Accelerating Hash-Based Query Processing Operations on FPGAs by a Hash Table Caching Technique

Behzad Salami[1,2], Oriol Arcas-Abella[1], Nehir Sonmez[1], Osman Unsal[1], and Adrian Cristal Kestelman[1,2,3]

[1]Barcelona Supercomputing Center (BSC), [2]Universitat Polytecnica de Catalunya (UPC),
[3]IIA- Artificial Intelligence Research Institute- Spanish National Research Council
{behzad.salami, oriol.arcas, nehir.sonmez, osman.unsal, adrian.cristal}@bsc.es

Abstract. Extracting valuable information from the rapidly growing field of Big Data faces serious performance constraints, especially in the software-based database management systems (DBMS). In a query processing system, hash-based computational primitives such as the hash join and the group-by are the most time-consuming operations, as they frequently need to access the hash table on the high-latency off-chip memories and also to traverse whole the table. Subsequently, the hash collision is an inherent issue related to the hash tables, which can adversely degrade the overall performance.

In order to alleviate this problem, in this paper, we present a novel pure hardware-based hash engine, implemented on the FPGA. In order to mitigate the high memory access latencies and also to faster resolve the hash collisions, we follow a novel design point. It is based on caching the hash table entries in the fast on-chip Block-RAMs of FPGA. Faster accesses to the correspondent hash table entries from the cache can lead to an improved overall performance.

We evaluated the proposed approach by running hash-based table join and group-by operations of 5 TPC-H benchmark queries. The results show 2.9X – 4.4X speedups over the cache-less FPGA-based baseline.

1 Introduction

In the era of Internet of Things (IOT) and Big Data, fast query processing is a crucial requirement of the modern DBMS. In an attempt to move the computation closer to the storage, many previous studies have looked into accelerating database operations in the hardware platforms. Examples include employing vector architectures [9], ASICs [18], GPUs [10], or hybrid [25]. Other approaches either used FPGAs statically [15], [8], [3], [17], [26], or leveraged dynamic reconfigurability characteristic of FPGAs to better fit the requirements of the queries [6], [13]. The industry has also invested in products such as IBM Netezza [2] and Teradata Kickfire [14].

The hash-based operations, i.e. hash join and group-by are the most time-consuming operations of databases query processing systems. Previous studies have demonstrated that these operations account for more than 40% of total execution time while running queries from the TPC-H benchmark [9].

The hash join operation combines two data tables *S* and *T* together with a common *key*. The algorithm consists of ***i)*** a build phase to construct a hash table using the rows of the table *S*, and ***ii)*** a probe phase, where all keys in the table *T* are looked into the hash table to find whole the possible matches. Similarly, the group-by operation groups the rows of a given table based on common values of the *key* column, which can also be implemented using hash tables. The main issue that can degrade the performance of a hash engine is the hash collision, which is the situation of mapping two distinct keys into the same hash index. By design and in practice, these cases are inevitable for database applications and need to be handled appropriately. Among the possible solutions, software fallback mechanisms [17] or rehashing [7] can cause additional latencies that reduce the performance. On the other hand, collision resolution in the hardware implies chaining the hash table entries that can also undermine the hash table performance, especially under DDR memory latencies.

Due to the scarcity of on-chip BRAM resources that cannot guarantee to locate the entire hash table, previous FPGA implementations envisioned building the hash table in the off-chip DDR memory [17], [20]. Alternatively, in this work we propose a hash table caching technique, exploiting the on-chip BRAMs of FPGAs to mitigate the memory latencies. Also, our design resolves hash collisions without reverting to software fallbacks. For the evaluations, we run the hash join and group-by operations of 5 queries of the TPC-H benchmark suite and demonstrate up to 4.4X performance speedups, compared to a hardware baseline that does not employ any caching technique. The hardware baseline is an improved version of Ibex [17]. Despite Ibex that uses software fallbacks to resolve the hash collisions, in our baseline, we follow a pure hardware-based pointer chasing method.

In a nutshell, trading off the size and the latency of on/off-chip memories, we ***(i)*** can support large datasets using a hash table located in the off-chip memory, and ***(ii)*** avoid the high memory latencies by utilizing the on-chip BRAMs of FPGAs as the hash table cache. The contributions of this paper can be summarized as below:

- We propose a hash table caching mechanism that efficiently exploits the on-chip BRAMs of FPGA to serve some of the hash table inquiries. This method can be significantly faster than the conventional way to retrieve the hash table entries from the off-chip memories.
- We investigate the proposed technique for the hash-based operations of query processing systems, i.e. hash join and group-by. Hash collisions are resolved purely in the hardware, which taking advantage of the hash table caching method. We design the proposed method by leveraging Bluespec, a high-level synthesis (HLS) tool. The design is implemented on a Virtex 7 FPGA development board (VC709). We achieved up to 4.4X speedup, compared to the hardware baseline.

The rest of the paper is organized as follows: Section 2 includes the background information, as well as an illustration of the hash table caching technique. The proposed architecture is elaborated in Section 3. Section 4 introduces the evaluation methodology. Section 5 includes discussions the experimental results. In Section 6 we review related work and finally, Section 7 concludes the paper.

2 Background

Conventionally, data can be organized in either structured or unstructured management systems. Although, the proposed hash table caching technique can be customized in both the systems, our focus will be on the relational DBMS, as a common type of structured data management systems.

In an RDBMS, data is organized into tables using a model of vertical columns and horizontal rows. The rows represent entries in the database and columns define the data types. Data in the tables are formed as a pair of (*key*, *value*), where *key* points to one of the columns that play the main role in the query analysis such as sort key, hash join key, etc. Other columns are merged into the *value*. In order to access the data into the tables, query languages such as Scripting Query Language (SQL) have been introduced. In a typical SQL query, several language elements such as **SELECT**, **GROUPBY**, **ORDERBY**, etc.**,** can exist. These operations can be semantically mapped to specialized hardware accelerators such as filtering, aggregation, hash join, sorting, etc. The hash-based operations, i.e. hash join and group-by are considered in this work because they are the most time-consuming DBMS operations.

2.1 Hash Join Background

One common type of join operation is the equijoin or θ-join. It means combining rows from two or more tables with a common cell. The hash-based join or hash join is the most common type of table join algorithms.

The objective in the hash join is to reduce the search space using a hash function over the common cell, or *key*. It consists of build and probe phases. In the build phase, the hash table is constructed using the input table (*S*). In this phase, for each tuple (k_s, v_s) a hash index is calculated using a hash function and correspondingly, a hash table entry is created in that given index of the hash table. In the probe phase, the hash table is being scanned in the hash index. The corresponding hash index is generated by the hash function applied on the each input tuple (k_T, v_T) of data table (*T*). If any match is found the resulting 3-tuple (k, v_S, v_T) is output, where $k = k_S = k_T$. Otherwise, it means that the current input tuple does not exist in the hash table and it is skipped. It is worth noting that as the hash table construction is more costly operation than the probing of the hash table, the smaller input table is used in the build phase (|S| < |T|).

2.2 Group-by Background

Group-by is another query processing operation that can be implemented using the hash tables, as well. It is usually used in conjunction with an aggregation function to produce the aggregation of the rows in the same group, called group-by aggregation [24]. For a given table S with rows (k_{S1}, v_{S1}) and (k_{S2}, v_{S2}), the group-by and group-by aggregation operations will produce tuples with (k, v_{S1}, v_{S2}) and $(k, aggrFunc(v_{S1}, v_{S2}))$ fields, respectively $(k = k_{S1} = k_{S2})$**.** The aggregation function can be **SUM**, **AVERAGE**, **MAX**, **COUNT**, etc. It is worth noting that constructing the hash table on *key* consists of adding the grouped data into the hash table. Another word, data in the hash table are already grouped.

2.3 Collision in the Hash-Based Operations including Hash Join and Group-by

In practice, in the hash-based query processing operations an ideal hash function to generate a unique hash index for every input data tuple scarcely exist. Thus hash collisions inevitably happen, particularly for DBMS applications, and need to be appropriately handled. In order to resolve this issue, various mechanisms on FPGAs are proposed. Software fallback mechanisms [17] facilitate the hardware design. However, it may cause additional latencies due to the transfer time between the FPGA and the software. Rehashing [7] is another method, which could also cause extra overheads due to additional rehashing costs. On the other hand, supporting collision management in the hardware implies chaining the hash table entries. It means that the next address to be jumped to can only be determined after the previous line is read. Under DDR latencies it can adversely diminish the overall performance.

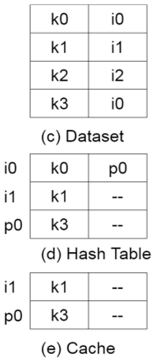
2.4 Illustrating the Hash Table Caching

The data/instruction caching is a widely used optimization mechanism to cover the speed gap between the storage and the processor. This paper is motivated by the fact that caching can also be employed to improve the performance of the hash-based operations of the query processing systems. As far as we know, this is the first work to design a hash join/group-by engine equipped with a caching mechanism.

For convenience, we illustrate the proposed technique using an example in the probe phase of the hash join operation to show how does this operation can take advantage of the hash table caching technique? The data tables that include the input dataset for probing, the hash table, and the contents of the cache are shown in Fig 1.c, 1.d, and 1.e, respectively. The hash table and cache are already filled in the build phase. The cache has the corresponding hash indexes of only *k1* and *k3*.

Step	Cycle	Operation	Step	Cycle	Operation
0	0	lookupHT i0	0	0	lookupC i0
1	1	lookupHT i1	1	1	lookupC i1, respC i0, missC, lookupHT i0
2	2	lookupHT i2	2	2	lookupC i2, respC i1, hitC, match k1
3	3	lookupHT i0	3	3	lookupC i0, respC i2, missC, lookupHT i2
4	35	respHT i0, match k0	4	4	respC, missC, lookupHT i0
5	36	respHT i1, match k1	5	31	respHT i0, match k0
6	37	respHT i2, mismatch k2	6	32	respHT i2, mismatch k2
7	38	respHT i3, collision k3, lookupHT p0	7	33	respHT i0, collision k3, lookupC p0
8	**69**	respHT i0, match k3	8	**34**	respC i0, hitC, match k3

(a) (b)



k0	i0
k1	i1
k2	i2
k3	i0

(c) Dataset

i0	k0	p0
i1	k1	--
p0	k3	--

(d) Hash Table

i1	k1	--
p0	k3	--

(e) Cache

Fig. 1: An example hash probe, baseline (a) vs. cache (b). Example dataset (c), the content of hash table (d) and the cache (e).

The dataset that needs to be probed in the hash table is shown in Fig 1c, with four keys and their corresponding hash indexes. The hash collision requires scanning a pointer chain from *i0* to *p0*. As it can be seen in this table, there is a hash collision for *k0* and *k3*, both having the same hash index *i0*. There are totally two directly matched key, one matched key after a collision, and one mismatched key.

In this example, the latency of the cache in the on-chip BRAM and the hash table in the off-chip DDR are assumed to be 1 and 30 cycles, respectively. The cycle-by-cycle execution of the cache-less baseline and cache-based hash probe are depicted in Fig 1.a, and 1.b, respectively. Several terms are used to describe the example clearer: *lookup* (to send read request for the hash table –HT or the cache -C), *resp* (to get response from the hash table –HT or the cache -C), *(mis-)match* (to show that an input key is (mis-)matched from the hash table or from the cache), *collision* (to show a detected hash collision), and *hit/miss* (to show a cache hit/miss).

As described in Fig. 1a, in the baseline execution, all the accesses are served from the hash table in DDR (*lookupHT*). The responses arrive 30 cycles later (*respHT*). In contrast, as it can be seen in Fig. 1b, in the cache-based version, all the inquiries are being looked up from the cache, first (*lookupC*). The successful requests (cache hit-*respC*) are being processed in the probe engine, and the unsuccessful (missed) ones are being forwarded to the hash table (*lookupHT*). Serving some of the requests from the cache reduces the total cycles to probe the example dataset from 69 to 34.

In this example, we showed both the cache hit and miss scenarios, to demonstrate the efficiency of the hit requests against the overhead of missed cache inquiries. However, in the real datasets other events such as a chain of colliding keys, redundancy chaining, the irregular latency of DDR, the complexity of the write requests in the build phase, etc., may appear.

3 The Overall Architecture of the Proposed Engine

The overall layout of the proposed accelerator is shown in Fig. 2. The connection of FPGA with the host and the off-chip DDR-3 is through the high-speed PCI-3 and DDR-3 interfaces, respectively. The host initializes DDR-3 with the input data tables. DDR-3 memory locates the hash table, as well. FPGA is comprised of several components: ***i)*** device drivers to manage the off-chip data transfer, ***ii)*** a central controller to manage the computations and data movements, ***iii)*** the accelerator engine (hash join and group by), and ***iv)*** finally, the on-chip Block RAMs, which are configured as the cache of the hash table.

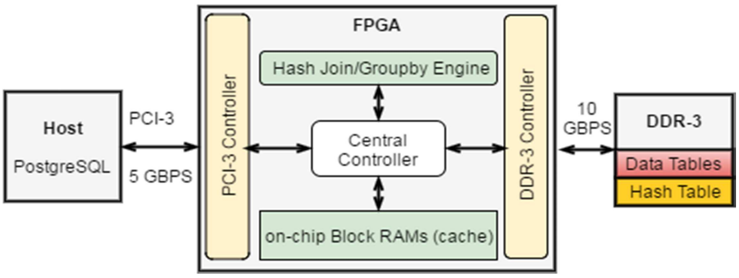


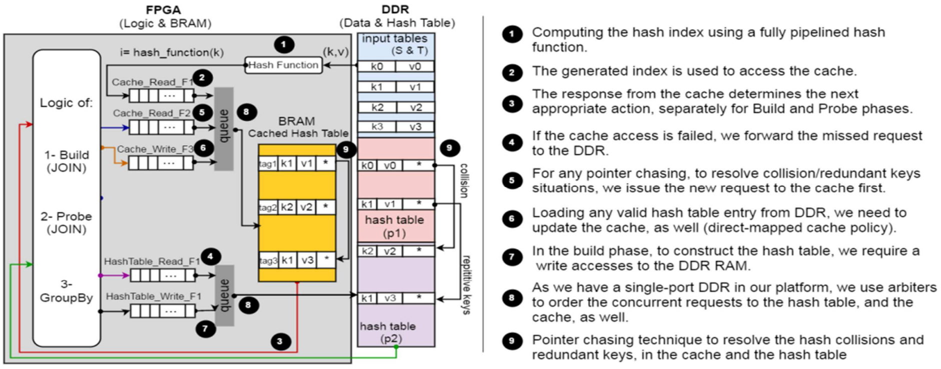
Fig. 2: The overall layout of the accelerator including Host, FPGA, and DDR-3.

The detailed structure of the accelerator is shown in Fig. 3. Its overall architecture is comprised of several components: ***(i)*** *a (Linear Feedback Shift Register) LFSR-based hash function:* It generates the hash index of the input *key* in a fully pipelined fashion. The generated hash indexes are used as the index of the corresponding hash table/cache entries. ***(ii)*** *The logic of the accelerator, i.e. hash join build, hash join probe, and group-by:* As a part of their functionality, the hash collisions of the colliding keys are resolved by chained together in a linked list fashion. The similar method is used to organize the repetitive keys in the hash table. ***(iii)*** *The hash table in the off-chip DDR-3:* In order to efficiently support pointer chasing in the aforementioned special cases, we partitioned the hash table into two distinct parts. The first half part of the hash table can be directly indexed by the hash function in normal cases. The second half part, which is excluded from the range of the hash function, is used for only the chains of the entries. This part of the memory is consecutively being accessed. ***(iv)*** *A cache of the hash table in the on-chip BRAMs:* The entries of the cache are exact copies of some of the hash table entries. The hash table inquiries will be served from the cache. Only the missed requests from the cache will be forwarded to the hash table.

In order to support the aforementioned features, each entry of the cache/hash table has several fields, including:

- ***valid*** bit to show the validity of the entry.
- ***key*** field to store the input data *keys*.
- ***value*** field to store the *value* of the input data.
- ***pointer$_c$*** that is used to resolve the hash collisions by storing the index of an allocated hash table entry, following the pointer chaining mechanism.
- ***pointer$_r$*** that is used to manage the repetitive keys in the hash table. Similar to the hash collision, it uses the pointer chaining mechanism. The exception is the group-by aggregation, where instead of storing key itself, we compute an aggregation of the keys. Thus, there is no need for pointer chasing in this particular case.
- ***cache tag*** field to discard false positives in the cache.

Consequently, having any successful inquiries from the cache correspond to skip of the hash table accesses in the off-chip DDR-3 memory. In addition, similar to Ibex [17], we use a Content Addressable Memory (CAM) to remove read-after-write hazards.



1. Computing the hash index using a fully pipelined hash function.
2. The generated index is used to access the cache.
3. The response from the cache determines the next appropriate action, separately for Build and Probe phases.
4. If the cache access is failed, we forward the missed request to the DDR.
5. For any pointer chasing, to resolve collision/redundant keys situations, we issue the new request to the cache first.
6. Loading any valid hash table entry from DDR, we need to update the cache, as well (direct-mapped cache policy).
7. In the build phase, to construct the hash table, we require a write accesses to the DDR RAM.
8. As we have a single-port DDR in our platform, we use arbiters to order the concurrent requests to the hash table, and the cache, as well.
9. Pointer chasing technique to resolve the hash collisions and redundant keys, in the cache and the hash table

Fig. 3: The detailed architecture of the proposed engine (hash table caching).

3.1 Hash Join: Build Phase- Constructing the Hash Table

In order to insert a new (k_s, v_s) pair into the hash table, first, a hash index of k_s is generated by the hash function. This index points to the corresponding index in the hash table/cache. We use an LFSR-based hash function to generate pseudo-random hash indexes. Later on, the content of the corresponding entry of the cache is retrieved. Due to the retrieved entry, ***(i)*** if it is not valid or is an undesirable (false positive) entry, a cache miss occurs. The false positive situations of the cache can be recognized by checking the cache tag. In these situations, we forward the same inquiry to the hash table. Or, ***(ii)*** if the cache hits, or we get the corresponding entry from the hash table, three different cases can occur:

- If the retrieved entry is not valid, a new entry is added to the corresponding index of both the hash table and the cache.
- If the accessed entry is valid, with the same $\mathrm{k_s}$, it needs to allocate a new entry and appropriately update the pointer fields, to manage repetitive keys in a linked-list fashion. Accordingly, the hash table and cache are updated.
- If the accessed entry is valid, but with a different $\mathrm{k_s}$, a hash collision occurs. Similar to the case of repetitive keys, a new hash table entry is allocated. Both the new and old entries are updated in the cache and the hash table, to preserve the linked-list behavior. Following this chaining method, nested hash collision can be resolved, as well.

Our engine can deal with an unlimited number of hash collisions/repetitive keys, as long as the hash table is not full.

3.2 Hash Join: Probe Phase- Scanning the Hash Table

In order to scan the hash table, first, we compute the hash index for the new k_T. Later on, retrieving the corresponding index from the cache, ***(i)*** if it is not a valid entry or is not the desired entry (false positive), thus, a cache miss occurs. Therefore, the same inquiry is forwarded to the hash table. And, ***(ii)*** if the cache hits or the response from the hash table arrives, three cases can occur:

- If keys do not match and there is no valid collision $pointer_c$ field in the retrieved entry, there is no entry in the hash table which matches with $\mathrm{k_T}$.
- If keys do not match, but there is a valid collision $pointer_c$ field in the retrieved entry, a hash collision occurs. Therefore, we first scan the subsequent hash table entries, retrieving them from the cache, first. This process may lead to a mismatch, if and only if no match can be found until the end of the chain. Nevertheless, at any point of the chain, it is possible to find a match.
- If keys match, their combination will produce a junction row. This match can be found directly, or after a pointer chasing process. Accordingly, all the $\mathrm{v_s}$ in the chain must orderly be read to generate the tuples of the junction table, $(\mathrm{k}, \mathrm{v_S}, \mathrm{v_T})$ that $\mathrm{k} = \mathrm{k_S} = \mathrm{k_T}$.

In the probe phase, the cache is updated by each valid response from the hash table.

3.3 Group-By Aggregation: Constructing a Hash Table to Group Data

Group-by operation intrinsically is similar to the build phase of the hash join operation, as data in the hash table are already grouped based on the *key* field. The main difference is that ***(i)*** usually in the SQL queries the group-by operation is accompanied by an aggregation function, such as **SUM**, MAX, **COUNT**, **AVERAGE**, etc. Consequently, instead of storing *key* itself in the hash table, an aggregation of the *key* needs to be stored, without any necessity for pointer chasing to manage the repetitive keys. ***(ii)*** As the number of groups is usually quite smaller than the size of the input dataset, there are often accesses to the same hash table entries. This can significantly take advantage of the hash table caching technique, as the repetitive accesses can be served from the cache.

In order to perform a group-by operation, similar steps to the hash join build phase are followed, except the step 2, where the input *key* is matched with an entry in the hash table/cache. In this particular case, we perform the aggregation on the *value* field and skip allocation a new hash table entry to store the *key* field.

3.4 Policies of the Cache

Various accessing methods to the cache and its different Read/Write policies can impact the performance. The cache policies in the proposed technique are as below:

- *Cache Contents:* The cache contains a number of the recently accessed valid entries of the hash table. Each cache entry is an exact copy of the corresponding entry in the hash table.
- *Cache Replacement Policy:* The hash table in the off-chip DDR memory is significantly larger than the cache. Thus, a replacement policy is required to substitute the new with the old entries of the cache. We use a direct-mapped policy, where all the valid retrieved entries from the hash table are overwritten into the cache.
- *Cache access policy:* For all the required hash table entries, first, we look up the cache. Any cache hit leads to skipping the DDR-3 accesses, but in contrast, the missed requests need to be forwarded to the hash table. In order to discard false positives, the cache has an additional field, the *cache tag*.
- *Cache Indexing:* We use the Least-Significant Bits (LSB) of the hash table index as the cache index. The Most-Significant Bits (MSB) are stored as the *cache tag* field.

4 EXPERIMENTAL METHODOLOGY

We used Xilinx ISE version 14.1 and Bluespec System Verilog compiler [1] in the development phase. Bluespec is a commonly-used cycle-accurate modern HLS tool, desired for control-oriented designs such as hash join. Our system was designed to work at 200 MHz on a VC709 development board with a Virtex-7 FPGA and a 4 GB

DDR-3 memory channel. Our device has about 50 MBit on-chip BRAMs that are employed as the cache. The PCI-3 controller works at 150 Mhz. Thus, the synchronizing FIFOs are exploited to exchange data among different clock domains properly. We have made all our modules fully parametrizable. We validated the experimental result by checking with the software (PostgreSQL [22]) runs of the same DBMS operations. (*key, value*) pairs are 64 bits, each of which is 32-bits.

4.1 Hardware and Software Comparison Baselines

In the hardware baseline, only DDR-3 RAM is exploited to store the hash table, without any caching mechanism. Many FPGA implementations follow the similar design point. For instance, recently Ibex [15] is presented that uses the DDR-3 to locate the hash table but unlike our baseline, it falls back software for the hash collisions. Thus, our hardware baseline is efficient, cache-less, and pure hardware FPGA-based implementation of the corresponding operations, i.e. the hash join and the group-by.

The second comparison case is a state-of-the-art software-based DBMS (PostgreSQL) that is running in the warm cache setup on a server with 64 GB RAM and a Xeon E5-2630 CPU. PostgreSQL does not support multi-threading. Thus, we use the single-thread execution times of the queries for the comparisons. In order to get the warm execution time, we run PostgreSQL two consecutive times. The second run is supposed to be from its internal buffers, where data tables are already located into the system memory. There are no disk I/O transactions in the warm cache mode of the software runs.

It is worth noting that the execution model in the software baseline is different with the FPGA-based solutions, including the proposed cache-based method and the hardware baseline. We follow a dataflow execution model in the FPGA-based accelerators, which allows deep pipelining and data streaming capabilities to achieve the peak performance. In contrast, PostgreSQL runs on the scalar processor with a control-flow execution model, which suffers from its conventional implications.

4.2. The Structure of the Benchmarks

In order to evaluate the proposed engine, we run a set of complex queries from the TPC-H benchmark suite [23]. Specifically, we selected Q03, Q04, Q12, Q13, and Q14, because they have different table sizes and also different join selectivity (the size of the output data table divided by Cartesian product of the two input tables). However, as the given queries are composed of several other operations, such as sorting, aggregation, etc., we made a sub-query to extract only their hash join and group-by part.

Furthermore, some of the queries such as Q03 are composed of multiple hash-based operations. For these cases, we extract different sub-queries for each hash join/group-by operation, run them separately, and get their distinct execution times. Later on, in order to compute the total execution time of the given query, we sum up all those separate parts.

Table. 1: The general format of the sub-queries used in the benchmarks.

hash join		group-by	
SELECT	v_S, v_T	**SELECT**	**SUM**(v_S)
FROM	S, T	**FROM**	S
WHERE	$k_S = k_T$	**GROUPBY**	k_S

The general format of the generated subqueries is shown in Table. 1, separately for the hash join and the group-by operations. We assumed two data tables *S* and *T* with data tuples *(k_S, v_S)* and *(k_T, v_T)*, respectively. In addition, we used various sizes of data tables in the experiments, including 1GB and 10GB scales. We repeat the query runs 10 times. The reported total execution time of each given query is the average of the execution times of its various runs.

5 EXPERIMENTAL RESULTS

In this section, we evaluate the proposed cache-based engine for the hash join and group-by operations. Due to the size of the each entry of the cache and also the size of the available BRAMs in our device, the cache can cope with about 256K entries. Thus, in 1GB scale, we observed that BRAMs could entirely store the corresponding hash tables without any need for accessing the off-chip DDR memory. Followingly, in 1GB scale, we exploited the on-chip BRAMs as the hash table (not as the cache). In contrast, for 10GB scale, as the sizes of hash tables are larger than the BRAMs, we follow the proposed hash table caching technique.

5.1 Analyzing the Hash Table Caching

Table. 2 includes the experimental results for 10GB scale. In this table, *table size* refers to the number of the rows (key-value pairs) of the input data tables. The total *number of the collisions* is also shown in this table. Another important parameter is the *number of lookups* for the cache and for the hash table, as well. The *hit ratio (H.R)* of the cache that is an important metric to determine the performance achievement can be computed as the equation. 1:

$$H.R = \frac{\#cach\_lookup\ - \#ht\_lookup}{\#cache\_lookup} \qquad (1)$$

For convenience, we describe a sample result of Table. 2, probe phase of Q03. For this particular case, we observed that ***i)*** totally 44.2M cache read requests are issued; 32.1M read requests of the original dataset and 12.1M additional requests (37.6% of the table size) for pointer chasing cases that 8.4M of them are as the result of hash collisions and the rest 3.7M as the result of repetitive keys. Furthermore, ***ii)*** 33.4% of cache read requests are successfully served from the cache and the rest are forwarded to the hash table. Thus, the *H.R* is 33.4%.

Table 2: The experimental results of hash table caching, 10 GB scale.

Query	Operation	table size (M)	#cache_lookup (M)	#ht_lookup (M)	Collision (M)	H.R (%)
Q03	Build	1.4	1.7	1.49	0.3	12.3
	Probe	32.1	44.2	29.4	8.4	33.4
	Groupby	0.3	0.35	0.05	0.02	85
Q04	Build	0.56	0.69	0.56	0.13	18.8
	Probe	37.2	57.1	41.2	13.2	27.8
	Groupby	0.52	0.52	~ 0	0	100
Q12	Build	0.3	0.35	0.3	0.05	14.2
	Probe	15	16.2	15	1.2	7.8
	Groupby	0.31	0.31	~ 0	0	100
Q13	Build	1.5	1.8	1.56	0.3	13.3
	Probe	14.8	19.5	11.9	2.6	38.9
	Groupby	1.5	1.5	~ 0	0	100
Q14	Build	0.7	0.78	0.7	0.08	10.2
	Probe	2	2.2	2	0.2	9

The experimental results show that the *H.R* is on average 34.75%. It ranges from 7.8% to 100%. More specifically, about the hash join cases, we observed that:

- The average *H.R* in the build phase of the given queries is 13.7% that is significantly less that the total average *H.R* (34.75%). For all of the studied queries, the hash join *key* in the build phase is a primary (no repetitive) key. Thus, the cache is not efficiently utilized as a consequence of the less data locality in the hash table accesses, for this case.
- Probe keys of Q12 and Q14 are the primary keys, as well. Thus, we observed less *H.R* for these queries compared to others (8.4% vs. 24.8%).
- Input data tables in the probe phase are significantly larger than in the build phase, on average 30X. Thus, although, the hash table construction in the build phase is a more expensive operation, we observed that the execution time of the probe phase is dominant.

Although, the cache misses incur additional overheads, the substantial improvement of the cache hits, in terms of mitigating the latency of the memory, covers its side effects and leads to better performance compared to the cache-less hardware baseline.

In addition, most of the studied queries, except Q14, are composed of a group-by aggregation operation. For instance, in 10GB scale of Q03, 300K tuples are grouped into about 100K individual groups, or 520K tuples of Q04 are grouped into only 5 groups. The experimental results in Table. 2 show that the *H.R* of the cache for the queries with a small number of the groups is 100%, which is the consequence of the small enough hash tables that can be entirely located in the cache. In addition, in the group-by aggregation operation, each hash table/cache entry points to an individual group. Thus, repetitive keys that are located in a same group are also served from the same indexes of the hash table/cache. This situation leads to a high hit ratio of the cache.

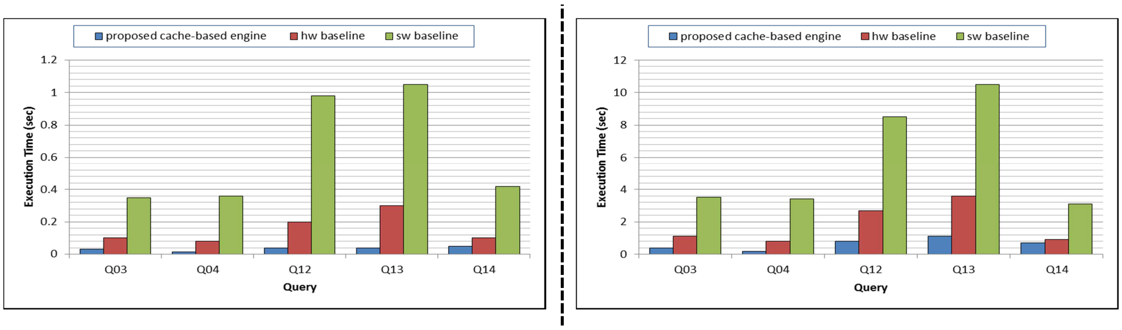


Fig. 4: The overall performance, comparing the proposed engine with a cache-less hardware and also software baselines for (left) 1GB and (right) 10 GB scales.

5.2 The Overall Performance Analysis

The total execution time of the studied queries is shown in Fig. 4. It includes the execution time of ***(i)*** the BRAMs-based design, where BRAMs are either used as the main hash table in 1GB scale or as the cache in 10GB scale, ***(ii)*** cache-less FPGA-based hardware baseline, and ***(iii)*** the software baseline.

For 1GB scale we achieved on average 4.6X and 18.9X, and for 10GB scale the speedup is on average 3X and 9.7X, comparing proposed hash join engine against hardware and software baselines, respectively. More specifically, we observed that:

- For 1GB scale that we could run all the studied benchmarks by exploiting BRAMs as the hash table, the speedup ranges from 2X to 7.5X, comparing proposed architecture to the hardware baseline.
- For the cache-based version in 10GB scale, the speedup ranges from 1.2X (Q14) to 4.4X (Q04). In Q14, the *H.R* of the cache is 9.6% on average, while it is 45.9% on average for the other queries. The main reason oh less *H.R* in Q14 is that it has no group-by operation, where the cache efficiently works.

Furthermore, comparing the proposed hash join engine to the software baseline, the achieved throughput improvement is mainly the consequence of the inherent capability of FPGA to perform dataflow execution in a deep pipelined fashion. As it can be seen, even baseline hardware version is on average 4X faster than software. However, additional optimizations in the proposed hash table caching mechanism substantially increase the speed up. We observed on average 14.3X speed up.

5.3 The Resource Utilization

The hardware resource usage of the baseline and proposed cache-based engines are shown in Table. 3. We observed that although, the utilization rates of the Look-Up Table (LUT) and Flip-Flop (FF) are almost similar in both versions, the usage of BRAMs is significantly different. Entirely 62% of available BRAMs are used as the cache that can deal with about 256K entries.

Table. 3: Hardware Resource Utilization Rates

	LUT	FF	BRAM
Baseline	128581 (1%)	150123 (2%)	12 (1%)
Cache-based	16368 (1.5%)	163854 (2%)	724 (62%)

6 RELATED WORK

Our design can be seen as a combination of the Ibex engine [17] and the hardware hash table chaining approach [8] with the main contribution of caching. For joining tables, hash joins are the most commonly used approach [19]. However, many examples of other types of table joins exist such as the merge join algorithm [3], the handshake join [16], etc.

Multithreading the build and probe phase engines have shown to offer direct performance benefits [7], [12]. Multithreading can effectively mitigate the DDR access latencies, with the overhead of needing more I/O bandwidth and the additional circuit to manage the concurrent threads. However, this technique can be integrated with the proposed hash table caching mechanism in this paper to achieve a significant throughput.

In [3], the authors design an FPGA prototype that can perform a parallel sort-merge join, making use of a sort tree as a prerequisite. In this work, we implement a hash join that can be inherently faster, as we do not perform any initial sorting step on the input data tables.

In Widx [12], an out of order SPARC v9 processor core is powered with a small core to accelerate the hash join operation with index walkers that walk multiple buckets, concurrently. This technique improves indexing performance of the TPC-H queries by 3.1X on average, while saves on average 83% of energy. Widx is similar to our approach, as it also aims to reduce the overheads of the pointer chasing (walking). However, Widx is a hardware-software codesign that follows a different approach with the proposed hash table caching method in this paper, which is entirely deployed in the hardware.

LINQits [4] accelerates a domain-specific query language called LINQ and is prototyped on a Zynq processor. It compares queries into hardware accelerator templates and for the hash join case, keeps the hash table in a sparse key table. It keeps the collided hash keys in the Spill Queue. Once reading its current partition is finished, it re-circulates the content of the Spill Queue (and its partition) until all the elements have been processed. Our proposed technique is designed not to require any Spill Queue or rehashing.

Finally, the proposed design could also be used together with the recent research on key-value stores [5], [11], [20]. Key-value stores are kind of unstructured (non-relational) databases, where the hash table is their key comprising component. The proposed hash table caching mechanism can be customized to improve the throughput of the key-value stores, as well.

7 CONCLUSIONS

In this paper, we have demonstrated the design of a novel cache-based query processing operations, i.e. hash join and group-by on FPGAs. Our contributions include hash table caching in the hardware and featuring collision, without reverting any software fallbacks. We showed the usefulness of the proposed hash table caching technique to process relevant hash join and group-by kernels in the TPC-H queries, with a maximum of 4.2X speedup over a pipelined baseline. Our experimental results show that we are enabled to both ***(i)*** use the full capacity of the DDR memory to store complete hash tables, and by employing a "hash table cache", ***(ii)*** to mitigate the long and irregular latencies of DDR memories, exploiting the fast BRAM resources of FPGA, which in turn significantly improves the performance of the hash join and group-by operations.

Acknowledgments. The research leading to these results has received funding from the European Union's Seventh Framework Program (FP7/2007-2013), for Advanced Analytics for Extremely Large European Databases (AXLE) project under grant agreement number 318633, and from the Ministry of Economy and Competitiveness of Spain under contract number TIN2015-65316-p.

REFERENCES

[1] Bluespec, Inc. http://bluespec.com/

[2] Netezza. The Netezza FAST engines framework. http://www. monash.com/uploads/netezza-fpga.pdf.

[3] CASPER, Jared; OLUKOTUN, Kunle. Hardware acceleration of database operations. In Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays. ACM, 2014. p. 151-160.

[4] CHUNG, Eric S.; DAVIS, John D.; LEE, Jaewon. Linqits: Big data on little clients. In ACM SIGARCH Computer Architecture News. ACM, 2013. p. 261-272.

[5] DE, Arup, et al. Minerva: Accelerating data analysis in next-generation SSDs. In Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on. IEEE, 2013. p. 9-16.

[6] DENNL, Christopher; ZIENER, Daniel; TEICH, Jurgen. On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library. In Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on. IEEE, 2012. p. 45-52.

[7] HALSTEAD, Robert J., et al. FPGA-based Multithreading for In-Memory Hash Joins. In Biennial Conference of Innovative Data Systems Research (CIDR), 2015.

[8] HALSTEAD, Robert J., et al. Accelerating join operation for relational databases with FPGAs. In Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on. IEEE, 2013. p. 17-20.

[9] HAYES, Timothy, et al. Vector extensions for decision support DBMS acceleration. In 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, 2012. p. 166-176.

[10] HE, Jiong; LU, Mian; HE, Bingsheng. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. Proceedings of the VLDB Endowment, 2013, vol. 6, no 10, p. 889-900.

[11] ISTVÁN, Zsolt, et al. A flexible hash table design for 10gbps key-value stores on FPGAs. In 2013 23rd International Conference on Field Programmable Logic and Applications. IEEE, 2013. p. 1-8.

[12] KOCBERBER, Onur, et al. Meet the walkers: Accelerating index traversals for in-memory databases. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2013. p. 468-479.

[13] KOCH, Dirk; TORRESEN, Jim. FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays. ACM, 2011. p. 45-54.

[14] KRISHNAMURTHY, Ravi, et al. Methods and systems for generating query plans that are compatible for execution in hardware. U.S. Patent Application No 12/168,821, 7 Jul. 2008.

[15] MUELLER, Rene; TEUBNER, Jens; ALONSO, Gustavo. Data processing on FPGAs. Proceedings of the VLDB Endowment, 2009, vol. 2, no 1, p. 910-921.

[16] OGE, Yasin, et al. An implementation of handshake join on FPGA. In Networking and Computing (ICNC), 2011 Second International Conference on. IEEE, 2011. p. 95-104.

[17] WOODS, Louis; TEUBNER, Jens; ALONSO, Gustavo. Less watts, more performance: an intelligent storage engine for data appliances. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, 2013. p. 1073-1076.

[18] WU, Lisa, et al. Q100: the architecture and design of a database processing unit. ACM SIGPLAN Notices, 2014, vol. 49, no 4, p. 255-268.

[19] ZELLER, Hansjörg; GRAY, Jim. An Adaptive Hash Join Algorithm for Multiuser Environments. In VLDB. 1990. p. 186-197.

[20] BLOTT, Michaela, et al. Achieving 10Gbps line-rate key-value stores with FPGAs. In Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing. 2013.

[21] ROY, Pratanu; TEUBNER, Jens; GEMULLA, Rainer. Low-latency handshake join. Proceedings of the VLDB Endowment, 2014, vol. 7, no 9, p. 709-720.

[22] Latest version of PostgreSQL 5.3. https://2ndquadrant.com/en/

[23] TPC-H benchmark set, http://www.tpc.org/tpch/.

[24] HAYES, Timothy, et al. Future vector microprocessor extensions for data aggregations. In Proceedings of the 43rd International Symposium on Computer Architecture. IEEE Press, 2016. p. 418-430.

[25] ARCAS-ABELLA, Oriol, et al. Hardware Acceleration for Query Processing: Leveraging FPGAs, CPUs, and Memory. Computing in Science & Engineering, 2016, vol. 18, no 1, p. 80-87.

[26] SALAMI, Behzad; ARCAS-ABELLA, Oriol; SONMEZ, Nehir. HATCH: Hash Table Caching in Hardware for Efficient Relational Join on FPGA. In Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on. IEEE, 2015. p. 163-163.