Memory Access Scheduling and Loop Pipelining

Xuejun Liang* and Jack Jean**
*Department of Computer Science, Jackson State University
xuejun.liang@ccaix.jsums.edu
**Department of Computer Science and Engineering, Wright State University
jjean@cs.wright.edu

Abstract: A memory access scheduling technique is presented to avoid memory access conflicts in a loop pipelining computation on reconfigurable computers. The technique ensures that there is a modulo schedule of a pipelined loop that can achieve the minimum initiation interval of the pipelined loop provided that there is no loop carry dependency. Algorithms are presented to produce such a schedule to control the access of external memories. These algorithms have been applied to an automated FPGA design tool at Wright State University.

Keywords: Memory Access, Loop Pipelining, Design Automation, Reconfigurable Computing, and FPGA

1 Introduction

Loop pipelining is a common technique used for the speedup of the loop computations in reconfigurable systems. Modulo scheduling [1,2] is a particular loop pipelining technique, with which each iteration uses the same schedule and consecutive iterations are initiated at a constant rate, i.e. one initiation interval (II) apart. When the II is smaller than the latency of the loop body schedule, there exists some degree of overlapping between consecutive iterations. The smaller the II is, the higher the throughput of the pipelined loop is. However, the minimum II is usually restricted by the available hardware components and the dependency among loop iterations. In reconfigurable systems based on FPGA chips, the concurrency of hardware components can be very flexible. Special hardware components can be created according to the need of a particular schedule. Therefore, whenever there is enough area in the FPGA chip, the hardware components may not be the constraint of the minimum II. But, when the loop body involves the external memory accesses, because of the limited memory ports, the number of external memory accesses becomes the main restriction on the minimum II. In this paper, under the assumption that the FPGA chip has enough area and there is no loop carry dependency among loop iterations, the minimum II of a pipelined loop is proven to be the number of external memory accesses by the loop body provided that there is only one memory port. This result can be extended to multiple memory case directly.

This paper makes two contributes. First, in the design space exploration of pipelined loop computations that involve the external memory access in a multiple FPGA system, different memory access patterns [5], which are the number of memory readings and the number of memory writings of each memory port, can be enumerated to provide the design options without worrying about the memory access conflict. Second, the memory interface can be automatically generated with the proposed algorithms to control the external memory access of the pipelined loop.

Section 2 presents the concepts of memory access schedule, memory access control schedule, and memory access scheduling. Section 3 gives an algorithm to generate a memory access control schedule, which can be used to control the memory accesses for a pipelined loop with the minimum II, from a standard memory access schedule of the loop body. Section 4 presents an algorithm to standardize a memory access schedule. Section 5 concludes the paper.

2 Memory Access Scheduling

Figures 1 and 2 show typical memory access timing diagrams adapted from STARFIRE ™ Reference Manual [4]. When an internal circuit as shown in Figure 3 needs to read or write certain data from or to the on-board memory, designers need to assign a sequence of logic values to the control signals *Strobe_n* and *Write_Sel_n* at proper clock cycles, and to provide *Addr* with corresponding address values at correct cycles, according to the timing diagrams in Figures 1 or 2 and the specification of the internal circuit about when and from where the memory data is needed, or when and to where the result is ready for memory write.

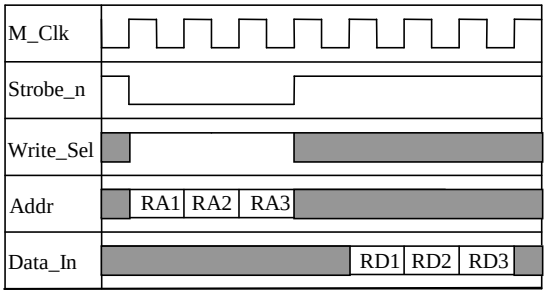

Figure 1: Typical Burst Read Cycle from On-board Memory

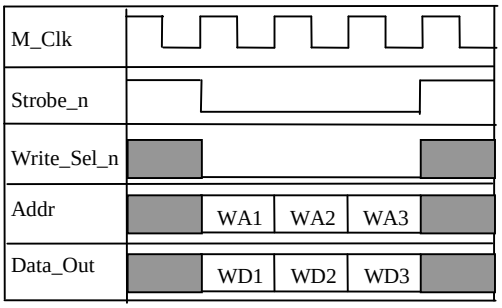

Figure 2: Typical Burst Write Cycle to On-board Memory

In general, the specification of the internal circuit about when and from where the memory data are needed, or when and to where the result is ready for memory write can be represented as an array whose index stands for the clock cycle and whose entry stands for the memory access (read, write, read and write, or none) and the memory location. For simplicity, the memory location information will be not considered in the following discussion without loss of generality, and the following numbers are used to identify the memory access activities; 1 for Read, 2 for Write, 3 for Read and Write, and 0 for None.

For example, the following array specifies that memory data are needed for the first two consecutive clock cycles, and in the last clock cycle a result is ready to be written back to memory.

Clock Cycle (Index)	0	1	2	3	4
Memory Access	1	1	0	0	2

Table 1: A Memory Access Schedule

The array is called a **memory access schedule**. Note that the memory access schedule is for a single memory port. When the internal circuit uses multiple memory ports, there will be multiple memory access schedules, each corresponding to one memory port. A memory access schedule is said to be **standard,** if all the reads are consecutive and begin from the first clock cycle, and all the writes are consecutive and end at the last clock cycle. For example, the above memory access schedule is standard.

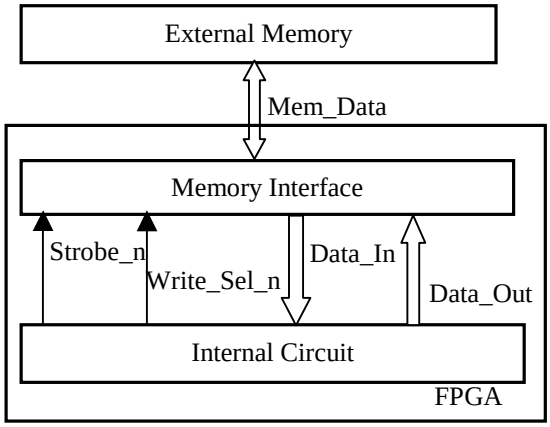

Figure 3: Internal Circuit Accessing External Memory via Memory Interface

The task of assigning a sequence of logic values to the memory access control signals is called the **memory access control scheduling**, and for each memory access control signal, the sequence of logic values can also be represented as an array with index indicating the clock cycle and is called the **memory access control schedule**

For a single port memory bank, a **memory access conflict** occurs when more than one datum is put on the memory data bus Mem_Data as shown in Figure 3 in the same clock cycle. Therefore, a memory access schedule that is free of memory access conflicts cannot have more than one reading or more than one writing at the same clock cycle. However, a memory access schedule that reads and writes at the same clock cycle may not cause memory access conflicts when reading and writing have different delays.

A memory access schedule has **memory access conflict** if one of its memory access control schedules has more than one different logic value at one clock cycle. For example, when reading delay is 4 clock cycles and writing delay is 0 clock cycle, the following memory access schedule has a memory access conflict because it requires *Write_Sel_n*[0] to be both 0 and 1. That is impossible.

Clock Cycle (Index)	0	1	2	3	4
Memory Access	2	0	0	0	1

By using the above terminology, the memory access timing diagrams are designed to guide how to obtain a memory access control schedule for each control signals from a memory access schedule. The guidance of memory access timing diagrams, for example, those shown in Figures 1 and 2, can be abstracted as the following: (1) The default value of *Strobe_n* is 1 (high) and the default value of *Write_Sel_n* is –1 (high impedance). (2) If reading at clock cycle n then *Strobe_n*[n-4] = 0, and *Write_Sel_n*[n-4] = 1, where the constant 4 is actually a reading delay time (cycles). (3) If writing at clock cycle n then *Strobe_n* [n-0] = 0, and *Write_Sel_n* [n-0] = 0, where the constant 0 in the index is the writing delay time (cycles). The guidance rules can be represented in terms of the memory access schedule. For simplicity, the default logic values of the control signals are not considered.

If *data*[n] = 1 or 3 then
- *Strobe_n*[n-d_R]=0, and
- *Write_Sel_n*[n-d_R] =1

If *data*[n] = 2 or 3 then
- *Strobe_n* [n-d_W] = 0, and
- *Write_Sel_n* [n-d_W] = 0

where *data* is a memory access schedule, d_R is the reading delay cycles, and d_W is the writing delay cycles. It is assumed that $0 \le d_W \le d_R$ without loss of generality.

Let *data* be a memory access schedule, Algorithm 1 computes the memory access control schedules for *Write_Sel_n* and *Strobe_n*. For example, assume d_R=2 and d_W=0 and let *data*=(1, 2, 1, 2). Then the following result can be obtained from Algorithm 1.

Index	-2	-1	0	1	2	3
Strobe_n	0	1	0	0	1	0
Write_Sel_n	0	-1	0	1	-1	1
data			1	2	1	2

Algorithm 1: **Memory Access Control Scheduling**

```
For each index n of data do
   Strobe_n[n]← 1
   Write_Sel_n[n] ← -1
For n ← the first index to the last index do
   If data[n] = 1 or 3 then
      If Write_Sel_n[n- d_R] = 0 then
         Return "memory access conflict";
      Else {
         Strobe_n[n- d_R] = 0;
         Write_Sel_n[n- d_R] = 1; }
   If data[n] = 2 or 3 then
      If Write_Sel_n[n- d_W] = 1 then
         Return "memory access conflict";
      Else {
         Strobe_n[n- d_W] = 0;
         Write_Sel_n[n- d_W] = 0; }
```

It can be noted that a memory access schedule causes a memory access conflict if and only if it requires multiple logic values assigned to the same control signal at the same time (clock cycle). The conflict may be avoided if memory write operations are allowed to be delayed. It is feasible by adding delay components on the bus Data_Out in Figure 3. Delaying a memory write operation by adding delay components is called the memory write scheduling. On the other hand, a memory read operation, by a similar argument, can be scheduled to perform before the datum is consumed by the internal circuit. In this case, delay components need to be added on the bus Data_In in Figure 3 so as to keep the right timing. This is called the memory read scheduling. Both the memory read scheduling and the memory write scheduling are called the memory access scheduling.

3 Loop Pipelining with Modulo Scheduling

In this section, it is assume that there is a single memory port connecting to an FPGA chip and the FPGA chip has enough area. Let N_{RD} and N_{WR} be the number of reads and the number of writes by the loop body computation. It is obvious that the minimum II of the pipelined loop cannot be less than $N_{RD}+N_{WR}$.

Without considering the external memory access timing, the loop body can be scheduled with its II equal to $N_{RD}+N_{WR}$. The resulting modulo schedule of the loop body contains the information about when the input data from

memory are consumed and when the results that are needed to be stored in the memory are ready. This information is, in fact, a memory access schedule of the loop body. In general, this memory access schedule may cause memory access conflicts during the pipelined loop computation.

In this section, it is proven that when the memory access schedule of a loop body is standard, memory access conflicts during the pipelined loop computation with II equal to $N_{RD}+N_{WR}$ can be avoided by performing the memory write scheduling to the memory access schedule of the loop body. Note that using the memory read scheduling is also possible. Also note that the pipelined loop computation will remain the same. In the next section, this result is generalized to any memory access schedule of a loop body by transforming the memory access schedule to a standard one.

A standard memory access schedule can be simply represented as a 3-tuple (N_{RD}, N_C, N_{WR}), where N_{RD} is the number of reading cycles, N_C is the number of other computation cycles, and N_{WR} is the number of writing cycles. It is shown as follows.

N_{RD}	N_C	N_{WR}

The top part of Figure 4 shows the execution of four consecutive iterations where each iteration is initiated with II=$N_{RD}+N_{WR}$ cycles apart. In order to avoid memory access conflicts, the memory write schedule may be delayed by *D* clock cycles as shown in dashed boxes, while the memory read schedule remains unchanged. The bottom part of Figure 4 shows such a memory control signal schedule, where the dashed arrows show the memory access control schedule for the read operations, and the spaces labeled N_{WR} are placeholders of the memory access control schedule for the write operations.

Assume that the memory write control schedule for the first iteration is scheduled after $m\times(N_{RD}+N_{WR})+N_{RD}$ clock cycles (m=1 in Figure 4). The memory write control schedule for the remaining iterations can be easily settled. In order to minimize the delaying cycles of memory write operations, i.e., minimizes *D*, the number *m*, which is called the **prologue number,** can be computed with the following formula.

$$m = \left\lceil \frac{N_C - d_W + d_R}{N_{RD} + N_{WR}} \right\rceil$$

The delay cycles of the write operation can be computed by

$$D = m\times(N_{RD} + N_{WR}) - d_R - N_C + d_W$$

Note that in order for the write operations to keep the right timing after memory write scheduling, *D* delay registers are needed to delay the output results to the memory of the loop body circuit

Theorem 1: Assume that a loop body has a standard memory access schedule (N_{RD}, N_C, N_{WR}) and the number of the loop iterations is N_{ITER}. Then the minimum II of the pipelined loop is $\mathbf{N_{RD}+N_{WR}}$ and the number of iterations of steady state of the pipelined loop is $\mathbf{N_{ITER} - \boldsymbol{m}}$. The total number of cycles required to execute the modulo schedule is $\mathbf{II_{MIN}\times(N_{ITER}+m)}$.

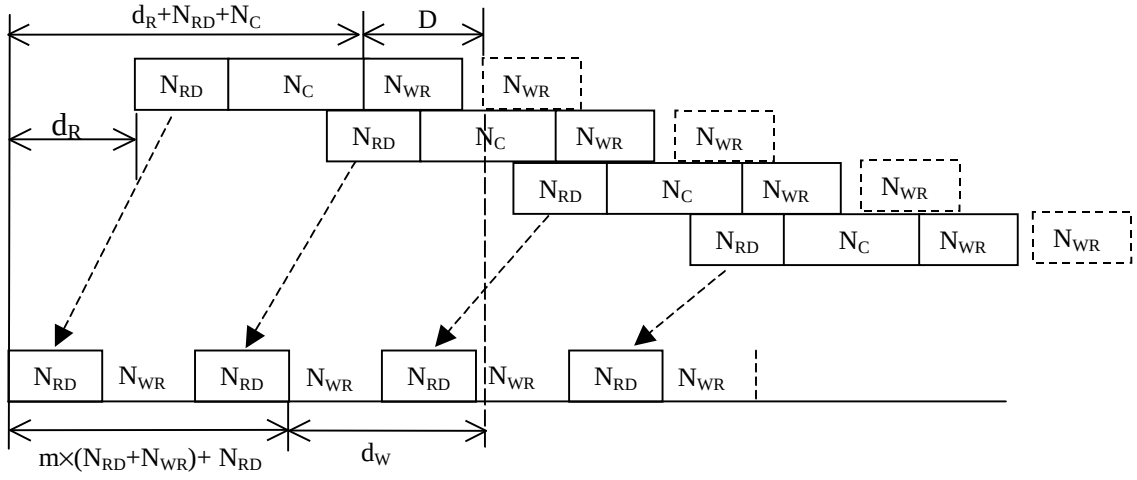

Figure 4: Loop Pipelining with a Modulo Schedule (m=1)

Recall that a memory access schedule (N_{RD}, N_C, N_{WR}) has an array representation. For example, *data*=(6,3,2) is equivalent to *data*= (1,1,1,1,1,1,0,0,0,2,2), where N_{RD}=6, N_C=3 and N_W=2. A right shift operation RSH(*index, distance, data*) of a memory access schedule (an array) is defined to shift the array elements that are in between *index* and the last index to the right by *distance* and insert *distance* number of 0's in the undefined entries after shifting. For example, RSH(9,1,*data*)=(1,1,1,1,1,1,0,0,0,0,2,2) and RSH(0,8,*data*)=(0,0,0,0,0,0,0,0,1,1,1,1,1,1, 0,0,0,2,2). Addition of two arrays is defined to add corresponding entries of the two arrays. If two arrays to be added are of different length, the undefined entries of the shorter array at the end are considered to be zero.

Algorithm 2 generates the memory access control schedules for the prologue, the steady state and the epilogue of the pipelined loop from a standard memory access schedule of a loop body, *data*.

Algorithm 2: **Generate the memory access control schedules of pipelined loop**.

(1) Compute the minimum II.
$II = N_{RD}+N_{WR}$

(2) Compute the prologue number.
$$m = \left\lceil \frac{N_C - d_W + d_R}{N_{RD} + N_{WR}} \right\rceil$$

(3) Compute the delay cycles of the write operations.
$$D = m \times (N_{RD} + N_{WR}) - d_R - N_C + d_W$$

(4) Shift all writing operations in *data* to the right by D, i.e.
data = RSH(N_{RD}+N_C,D,*data*)

(5) Shift and add schedules
data_s = *data*;
For i=1 to *m* do
data = *data_s* + RSH(0, II, *data*);

(6) Using Algorithm 1 to perform Memory Access Control Scheduling for *data*

(7) Number of execution cycles of prologue:
II×*m*.
Number of execution cycles of steady state:
II×($N_{ITER} - m$)
Number of execution cycles of epilogue:
II×*m*.

Example 1: Assume *data*=(1,1,0,0,2), d_R=2 and d_W=0. Then
(1) II=3. (2) *m*=2. (3) *D*=2.
(4) *data* = (1,1,0,0,0,0,2).
(5) *data* = (1,1,0,1,1,0,3,1,0,2,0,0,2)
(6) *Write_Sel_n* = (0,0,-1,0,0,-1,0,0,1,-1-1,1,-1, -1, 1)
Strobe_n = (0,0,1,0,0,1,0,0,0, 1,1,0,1,1,0)
(7) **Prologue**: length = *m*×II=6
Write_Sel_n = (0,0,-1,0,0,-1)
Strobe_n = (0,0, 1,0,0, 1)
Steady State: length=II=3
Write_Sel_n = (0,0,1)
Strobe_n = (0,0,0)
Epilogue: length= *m*×II =6
Write_Sel_n = (-1-1,1,-1,-1,1)
Strobe_n = (1, 1,0, 1, 1,0)

Note that when an FPGA buffer is used to store the incoming external memory data, the loop body gets data from the buffer. In this case, the number of external memory accesses by the pipelined computation is reduced, and then the minimum II is reduced. In this case, the above theorem still holds, but the above algorithm needs to be modified slightly. First, the reading delay cycles should be d_R+1 because it needs one clock cycle for data to enter the buffer. Second, the loop body computation begins only after the FPGA buffer is fully filled with the required external memory data. Therefore, the memory readings in the current iteration are used for the next iteration, and the last iteration of the loop does not need to read external memory again. (Please see [3] for details.)

4 Standardizing Memory Access Schedule

This section presents a technique and corresponding algorithms to transform a memory access schedule into a standard one by using memory write scheduling to delay the write operations and using memory read scheduling to advance the read operations.

In case of memory write scheduling, a delay component may be needed for the memory write operations specified in the original memory access schedule to keep the correct timing. A circular queue and a multiplexer as shown in Figure 5 are used for this purpose.

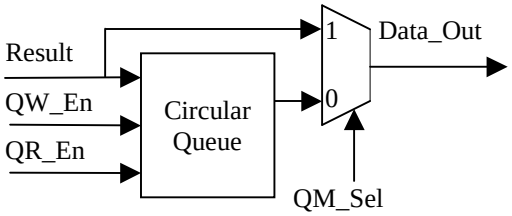

Figure 5: Circular Queue with Memory Write Scheduling

The logic values for the signals *QW_En* (queue write enable), *QR_En* (queue read enable) and *QM_Sel* need to be determined. Note that either the multiplexer or the circular queue may not be necessary depending on a particular memory access schedule. Also, the length of the circular queue should be as short as possible.

Now look at an example of memory write schedule. As shown below, the memory access schedule contains six write operations. The data that are supposed to write back into the memory are recorded instead of write operation 2.

0	1	2	3	4	5	6	7	8	9
	3	4		8	2		5		7

The above memory access schedule is to be standardized as

0	1	2	3	4	5	6	7	8	9
				3	4	8	2	5	7

A circular queue of length two and a multiplexer are needed. The logic values for the signals *QW_En*, *QR_En* and *QM_Sel* as well as the contents in the circular queue at each clock cycle are listed as below, where at clocks 2, 3, and 4, the datum 3 is at the head of the queue, and at clocks 5, 6, 7, and 8, the data 4, 8, 2, and 5 are at the head of the queue, respectively.

	0	1	2	3	4	5	6	7	8	9
QW_En	0	1	1	0	1	1	0	1	0	0
QR_En	0	0	0	0	1	1	1	1	1	0
QM_Sel	0	0	0	0	0	0	0	0	0	1

	0	1	2	3	4	5	6	7	8	9
Queue				4	4	8	2			
Head			3	3	3	4	8	2	5	

Assume that the memory access schedule is represented as an array, *data*, and the schedule length is N, the following algorithm calculates the schedules of the signals *QW_En*, *QR_En*, *QM_Sel* and the minimum length L_{MIN} of the circular queue. The algorithm deals with two cases, one requiring the multiplexer and the other not. It scans the original schedule twice. In the first scan, the number of data needed to store in the queue and the number of data not needed to store in the queue are computed, and recorded in L_{MIN} and N_k respectively. Also the logic values of the signals *QW_En*, *QM_Sel* at each clock cycle are computed. In the second scan, the logic values of the signal *QR_En* are computed, the number of data needed to store in the queue during the queue reading period are subtracted from L_{MIN} computed in the first scan. Therefore, after the second scan, the minimum length of the circular queue is the value of L_{MIN}.

Algorithm 3: Compute *QW_En, QR_En, QM_Sel,* and L_{MIN} for write operations:

(1) Case 1: *data*[*N*-1] = 2 or 3

The circular queue and the multiplexer are needed only if there exist n_1 and n_2 such that $0 \le n_1 < n_2 < N$-1 and *data*[n_1] is 2 or 3 and *data*[n_2] is neither 2 or 3.

(1.1) Compute *QW_En* and *QM_Sel*

```
N_k = 0; flag = 1; L_MIN = 0;
For n ← N-1 downto 0 do
   QW_En[n] = 0;
   QM_Sel[n] = 0;
   If flag = 1 then
      If data[n] = 2 or 3 then
         QM_Sel[n] = 1; N_k++;
      Else flag = 0;
   Else
      If data[n] = 2 or 3 then
         QW_En[n] = 1; L_MIN++;
```

(1.2) Compute *QR_En* and L_{MIN}

```
num = L_MIN;
For n ← N-1 downto N-N_k do
   QR_En[n] = 0;
For n ← N-N_k-1 downto N-N_k-num do
   QR_En[n] = 1;
   If data[n] = 2 or 3 then
      L_MIN --;
For n ← N-N_k-num-1 downto 0 do
   QR_En[n] = 0;
```

(2) Case 2: *data*[*N*-1] is neither 2 nor 3

The multiplexer is not needed. The circular queue is needed only if there exists n such that n < *N*-1 and *data*[n] is either 2 or 3.

(2.1) Compute *QW_En*

```
L_MIN = 0;
For n ← N-1 downto 0 do
   QW_En[n] = 0;
   If data[n] = 2 or 3 then
      QW_En[n] = 1; L_MIN++;
```

(2.2) Compute *QR_En* and L_{MIN}

```
num = L_MIN
For n ← N –1 downto N-num do
   QR_En[n] = 1;
   If data[n] = 2 or 3 then
      L_MIN --;
For n ← N-num-1 downto 0 do
   QR_En[n] = 0;
```

In case of memory read scheduling, a delay component may also be needed for the read operations specified in the original memory access schedule to keep the correct timing. Similarly with the memory write scheduling, a. circular queue and a multiplexer are needed. (Please see [3] for details.)

5 Conclusions

When a loop body computation uses multiple memory ports, there are multiple memory access schedules of the loop body, each for one memory port. In that case, the minimum initiation interval is the maximum number of memory access in all schedules. Therefore, the results in this paper can be extended to the case of multiple memory ports directly by applying the memory access scheduling to each individual memory schedule.

From the results presented in this paper, it can be seen that the minimum II of a pipelined loop can be determined by the external memory accesses, and the memory access scheduling and the memory access control scheduling can be carried out separately from the loop computation synthesis. This technique enables a pipelined design of loop computations to achieve the maximum throughput and makes the loop pipelining synthesis independent with the underlying memory access timing and free of the memory access conflict consideration. Thus it reduces the complexity of the corresponding synthesis (modulo scheduling) algorithm. This technique also enables the automatic memory controller generation for a pipelined loop computation.

The presented memory access scheduling algorithms have been applied to an automated FPGA design tool at Wright State University [3,5,6]. The tool maps generalized template matching operations onto a reconfigurable computer, a host computer with an FPGA board, and produces an optimal FPGA design running the FPGA board. The current design tool can produce VHDL codes for the optimal mapping results targeting the WildForce FPGA board.

6 References

[1] A. E. Charlesworth, "An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family", in Computer, 14(9):18-27, September 1981

[2] M. Lam, "Software Pipelining: An effective scheduling technique for VLIW machines," in Proceeding of the ACM SIGPLAN' 88 Conference on Programming Language Design and Implementation, pp. 318-328, June 1988

[3] X. Liang “Mapping of Generalized Template Mapping on Reconfigurable Computers”, Ph.D dissertation, Wright State University, December 2001

[4] "StarFire ™ Reference Manual", Annapolis Systems, Inc.

[5] X. Liang and J. Jean, “Memory Access Pattern Enumeration in GTM Mapping on Reconfigurable Computers”, Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 8-14, June 2001

[6] X. Liang, J. Jean and K. Tomko, “Data Buffering and Allocation in Mapping Generalized Template Matching on Reconfigurable Systems”, The Journal of Supercomputing, Special Issue on Engineering of Reconfigurable Hardware/Software Objects, 19(1):77-91, 2001