Github link: https://github.com/Go0zx/EE599_ZixiGu_6974574312.git

FPGA-based High Throughput Parallel Hash Table with hazard control

Zixi Gu
Ming Hsieh Department of
Electrical and Computer
Engineering
University of Southern California,
Los Angeles, CA 90089
zixigu@usc.edu

1 Introduction

Hash table is widely used in a lot of areas like using in encryption applications and network application and also has trend in machine learning or AI applications. As an index structure, hash table can find and retrieve an item quickly, which gives it advantage in fast data access. Parallel Hash Table is kind of hash table, but instead of inserting, finding or deleting one key at a time, each time can support multiple operations.

Nowadays, with different platforms appearing, Field Programmable Gate Array (FPGA) is a promising platform for implement Parallel Hash Table, as its low latency, low cost, and high energy efficiency. FPGA on-ship memory is a very useful resource for me to consider about, and I want to use it to implement a fast hash table.

There are several works for developing high-throughput hash table by parallelizing. **FPGA-based High Throughput Parallel Hash Table** [1] implements a parallel design with separate memory block each Process element support throughput as high as 5360 million operations per second with PEs running at 335 MHz for static hashing. However, this design cannot handle any type of hazard, RAW like PE0 Search(10) after Insert(10,100); WAW like PE0 insert(10,100) when PE1 insert(10,200).

In this project, I propose a high throughput Parallel Hash Table using FPGA on-chip memory **with hazard control**, which will follow **FPGA-based High Throughput Parallel Hash Table's** [1] idea and make it more reliable. The proposed architecture supports concurrent operation including search, insert and can handle most of unsafe hazard. The length of key can be 16, 32, 64 bits, value can be 32, 64 bit and number of thread P is larger than 1.

p: the number of PE.

2 Problem Definition

Data hazards can seriously influence the result of instruction, even though in the shallow pipeline we need to design control unit to handle them, like in the basic 5 stage pipelining microprocessor, we need hazard control unit to address with data hazard, etc. With pipeline deeper, data hazards can cause more often than shallower one. And parallel design causes this situation even worse. For 4 PE design, there are 7 stages in the pipeline, which means totally 4*7=28 operations running at same time; For number of PE increasing, the frequency of data hazards happen will be increasing largely.

Each insert operation in **FPGA-based High Throughput Parallel Hash Table** [1] takes long time if it updates blocks, search or insert with same key operation should be very carefully processed, so it is necessary to add hazard controller, otherwise operations need to be well rearranged from software side.

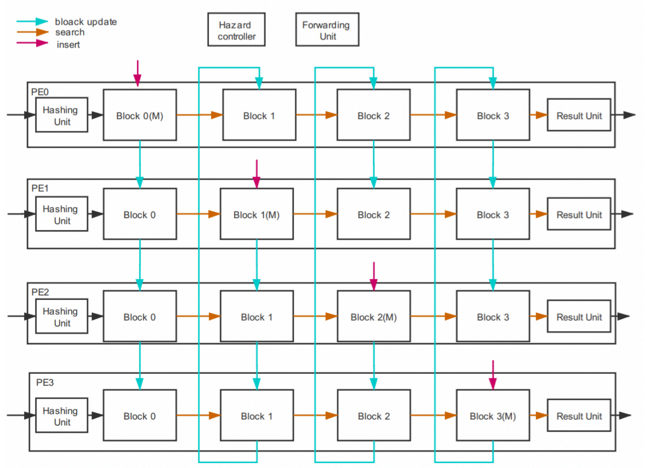

Figure 1: Overall architecture 4 PE as example

2.1 Objective

In my work, I want to add some control logic to improve its reliability, but not to decrease its performance or a little bit is accepted. For WAW hazard, hazard unit will flash the operation entering most recently or flash the operation based on fixed priority. For RAW hazard, forward unit will forward result when the operation is finished to the stage causing RAW hazard. The overall architecture is not change, but two units (hazard controller and forwarding unit) are added as illustrated in Figure 1. And detail in Figure 2.

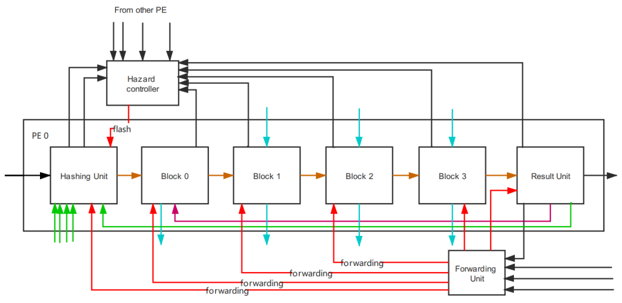

Figure 2: Overview one PE in the whole architecture

2.2 Key Idea

The key idea for detecting data hazard is first check operation type, then from the result select these operations can cause hazard and compare their key values. For avoiding too much time cost, these comparisons are all doing parallel, which means all comparison should be done at very closely timing.

For comparing opcode and key value, all of them are directly getting from stage registers, there won't be any extra memory space cost. Total number of comparisons:

1) for **RAW hazard** is (p+2) *p*p
2) for **WAW hazard** is (p+3) *p*p

The width of comparison will be based on key width.

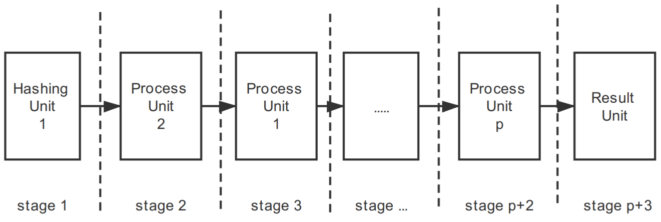

Figure 3: Pipelined architecture

3 Design Details

In the proposed design, each memory block in PE has multiple slots with own valid bit. When doing insert operation, first thing is checking slot valid bit from first slot to following's, if one slot is available, it can be insert value successfully.
The operations are mapped into a p+3 stage pipeline as illustrated in Figure 3.

1) **hash unit (stage 1–stage 2):** Transform key values.
2) **process unit (stage 3–stage p+2):** Process operations.
3) **result unit (stage p+3):** Output result and enable write.
4) **hazard controller:** Control hazard.
5) **forwarding unit:** Forward result.

All operation input should be this format:

[valid | function opcode | key | value]

All operation output will be this format:

[Result valid | function opcode | key | value]

3.1 Hash unit

The hashing function is based on class H3 hash function [2], which has been proved as an effective method in distributing random keys among hash table entries. This hashing function is defined as follows:

Definition 1. Let K be a set of keys with i bits, and H be a set of hash index with j bits. And let Q be a set of i*j matrix. Given q(n)=Q(n,), nth row of Q; k(m)=K(m), mth bit of K. h=H as final index.

$$h = (k(0) \cdot q(1)) \oplus (k(1) \cdot q(1)) \oplus \dots \oplus (k(i) \cdot q(i))$$

The hash unit are mapped into 2 stages, like in **FPGA-based High Throughput Parallel Hash Table [1].** Based on **hashing** function method I use, the first stage is doing AND operation, then second stage will do XOR operation.

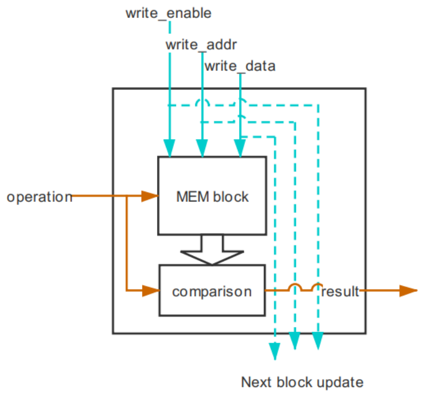

Figure 4: PE overview Master unit as example

3.2 Process unit

The process unit is the basic unit to process different operations. And there are two direction signals will go, horizontal and vertical. As Figure 4 shows.

For the horizontal part, process unit logic unit will do the search operation which read from memory block within its range.
If key has already existed, result valid will keep 0, otherwise result valid will be 1.

For the vertical part, process unit logic unit will do block update, which means write the data into the memory block within its range if need.

3.3 Result unit

The Result unit is the final unit to generate output signal and is able to start the block update operation for insert operation.

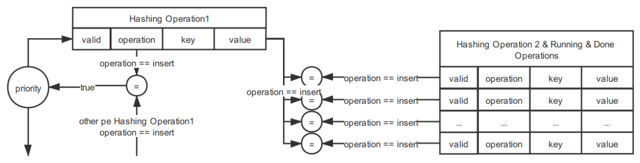

Figure 5: WAW hazard comparison

3.4 Hazard controller

The Hazard unit is for detecting WAW hazard between new operation and old operation. First, all insert operations will be found and based on the result of checking insert operations, the latest insert operations which from first stage (hash unit 1) will compare with running insert operations with their key value. Show as figure 5. WAW hazard can happen when PE0 insert(10,100) while PE1 insert(10,200). Two situations could happen.

1) **special case** (at same time): Flash the operation with lower priority.

2) **common case** (at different time): Flash the latest operation.

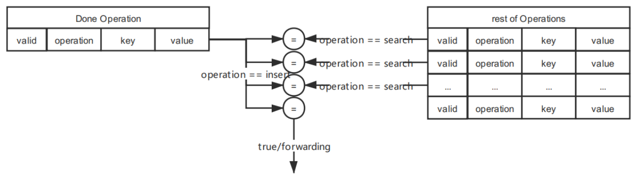

Figure 6: RAW forward comparison

3.5 Forwarding unit

The Hazard unit is for all RAW hazard to detect if there is some operation need to be helped. First, all search operations will be found and based on the result of checking search operations, the oldest insert operations which from last stage (result unit) will compare with running search operations with their key value. Show as figure 6. RAW hazard can happen when PE0 Search(10) after Insert(10,100).

4 Experimental Setup

I implement the proposed design on FPGA and test by vivado. My target device is Xilinx Zynq UltraScale+. All reported results are post place and route results using Xilinx Vivado 2019.2.

The key sizes I use in My experiments are 16, 32, 64 bits, and data value size is 32, 64 bits. And address width for entry is also modified for 4 PE with 7, 8, 9, 10, 11 bits. All experiments are with 2 slots.

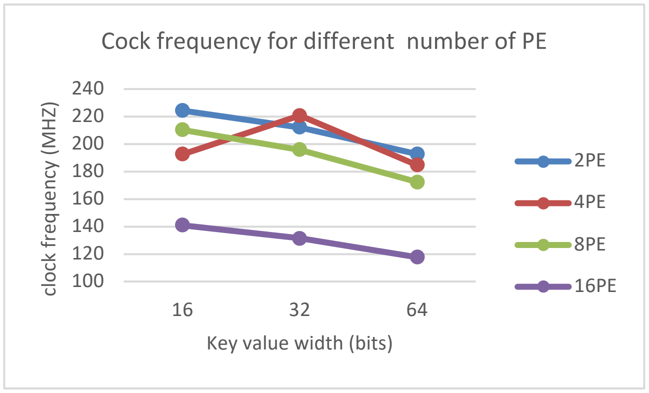

Figure 6(a): clock frequency required versus key width 16,32,64 bits with data width 32 bits, entry address width 7 bits, 2 slots.

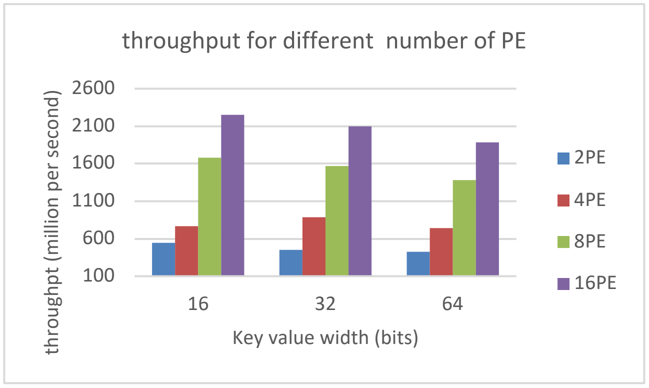

Figure 6(b): throughput versus key width 16,32,64 bits with data width 32 bits, entry address width 7 bits, 2 slots.

5 Analysis of Results

As Figure 6(a) shows, with number of PE increasing, clock frequency required is decreasing linearly and when p=16, the clock frequency decreased a lot. However, comparing with final throughput in Figure 6(b), 16 PE get highest throughput with better throughput. It worth reducing clock frequency to increase final throughput in 16 PE design

As Figure 7(a) shows, with number of entries increasing, the latency increases a lot. From 7 bits (128) to 11 bits(2k), clock frequency reduces nearly to half. And in Figure 7(b), the throughput for 4 PE design is also decreasing by lower clock frequency.

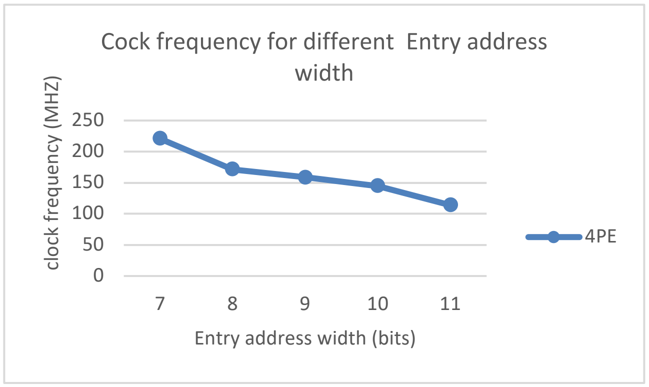

Figure 7(a): clock frequency required versus entry address width 7, 8, 9, 10, 11bits with key width 32 bits, 2 slots, data width 32 bits.

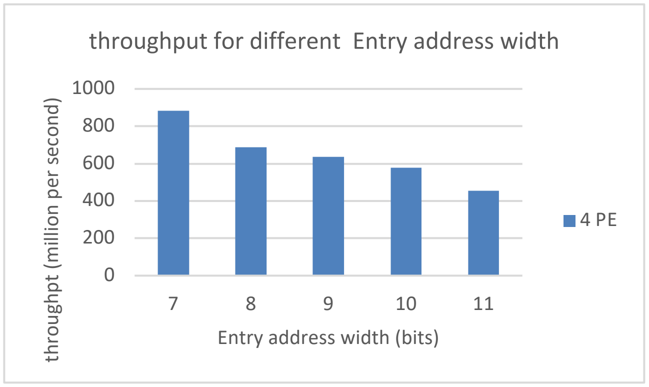

Figure 7(b): throughput versus entry address width 7, 8, 9, 10, 11bits with key width 32 bits, 2 slots, data width 32 bits.

6 Conclusion

For the final result, as I cannot use same FPGA as **FPGA-based High Throughput Parallel Hash Table** [1] uses, but comparing with **FPGA-based High Throughput Parallel Hash Table** [1] implementing support throughput as high as 5360 million operations per second with PEs running at 335 MHz for static hashing, my design's performance decreases.

The new design with hazard control achieves up about 2000 million operations per second for 16 PE.

REFERENCES

[1] Yang, Y., Kuppannagari, S., Srivastava, A., Kannan, R. and Prasanna, V., 2020. FASTHash: FPGA-based High Throughput Parallel Hash Table.

[2] [10] J. L. Carter and M. N. Wegman, "Universal classes of hash functions (extended abstract)," in Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, ser. STOC '77. New York, NY, USA: ACM, 1977, pp. 106–112. [Online]. Available: http: //doi.acm.org/10.1145/800105.803400

[3] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hard- ware hashing functions for high performance computers," Computers, IEEE Transactions on, vol. 46, no. 12, pp. 1378– 1381, Dec 1997.

[4] Tong, D., Zhou, S., Prasanna, V.K.: High-throughput online hash table on fpga. In: 2015 IEEE International Parallel and Distributed Processing Symposium Work- shop. pp. 105–112 (2015)