

APPENDIX A FPGA PROTOTYPE IMPLEMENTATION

For the requirement of high throughput from the big data, BitMatcher can be accelerated by hardware. The calculation on the two buckets in BitMatcher can be split into pipelines so that the throughput is improved. The calculation for the insert operation can be realized in parallel. What's more, multiple engine units can work for the different memory segments to make BitMatcher scalable on hardware implementation.

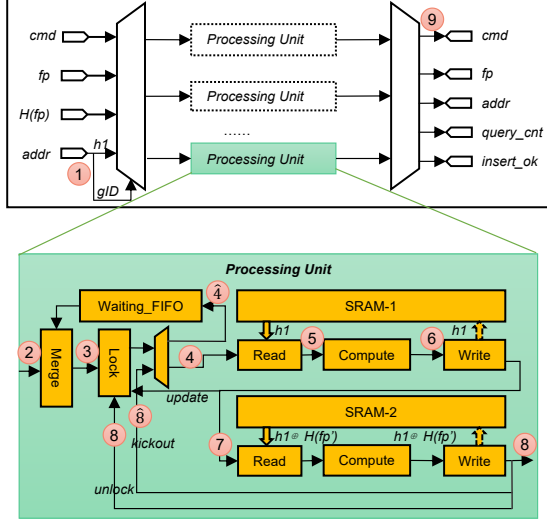


Fig. 23: The FPGA architecture of BitMatcher.

Based on these observations, we design a scalable and high-throughput FPGA implementation for BitMatcher. As depicted in Fig. 23, there are four inputs, i.e. *cmd* which indicates whether the operation is insert or query, *fp* which is the *fingerprint*, $H(fp)$ which is the *hash(item's fingerprint)* and *addr* which is the *hash(e)* in §III.A. *addr* is split to two parts, *h1* and *gID*. *gID* is in the highest address of *addr*, *h1* has the same bit number as $H(fp)$ and there are $2^{\text{sizeof}(gID)}$ processing units. There are 9 steps for an operation as follows.

- 1) The operation is distributed to different processing units based on their values of *gID*.
- 2) The operation gets through the Merge module to aggregate the insert operations with the same *h1* and *fp*. By the module, the back-end computing unit increases the *counter* of the corresponding *entry* not by 1 but a variable *insert_cnt* which Merge module outputs. It can significantly decrease the number of commands to complete and reduce the pressure on the back-end computing unit.
- 3) The merged operation reaches the Lock module. The Lock module is used to avoid the conflict of the operations on the same *entry*. If there is an operation processing and the new input has the same *h1* value or $h1 \oplus H(fp)$ value, the input will be not allowed into the back-end computation.
- 4) If there is no conflict detected in the Lock module, the bucket value will be read from SRAM-1 which costs 1 clock cycle (④). If there is conflict, the operation will be pushed into the Waiting_FIFO and try again from the beginning (④).

- 5) After the bucket value is returned, the Compute module calculates the new bucket value according to the inputs. The computation takes three clock cycles.
- 6) If there is a new bucket value from ⑤, the value will be written into the SRAM-1 in the next clock cycle. What's more, the *fp*, $H(fp)$ and *insert_cnt* will be replaced by the stored kickout entry if kickout happens.
- 7) The operation that has finished, i.e. there is a match and no kickout, is moved backward directly without interaction with SRAM-2. On the other hand, for the operation which has not finished after ⑥, it will pass through the processing of the next bucket. The bucket address is calculated by the $h1 \oplus H(fp')$ where *fp'* is the fingerprint of the kickout entry or just $H(fp)$ for no kickout.
- 8) After the processing of two buckets, the operation, that has finished, is output and instructs the Lock module to unlock the corresponding addresses. The other operations are pushed into the SRAM-1 again (⑧), and the case happens only for the kickout from the second bucket to the first.
- 9) The results from all Processing Units are into the final serialized outputs with *query_cnt* indicating the query result and *insert_ok* indicating whether the insert operation is successful. The insert operation can fail because only one kickout is allowed in the whole process.

There are three highlights in the design. The first is scalability. We divide two arrays into several segments and limit the $H(fp)$ to the same bit number as the address *h1* inside one segment. In this way, *h1* and *h2* have the same *gID* value and the entry in one segment will never need to visit another segment memory. As a result, the Processing Units do not disturb each other and operate independently. Besides, the Merge module aggregates the nearby operations, which improves the throughput capacity too.

Second, the read/write conflict is solved by our Lock module. Two entries, e_1 and e_2 , are stored in the same bucket $A_1[h_1(e_1)]$ with $h_1(e_1) = h_1(e_2)$; and we have two adjacent operations, cmd_1 and cmd_2 , to increase their counters respectively. For cmd_1 , the original value of the bucket with $v^0 = \{e_1.cnt, e_2.cnt\}$ is read from memory and a new value with $v^1 = \{e_1.cnt', e_2.cnt\}$ is computed but has not been written back. Without the lock design, it is v^0 instead of v^1 to be read for cmd_2 and the new value is $\{e_1.cnt, e_2.cnt'\}$, instead of $\{e_1.cnt', e_2.cnt'\}$, to be final stored content. In other words, the information about cmd_1 is lost. However, with the Lock module, cmd_2 is delayed until cmd_1 finishes so the correct result is stored finally. Besides, the kickout function in BitMatcher can also bring read-write conflicts, e.g. e_1 kickout e_2 from bucket 2 to bucket 1 with $h_2(e_1) = h_2(e_2)$ while e_2 has read bucket 1 and has not read bucket 2, which causes the system believe that there is no match for e_2 by mistake. Because the maxloop is set as 1 for BitMatcher, the kickout entry is either from $h_1(e_1)$ in bucket 1 or from $h_2(e_1)$ in bucket 2. Therefore, the Lock module can also avoid the conflict by locking the address $h_1(e_1)$ and $h_2(e_1)$ respectively. It can also be used to guarantee the sequence of the query and

TABLE II: FPGA resources for BitMatcher with 0.1MB memory.

Name	Slice LUT	Slice Register	BRAM Tile
Computation	3018	932	0
Merge	1886	917	0
Lock	1548	3088	0
BM (Total)	11639	7811	33

insert operations.

Third, the complex computation is implemented in the Compute module within three clocks. It takes full advantage of the parallelism in the calculation and improves the working frequency by pipeline. The pipeline consists of three steps:

- 1) It parses the bucket by the Flag value to obtain all entries' fingerprints and counters.
- 2) It compares the fingerprints with the input fp to judge whether there is a matched entry. Meanwhile, all fingerprints are compared with zero to locate the possible empty entry.
- 3) It calculates the new bucket value for the insert operation as described above and gets the query operation's match result (the entry counter or no match).

For the last cycle, there are three types of value adjustment if the input fp matches one entry in the bucket: 1) increase the corresponding counter if overflow does not happen; 2) change the order of local entries so that all entries are stored without overflow; 3) transition the state of the bucket as Fig. 8 in the original paper. For an insert operation, only one type of adjustment occurs so that three alternative values can be calculated ahead and in parallel, and one is selected according to the condition. It is similar for a new fp without any match.

The FPGA resource consumption is as shown in Table II. BitMatcher costs about 12K Slice LUTs and 8K Registers for one processing unit with 0.1MB buckets. A typical Xilinx Virtex-7 chip, xc7vx690tffg1157-2, has 433K Slice LUTs and 866K Registers, so the resource consumption takes less than 3% of the chip. The developers can adjust the number of processing units and bound memory, based on the requirements. The max frequency of BitMatcher on FPGA is 192MHz, which means that the max throughput is 192M commands per second.