Dual-Port Embedded Arrays," in *Field Programmable Logic and Applications, Lecture Notes in Computer Science*, 2004, pp. 111-123.

[14] L. Benini, L. Macchiarulo, A. Macii, et al, "Layout-driven memory synthesis for embedded systems-on-chip," in *IEEE Trans. Very Large Scale Integration Systems (TVLSI)*, 2002, pp. 96 - 105.

[15] N. Baradaran, P.C. Diniz, "A compiler approach to managing storage and memory bandwidth in configurable architectures," in *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 2008, Vol. 13, No. 4, Article 61.

[16] J. Cong, W. Jiang, B. Liu, Y. Zou, "Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization," in *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 2011, Vol. 16 Issue 2, Article 15

[17] L. Qiang, G.A. Constantinides, K. Masselos, et al, "Combining Data Reuse With Data-Level Parallelization for FPGA Targeted Hardware Compilation: a Geometric Programming Framework," in *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2009, pp. 305 - 315.

[18] L. Qiang, T. Todman, W. Luk, "Combining Optimizations in Automated Low Power Design," in *Proc. of Design, Automation and Test Europe (DATE)*, 2010, pp. 1791-1796.

[19] JM Software, H.264/AVC Software Coordination, http://iphome.hhi.de/suehring/tml/

[20] Xilinx ISE Design Suite, http://www.xilinx.com/

[21] P. Getreuer, "tvreg: Variational imaging methods for denoising, deconvolution, inpainting, and segmentation," online available: http://www.math.ucla.edu/getreuer/tvreg.html

[22] T. Chan, L. Vese, "Active contours without edges," in *IEEE Trans. on Image Processing*, 2001, vol. 10, no. 2, pp. 266-277.

[23] E.D'Agostino, F. Maes, D. Vandermeulen, and P. Suetens, "A viscous fluid model for multimodel non-rigid image registration using mutual information," in *the Int. Conf. on Medical Image Computing and Computer Assisted Intervention (MICCAI)*, 2002, pp. 541-548

[24] XST User Guide, online available: http://www.xilinx.com/itp/xilinx10/books/docs/xst/xst.pdf

[25] Frances Kirwan, "Complex Algebraic Curves," Cambridge University Press, 1992.

[26] P.R. Panda, N.D. Dutt, "Low Power Mapping of Behavioral Arrays to Multiple Memories," in *ACM/IEEE Int. Symp. on Low Power Electronics and Design (ISLPED)*, 1996, pp. 289 – 292.

[27] Y. Ben-Asher, N. Rotem, "Automatic memory partitioning: increasing memory parallelism via data structure partitioning," in *Proc. of the 8th Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010, pp, 155 - 162.

[28] J. Cong, P. Zhang and Y. Zou, "Combined Loop Transformation and Hierarchy Allocation in Data Reuse Optimization," in *Proc. of the 2011 Int. Conf. on Computer-Aided Design (ICCAD)*, 2011, pp. 185-192

[29] B. R. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," in *Proc. of the 27th Annual Int. Symp. on Microarchitecture*, 1994.

[30] P. Getreuer, "TV-Based Deconvolution for Medical Imaging," 2009.