Fast and Robust Hashing for Database Operators

Kaan Kara, Gustavo Alonso
Systems Group, Dept. of Computer Science, ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

***Abstract*—Hashing is an essential part of many database operators, such as joins or aggregation, especially when executed in parallel. Often, database engines resort to using easily computed hash functions like modulo to prevent that hashing becomes a bottleneck. The disadvantage of simple hash functions is that they produce imperfect data distributions, particularly when the data is skewed. Robust hash functions produce balanced distributions but they are computationally expensive. Our purpose in this paper is to break the present trade-off between robustness and performance. We achieve this by showing how to implement robust hash functions suitable for database operators on an FPGA. Our target platform (Intel QuickAssist QPI-FPGA) provides a shared memory architecture between the CPU and the FPGA, enabling database engines to use the hardware hashing without any modifications to their memory layout. Depending on the hash function, we achieve 6.6x improvement over pure software implementations. We also show how to integrate hardware hashing in a hybrid hash table without any acceleration overhead.**

I. Introduction

Current trends in big data analytics with constantly increasing data sizes demand higher computing capacity. Non-cryptographic hashing is a common operation in data analytics, be it for traditional relational databases operators or more advanced analytics like machine learning. Since it is done so frequently, applications often use simple arithmetic hash functions, which can be computed in just a few CPU cycles [1], [2]. Intuitively, simple arithmetic hash functions, like modulo or multiply-shift, are not robust against the characteristics of the input data, namely the key-space [3]. If the input data is skewed or has a specific distribution, the simple hash functions may produce high collision rates. Such a hash table will have values that are not distributed uniquely enough to achieve O(1) insertion or look-up cost.

Traditionally, FPGAs have been used as external accelerators connected to CPU sockets via PCI. This kind of architecture limits the acceleration capabilities of the FPGA, since PCI is a high latency bus, which also does not provide good random-access capabilities to main memory. Therefore, the established consensus is to utilize acceleration only for compute-intensive operations on large amounts of data; otherwise the overhead introduced by moving data to an external processor does not pay off [4]. Recently, there has been a lot of interest and development in heterogeneous architectures combining multi-core CPUs and FPGAs. Platforms such as Intel QuickAssist QPI-FGPA [5], IBM Netezza [6], CAPI [7], Xilinx Zynq [8] are available. The purpose of these platforms is mainly to bring the FPGA closer to the multi-core CPU and treat it as a specialized processor. Such heterogeneous architectures put the FPGA as a first-class citizen in the platform and give it cache-coherent access to main memory. This type of architecture enables true shared memory, hybrid hardware-software applications. The applications can utilize acceleration justifiably even for small data sizes and moderately compute-intensive operations with an overhead for data access by the FPGA comparable to that experienced by a CPU.

In this paper we explore the implementation of two robust hash functions (murmur hashing and simple tabulation hashing) on an FPGA, embedded into a heterogeneous multi-core CPU-FPGA platform. Our experiments show that our hardware hash functions are up to 6.6x faster than their software counterparts. We achieve this speed-up despite being limited by the memory bandwidth available to the FPGA. Our hardware hash functions are fully pipelined and saturate the QPI bandwidth, which means that in future platforms, if the bandwidth between the FPGA and main memory is higher, the speed-up would be proportionally higher. We also use hardware hashing in a hybrid hash table and show how existing software applications can benefit from using hardware functions without changes to their memory layout. The results are especially valuable for in-memory, column-store database operators, where hashing is performed very frequently on 64-bit keys [1], [9] and robust hashing is required to be able to deal with skew.

II. Related Work

FPGAs have been used for accelerating cryptographic hashing extensively [10], [11], [12], [13], [14], which utilize the FPGA as a node in the network path to perform cryptographic hashing on streaming data. There are some hash table implementations on FPGAs [15], [16] focusing on implementing an entire hash table as part of a larger application (a key-value store).

To our knowledge, non-cryptographic hashing, which would only produce hash values to be used by a software application, has not yet been studied as a candidate for FPGA acceleration. The reason is that hashing is only a moderately compute-intensive operation and the overhead introduced by a round-trip to an external accelerator would seem excessive. Yet, a recent analysis [3] showed the importance of selecting the right hash function and the hashing scheme in the context databases. In the analysis provided the trade-off between performance and robustness among hash functions is often mentioned. Our contribution lies in showing that an FPGA implementation can break this trade-off by providing both robustness and performance.

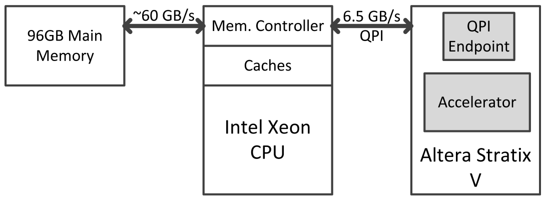

Fig. 1: Intel QuickAssist QPI FPGA Architecture

III. BACKGROUND

A. Heterogeneous Multi-Core Architecture

The target platform in this paper is the Intel QuickAssist QPI-FPGA made available through the *Intel-Altera Heterogeneous Architecture Research Platform*[1][5]. It is a dual socket machine with a 10-core CPU (Intel Xeon E5-2680 v2, 2.8 GHz) on one socket and an FPGA (Altera Stratix V 5SGXEA) on the other. The FPGA is connected to the CPU socket via QPI. It has cache-coherent access to 96 GB of main memory, with 64 B cache line granularity. We measured the QPI bandwidth to be around 6.5 GB/s on combined read and write channels and with equal amount of reads and writes. During runtime, hardware acceleration is made possible to any software application via shared memory and direct writes through QPI. At startup, the application allocates the necessary amount of memory, consisting of 4 MB pages, and transmits the 32-bit physical addresses to the FPGA. The addresses are stored in a page table, implemented as a BRAM. An encrypted QPI end-point IP is provided by Intel, which implements the QPI protocol and local cache (128 KB two-way associative) on the FPGA. Data transmission happens via read and write requests issued by the accelerator to QPI. However, the CPU is also able to issue direct writes through QPI, a useful method for configuring some registers on the accelerator, for example to start/stop an operation.

B. Hash Functions

In our evaluation we use the following 6 non-cryptographic hash functions, which are widely used in database engines and are also included in the extensive analysis in [3]:

1) Modulo: A hash value of n bits is produced by taking the n least-significant bits of a w-bit key.

2) Multiply-Shift: A hash value of n bits is calculated as follows out of a w-bit key, where Z is an odd w-bit integer:

$$hash = (key \cdot Z \, mod \, 2^w)/2^{w-n}$$

3) Murmur: It is a frequently used hash function in practice because of its relatively simple computation and robustness. We use the 64-bit finalizer both in software and hardware implementations [17]:

```
key ^= key >> 33;
key *= 0xff51afd7ed558ccd;
key ^= key >> 33;
key *= 0xc4ceb9fe1a85ec53;
key ^= key >> 33;
```

[1] Following the Intel legal guidelines on publishing performance numbers we want to make the reader aware that results in this publication were generated using preproduction hardware and software, and may not reflect the performance of production or future systems.

TABLE I: Resource Usage

Unit	Logic	BRAM	DSP
Total	34%	5%	50%
QPI Endpoint	30.8%	3.3%	0%
Murmur Hasher	0.9%	0%	50%
SimpleTab Hasher	0.74%	1%	0%

4) Simple Tabulation: It is a very robust hash function, based on random value look-ups. It can be proven that $O(1)$ hash table performance (insertion, deletion, look-up) is achieved when simple tabulation and linear probing are used together [18]. During its calculation a w-bit key is split into $w/8$ characters $c_1, ..., c_{w/8}$. Look-ups are performed at pre-populated (with true random values) tables T_i with c_i as the look-up address. The resulting hash value is obtained by XORing the output values from the tables:

$$hash = \oplus_{i=1}^{w/8} T_i[c_i]$$

5) LookUp3 and City Hash: These are well established hash functions, often used in practice. We use their source code without any changes [19], [20]. They are included in the analysis mainly for comparison purposes.

IV. IMPLEMENTATION

In one read request the accelerator receives a 64 B cache line containing 8 64-bit keys. As a result, parallel hashing with 8 hash function units is possible as depicted in Figure 2. The data is received out-of-order in terms of addressing. To maintain the ordering in writes, we save the addresses of the received cache lines in a FIFO queue and use them to write back the hash values to the same address plus an offset. This results in a memory layout where keys and hashes are known to belong to each other because of their ordering in memory, a common strategy in column store databases. Thus, the accelerator does not need to write the keys back, only the hashes.

Since murmur hashing is basically a series of bitshifts, XORs and multiplications, it is very suitable for a pipeline implementation of few stages. Two 64-bit multiplications are implemented using 16 cascaded DSPs. 8 murmur hashers need a total 128 DSPs, which is 50% of the available DSPs on the target FPGA. Simple tabulation in hardware is implemented using BRAMs as look-up tables populated with true random values. The one time population of tables happens prior to operation. One BRAM holds 256 32-bit values, resulting in 1 KB tables. 8 tables per hashing unit and 8 hashing units in total results in 64 KB of BRAM usage, which is 1% of available BRAM capacity on the target FPGA.

All hardware implementations are done in VHDL. The logic for both hash functions is able to consume and produce a 64 B cache line per clock cycle, utilizing 100% of the available QPI bandwidth. In Table I the total resource usage is depicted when both hash functions are synthesized and loaded onto the FPGA at the same time. Apart from the QPI endpoint and the hash functions, some glue logic is needed for example to save addresses or to select which hash function to use at runtime.

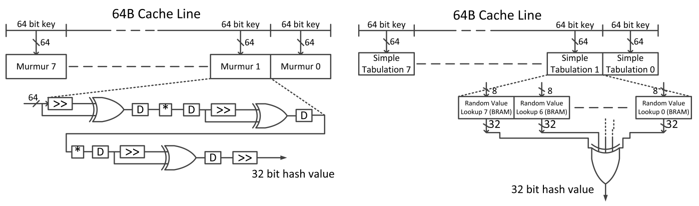

Fig. 2: Murmur and Simple Tabulation Hashing in Hardware

TABLE II: Used Data Distributions

Linear	Random	Grid	Reverse Grid
0x0000_0001	0x2E4F_5929	0x1111_1111	0x1111_1111
0x0000_0002	0x82FA_C7B1	0x1111_1112	0x2111_1111
0x0000_0003	0x186C_BA1F	0x1111_1113	0x3111_1111
...	...	...	...
0x0001_1AF0	...	0x111E_14E1	0x1E41_E111
...	...	...	...

V. Experimental Evaluation

A. Data Distribution

In the experiments we use N 64-bit keys and 4 different key distributions similar to [3], as shown in Table II. In the linear distribution the keys are in the range $[1 : N]$. The keys in the random distribution are generated by the C pseudo-random generator in the range $[1 : 2^{64} - 1]$. In the grid distribution every byte of a 64 bit key takes a value between 1 and 14. They are generated by incrementing the least significant byte until it reaches 14 and then it is reset to 1 and the next least significant byte is incremented. The reversed grid distribution follows the same pattern as the grid distribution; however bytes are incremented starting from the most significant byte. Both grid distributions represent a different type of dense key-space and hashing these kinds of keys in real applications might be necessary at times (such as certain address patterns or strings). Every generated key distribution is shuffled randomly before experiments, so that no ordered data artifacts are produced.

B. Setup and Methodology

In our experiments we use 6 software hash functions (modulo, multiply-shift, murmur, simple tabulation, lookup3 and city hash) and 2 hardware hash functions (murmur and simple tabulation). Regarding hardware hashing performance, the only difference between the two hardware hash functions is the number of clock cycles they take to produce the hash value. This does not affect performance, since all hardware implementations are fully pipelined. Therefore, we observe the same measurements for both hardware hash functions. All performance measurements are performed on the Intel QuickAssist QPI-FPGA platform. Software is written in C++ and compiled with gcc-4.8.2 optimized at -O3. Each measurement is performed with warm caches and with maximum CPU frequency.

C. Experiment 1: Robustness of Hash Functions

In our first experiment we would like to demonstrate the importance of robustness in hashing. To achieve this we generate 1,468,000 keys using the 4 data distributions described above. The keys are hashed using 6 different hash functions in software. With the resulting hash values, 2 hash tables are built using linear probing and bucket chaining schemes, respectively. Out of the produced 32-bit hash values, 21 most-significant bits are used during hash table insertions. Thus, each 64-bit key hashes to a 21-bit hash (2,097,152 unique values), which results in a 70% fill rate for the hash tables. Both hash tables experience a serious performance reduction if the utilized hash function produces values with high collision rates.

TABLE III: Average Probes in Linear Probing

Key-Space/ Hash Func.	Linear	Random	Grid	Reverse Grid
modulo	1	2.1686	306357.65	674958.81
multiply-shift	1	2.1616	1.6417	17.0139
murmur	2.1735	2.1742	2.1677	2.1683
simpletab	2.1584	2.1602	2.2045	2.1369
lookup3	2.1678	2.1599	2.1713	2.1667
city	2.1634	2.1694	2.1661	2.1659

TABLE IV: Avg. and Max. Chain Length in Bucket Chaining

Key-Space/ Hash Func.		Linear	Random	Grid	Rev. Grid
modulo	Avg.	1	1.70108	534.98	498943.73
	Max.	1	7	535	537824
multiply-shift	Avg.	1	1.69997	1.51494	14.02882
	Max.	1	7	5	28
murmur	Avg.	1.70116	1.69966	1.7002	1.70055
	Max.	8	8	8	8
simpletab	Avg.	1.70088	1.70017	1.69647	1.70408
	Max.	8	8	8	8
lookup3	Avg.	1.70081	1.70166	1.70177	1.70073
	Max.	8	8	8	7
city	Avg.	1.70119	1.69951	1.70149	1.69940
	Max.	8	8	8	8

In Table III and IV we observe that simple arithmetic hash functions (modulo, multiply-shift) have a tendency to fail depending on the input data distribution. Although they behave perfectly for linearly distributed keys, modulo produces many colliding hash values for both grid distributions and multiply-shift is also inadequate for reverse grid distribution. On the other hand murmur, simple tabulation, lookup3 and city hash behave pretty much independently of the data distribution.

D. Experiment 2: Performance of Hash Functions

This experiment considers only hashing performance. We perform either software or hardware hashing on keys and measure the total execution time. The same measurement was performed on a varying the number of keys from 2^{20} up to 2^{26} to observe scalability, which resulted in linear increases in execution times. In Figure 3 we observe that simple tabulation hashing is 6.6x and murmur hashing is 1.7x faster in hardware compared to their software counterparts.

E. Experiment 3: Hybrid Hash Table Build

In this experiment a hash table is built using linear probing. Here, two factors affect overall performance: Hash calculation speed and the number of necessary probes to insert the key into

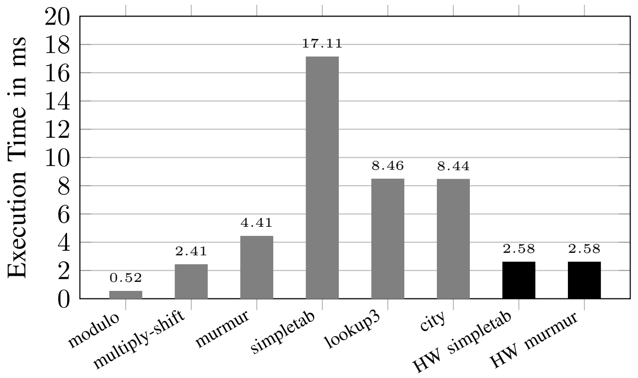

Fig. 3: Hashing Time for 2^{20} Keys

TABLE V: Hash Table Build Time

Key-Space/ Hash Func.	Linear	Random	Grid	Reverse Grid
modulo	7.72 ms	22.59 ms	510.08 s	1123.16 s
multiply-shift	11.88 ms	28.42 ms	**23.40 ms**	60.03 ms
murmur	31.25 ms	32.24 ms	32.29 ms	31.38 ms
simpletab	60.25 ms	61.47 ms	60.69 ms	60.90 ms
lookup3	39.87 ms	40.73 ms	41.37 ms	40.22 ms
city	39.52 ms	41.47 ms	41.08 ms	40.13 ms
HW simpletab	24.26 ms	24.38 ms	**24.23 ms**	24.28 ms
HW murmur	24.21 ms	24.22 ms	**24.26 ms**	24.25 ms

the hash table. The number of necessary probes is less, if the produced hash values collide with low probability. Since the build time consists of both hashing and insertion, it is a good metric to evaluate overall hashing quality. A hash function that is both fast to calculate and robust will result in the shortest build times. Similar to Experiment 1, 1,468,000 keys are generated in 4 distributions and inserted into a 2^{21} keys capacity hash table, resulting in a 70% fill rate. The results in Table V again show the lack of robustness in modulo and multiply-shift, leading to increased build times due to high collision rates, despite being fast in pure hash calculation.

In case of hardware hashing, the CPU and the FPGA collaborate and utilize the shared memory as follows: As soon as the FPGA writes a cache line containing the hashes to memory, the CPU iterates over the hashes in that cache line and inserts the keys into the hash table. Thus, the FPGA can be regarded as the data producer and the CPU as the data consumer, both operating in a pipelined fashion, one cache-line at a time instead of in large batches. This hybrid execution enables hardware acceleration without any visible overhead and hides data transfer latencies over QPI to the FPGA.

We can observe the absence of visible overhead in hardware acceleration best when we compare the build times using hardware hashing and multiply-shift hashing for grid distribution in Table V (in bold). We conclude this after the following analysis: From the pure hashing performance measurements in Figure 3 we know that multiply-shift in software and hashing in hardware have almost the same performance. Furthermore, we observed in the robustness measurements that multiply-shift behaves even slightly better than both simple tabulation and murmur for grid distribution. Keeping these two results in mind and observing the same build times in Experiment 3 leads us to conclude that using an FPGA to implement a more expensive hash function leads to the same results as using simple hash functions in software. Yet, the resulting system is far more robust and capable of supporting different data distributions than existing solutions, thereby showing that the FPGA can be used not only to attain better performance but also better quality of results.

VI. Conclusion

We implemented robust hash functions in a heterogeneous multi-core CPU-FPGA platform with shared memory and showed that these hash functions achieve up to 6.6x speed-up while providing increased robustness to skewed data distributions. We also utilized hardware hashing in a main memory hash table and achieved the fastest average build times among all hash functions and data distributions, as a result of fast and robust hashing. In future work we plan to embed the hybrid hash table in database operators such as hash join and aggregation.

Acknowledgment

We would like to thank Intel for the generous donation of the hardware platform as part of the Hardware Accelerator Research Program.

References

[1] C. Balkesen, J. Teubner, G. Alonso, and M. T. Ozsu, "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," in *ICDE*, 2013.
[2] R. Barber, G. Lohman *et al.*, "Memory-efficient hash joins," 2014.
[3] S. Richter, V. Alvarez, and J. Dittrich, "A seven-dimensional analysis of hashing methods and its implications on query processing," 2015.
[4] M. Jacobsen *et al.*, "Riffa 2.1: A reusable integration framework for fpga accelerators," *TRETS*, 2015.
[5] N. Oliver, R. R. Sharma *et al.*, "A reconfigurable computing system based on a cache-coherent fabric," in *ReConFig*, 2011.
[6] M. Singh and B. Leonhardi, "Introduction to the ibm netezza warehouse appliance," in *CASCON*, 2011.
[7] J. Stuecheli *et al.*, "Capi: A coherent accelerator processor interface," *IBM Journal of Research and Development*, 2015.
[8] L. H. Crockett *et al.*, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*, 2014.
[9] S. Blanas *et al.*, "Design and evaluation of main memory hash join algorithms for multi-core cpus," in *SIGMOD*, 2011.
[10] J. Deepakumara *et al.*, "Fpga implementation of md5 hash algorithm," in *CCECE*, 2001.
[11] R. P. McEvoy *et al.*, "Optimisation of the sha-2 family of hash functions on fpgas," in *ISVLSI*, 2006.
[12] N. Sklavos and O. Koufopavlou, "Implementation of the sha-2 hash family standard using fpgas," *The Journal of Supercomputing*, 2005.
[13] N. Pramstaller *et al.*, "A compact fpga implementation of the hash function whirlpool," in *FPGA*, 2006.
[14] N. Sklavos and P. Kitsos, "Blake hash function family on fpga: From the fastest to the smallest," in *ISVLSI*, 2010.
[15] Z. István, G. Alonso, M. Blott, and K. Vissers, "A flexible hash table design for 10gbps key-value stores on fpgas," in *FPL*, 2013.
[16] Z. Istvan, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a box: Inexpensive coordination in hardware," in *NSDI*, 2016.
[17] A. Appleby, "https://github.com/aappleby/smhasher," january 2016.
[18] M. Patraşcu and M. Thorup, "The power of simple tabulation hashing," *JACM*, 2012.
[19] B. Jenkins, "http://burtleburtle.net/bob/c/," may 2006.
[20] Google, "https://github.com/google/cityhash," august 2013.