Scalable and Flexible High-Performance In-Network Processing of Hash Joins in Distributed Databases

Johannes Wirth*, Jaco A. Hofmann*, Lasse Thostrup†, Carsten Binnig†, Andreas Koch*

**Embedded Systems and Applications Group*
TU Darmstadt
Darmstadt, Germany
{wirth,hofmann,koch}@esa.tu-darmstadt.de
†*Data Management Lab*
TU Darmstadt
Darmstadt, Germany
{lasse.thostrup,carsten.binnig}@cs.tu-darmstadt.de

***Abstract*—Programmable switches allow to offload specific processing tasks into the network and promise multi-Tbit/s throughput. One major goal when moving computation to the network is typically to reduce the volume of network traffic, and thus improve the overall performance. In this manner, programmable switches are increasingly used, both in research as well as in industry applications, for various scenarios, including statistics gathering, in-network consensus protocols, and more.**

However, the currently available programmable switches suffer from several practical limitations. One important restriction is the limited amount of available memory, making them unsuitable for stateful operations such as Hash Joins in distributed databases. In previous work, an FPGA-based In-Network Hash Join accelerator was presented, initially using DDR-DRAM to hold the state. In a later iteration, the hash table was moved to on-chip HBM-DRAM to improve the performance even further.

However, while very fast, the size of the joins in this setup was limited by the relatively small amount of available HBM. In this work, we heterogeneously combine DDR-DRAM and HBM memories to support both larger joins and benefit from the far faster and more parallel HBM accesses. In this manner, we are able to improve the performance by a factor of 3x compared to the previous HBM-based work. We also introduce additional configuration parameters, supporting a more flexible adaptation of the underlying hardware architecture to the different join operations required by a concrete use-case.

***Index Terms*—HBM, FPGA, Hash Join, INP, In-Network Processing**

I. Introduction

The amount of data stored and processed using databases has exploded in recent years. This is especially true for analytical workloads, but also for many other applications. As a single server is often unable to handle these huge amounts of data by itself, distributed database systems are used to store and process the information. A significant downside of this distributed architecture, though, is the communication overhead incurred when the required data is split across multiple nodes. While there are operations that can be executed independently on each server, e.g., an SQL filter, several widely used database operations require the exchange of data between the involved nodes. One example for this type of communication-intensive operation is the SQL join. Prior work has shown that these operations do not benefit from distributed execution, and thus limit the performance of the entire system [11].

One approach to cope with these communication-intensive operations is the use of In-Network Processing (INP) [3, 6, 10, 12]. INP uses specialized switches which can be programmed to execute a specific operation. By offloading the execution of this operation from the involved nodes to the switch, the volume of data sent over the network can often be reduced. However, the current generation of programmable switches lack large memories, making them unsuitable for memory-intensive operations such as the aforementioned SQL join.

The authors of [9] have suggested to use an FPGA-based INP-capable switch architecture, which can be integrated into the Data Processing Interface (DPI) [7]. More recently, the authors of [13] proposed an improved architecture using fast HBM instead of DDR for better performance. However, this improved version came with two problems: First, the available amount of HBM is limited, putting a relatively low upper bound on the table sizes for the join operation. Second, that design had the potential for data races, leading to erroneous entries in the join result under certain conditions.

Our first contribution beyond [13] is a significantly improved hardware architecture, which can both handle larger joins by being able to use large off-chip DRAM banks to retain entries spilled from HBM, as well added synchronization logic that can track all in-flight requests in the highly parallel INP system, completely eliminating the risk of data races. This new architecture thus guarantees correct operation under all workloads. Then, we evaluate the performance of our new architecture and compare it with both [9, 13]. Our evaluation shows that the new architecture easily outperforms a conventional eight worker distributed database setup not using INP, and also improves the peak performance over [13] - which

This work was partially funded by the DFG Collaborative Research Center 1053 (MAKI) and by the German Federal Ministry for Education and Research (BMBF) with the funding ID 16ES0999. The authors would like to thank Xilinx Inc. for supporting their work by donations of hard- and software.

used HBM as well - by a factor of 3x.

The remainder of this paper is structured as follows: In Section II we present background information regarding the use of HBM for efficiently handling random accesses, and the internals of our SQL join implementation. Afterwards, Section III introduces our suggested architecture combining HBM and DDR, while Section IV discusses implementation details of the central part of the architecture, the parallel hashing units. Finally, we report the results of our evaluation in Section V, and conclude with a discussion of some remaining limitations and solution ideas in Section VI.

II. Background

A. HBM

HBM is a fast on-chip dynamic memory which is available on some FPGAs in the Xilinx UltraScale+ family of devices in capacities of $4\,\mathrm{GB}$ to $16\,\mathrm{GB}$. According to Xilinx datasheets [1, 14], the HBM achieves transfer rates of up to $460\,\mathrm{GB/s}$, making it significantly faster than traditional off-chip DDR4-DRAM. These high transfer rates are achieved using parallelism: The HBM is actually not a single monolithic memory, but rather consists of 32 independent ones. Each of these 32 memories offers a capacity of 1/32 of the total HBM size, and can be accessed via its own AXI3 slave port. These 32 AXI3 slave ports use a data-width of $256\,\mathrm{bit}$, and support a clock frequency of up to $450\,\mathrm{MHz}$. However, this high clock frequency is not always required: [13] shows that for the use case of small, random accesses at a granularity of $256\,\mathrm{bit}$, the peak performance is already reached at $200\,\mathrm{MHz}$.

B. SQL Join

The *join* is a common SQL operation which is widely used in relational databases for analytical processing and thus also frequently occurs in data center settings [4, 5]. The join operation merges a relation $\mathtt{A}$ with another relation $\mathtt{B}$ based on shared join-keys. Multiple join operations can be chained to merge more than two relations. While there are several implementations for the join operation, we will focus on the no-partition hash join [2] in this work, as this is a commonly used and well understood parallel hash join implementation.

This implementation is divided into two steps: First, a hash table is build with the contents of one of the relations. For this, the smaller relation - the so-called *dimension table* - should be chosen. Afterwards the hash table is probed using the entries of the larger relation - the so-called *fact table* - producing the result. When chaining multiple joins, it is possible to execute each of the two stages in parallel for all joins.

Unfortunately, the execution of the join operation becomes problematic in *distributed* database setups, where the contents of the different relations are spread across multiple nodes $\alpha, \beta, \gamma, \ldots$. In this case, it must first be ensured that tuples with same join-key from the two relations are processed on a single node. The traditional way to achieve this is by using a *shuffling* (or re-partitioning) step before the execution of each actual join as shown in Figure 1. This shuffling leads to heavy network traffic, which will often dominate the overall runtime of the distributed join.

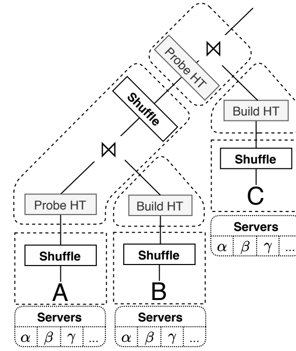


Fig. 1. Traditional distributed database join of the tables $\mathtt{A} \bowtie \mathtt{B} \bowtie \mathtt{C}$ distributed across multiple servers $\alpha, \beta, \gamma, \ldots$, requiring shuffle operations. HT stands for hash table.

To overcome this, we propose to execute the join operation following the *In-Network Processing* (INP) paradigm, using a custom high-performance FPGA-based INP-capable switch which is able to execute multiple chained joins on data flowing through the switch, thus eliminating the costly shuffles. The rest of this section will explain our INP implementation of the hash join algorithm in-depth, using the small example in Figure 2 for reference.

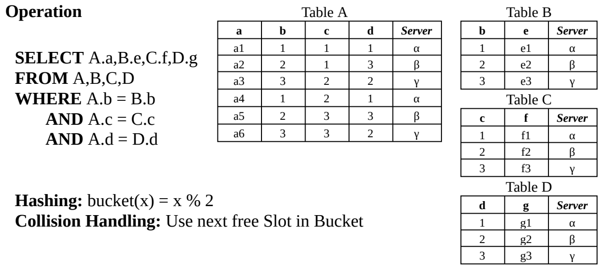
In the highly simplified example, we assume that each of the three servers α, β, γ holds just one row for each of the dimension tables, and just two rows for the fact table $\mathtt{A}$.

In the *Hashing Phase*, the three servers α, β, γ transfer the required contents of the dimension tables to the switch. This happens in parallel across multiple network ports of the INP switch, and is sensitive neither to the order of the transferred tables, nor to that of the individual tuples. The INP switch builds the hash tables of the received tuples by applying the hash function to the key to find the bucket number, and afterwards stores the tuple in this *bucket*. Hash collisions are resolved by having multiple *slots* within a bucket, and using the first available slot to hold the incoming tuple. In Figure 2, the buckets and slots used for the tuples incoming over the network are highlighted in the same colors.

After the hash tables have been built, the *Probing Phase* starts where the tuples of the larger fact table $\mathtt{A}$ are streamed in. As it, too, has been distributed over all of the servers α, β, γ, this again occurs over multiple network ports in parallel. Each of these tuples contains the three foreign keys for selecting the correct rows from the dimension tables. The actual values are then retrieved by using the foreign keys to access the correct bucket, finding the slot with the matching key, and returning the result to the server responsible for this part of the join. Again, we have marked the incoming foreign keys and the outgoing retrieved values in the same colors.

III. Architecture

In this section we present architecture, which improves upon prior solutions [9, 13] in three key aspects: First, [13] had



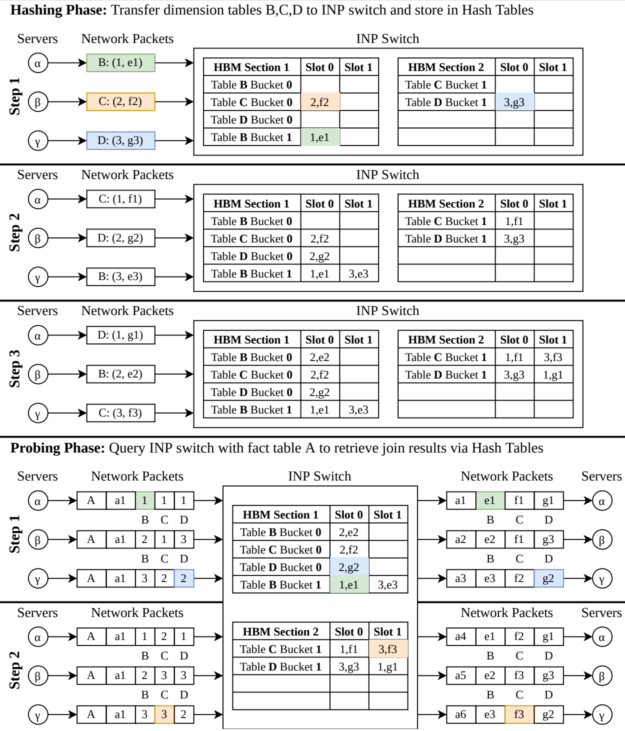


Fig. 2. Sample INP-style hash join over four tables, with data distributed over three servers. For clarity the join keys `A.b`, `A.c`, `A.d` are explicitly stated in a `WHERE` clause, instead of employing the `JOIN` keyword.

the, issue that read-after-write hazards could occur when a *second* request to a specific bucket was started while the *first* request to the bucket was still in-flight. While the probability for these errors was rather low (5.45×10^{-6}) [13], due to the large data sizes they still caused some incorrect results. Our new architecture completely eliminates these errors by always keeping tracking of in-flight requests.

Our second improvement concerns the maximum size of the hash tables, and thus the dimension tables of the join operation. In both [9] and [13] only 8 GB of memory was available, limiting the maximum size of the joins. Join operations exceeding that size would lead to the INP join failing, which would be reported to the database to redo the join in the traditional manner. Our new architecture uses off-chip DDR4-DRAM as a secondary memory to also support dimension tables exceeding the HBM capacity.

And finally, our new architecture improves the overall system performance by introducing a more *parallel* architecture.

The rest of this section will first give an overview of our architecture. Afterwards, we present the functionality of the different parts and how they interact. Due to the size and complexity of our architecture, especially for the many internal and external memory interfaces, significant optimization effort was required in order to reduce the resource usage and get the design through Vivado P&R at a reasonable clock frequency. We will discuss the most important of these optimizations, some of which are specific to the Xilinx UltraScale+ family of devices, in the following text.

Furthermore, our architecture is highly configurable. We will discuss the impact of the most important configuration parameters, which will be typeset in this text as configuration parameter (C) [1,2,4], e.g., with regard to the achievable performance and the resource usage. Some of these parameters also use a shorthand given in parentheses. Where applicable, we show example values in square brackets.

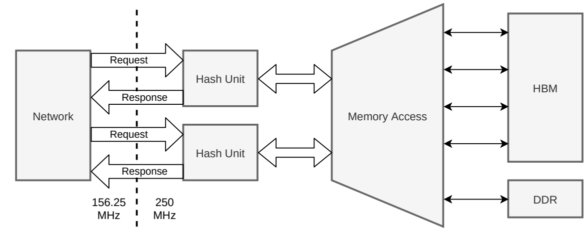


Fig. 3. Architecture overview with two hash units, four HBM ports and one DDR4-DRAM interface.

Figure 3 presents an overview of our architecture. From left to right, our architecture consists of three major parts: The *Network* part, the *Hash Units* and the *Memory Access Unit*. These will be detailed in Section III-A, Section III-B and Section III-C respectively.

Our design uses two clock domains: The network part runs at 156.25 MHz - this is required by the external 10G network ports - while the rest of our design runs at 250 MHz. As stated in Section II-A, the latter is sufficient to achieve the peak performance of the HBM here, because our architecture uses only 256 bit wide *random* memory accesses.

Two important configuration parameters affect the entire design: Each *Hash Unit* manages one hash table, thus the number of Hash Units (HU) [2,3,4] determines the number of parallel (chained) join operations the design can execute. As each *Hash Unit* has separate connections to the *Network* and *Memory Access* parts, this parameter affects all parts of the architecture.

The degree of parallelism (P) [1,2,4] determines the number of requests each *Hash Unit* can handle in parallel per cycle, and thus highly impacts the performance of the entire system. To allow parallel operation, the *Network* part needs to provide multiple requests per cycle to each *Hash Unit*, and the *Memory Access* part must be able to handle multiple memory accesses per cycle. Thus, the connections from the *Hash Units* to the *Network* and *Memory Access* part shown in Figure 3 are actually *multiple* independent connections, the number depending on the configured degree of parallelism. For the rest of this work, we will use the term *channel* when talking about these connections:

A single channel can transfer at most one data item per cycle per direction.

A. Network

The *Network* part contains multiple *Network Units*, which are grouped into P *Network Groups*, with each group having one *channel* to each of the hash units in the design. This ensures that the network part is able to provide sufficient requests to fully utilize the parallelism in the hash units. Each network unit connects to one of the external network ports and is responsible for parsing incoming Ethernet frames and forwarding the contained requests to the respective hash unit(s).

There are two types of requests: *hashing* requests contain a key-value pair that should be added to a given hash table, and *probing* requests, which contain several keys - one for each of the dimension tables - which are used to probe the respective hash table to find the matching value. Each Ethernet frame contains multiple requests of a single type.

It is a hard requirement by the external network port and vendor-provided Ethernet core that our network units never block. The incoming data width and clock frequency are determined by the Ethernet core as well: Our Network Units must be able to handle $64\,\mathrm{bit}$ each cycle at $156.25\,\mathrm{MHz}$. We use a large FIFO to buffer incoming requests before forwarding them to the respective hash units. This allows us to hide small stalls in the rest of the architecture without immediately having to drop frames. The FIFO also synchronizes the requests into the clock domain of the hash units. If there are too many incoming requests, we need to drop frames to avoid the FIFO filling up completely - which would stall further frame processing. When a frame does need to be dropped, we retain the sequence number of the dropped frame to re-request the data from the original sender, ensuring that the requests of the dropped frame are not lost, but will just be processed at a later time.

In addition, the network units are also responsible for sending probe results back to the sender. To allow this, each probe request to the hash units contains the ID of the originating network unit. This information is retained during execution of the request, and then used to return the reply to the correct network unit after completion. The network units collect these responses and send them back to the external sender in the same frame format as the incoming data.

B. Hash Unit

The *Hash Units* form the core of our architecture and execute the algorithm presented in Section II-B. Each hash unit is responsible for building the hash table for exactly one dimension table, thus the number of hash units is equal to the number of joins the design can process at once. As stated earlier, this is an important configuration parameter in our architecture.

Compared to previous work, the hash units have been heavily modified and augmented with two major design goals:

First, the hash units are now able to keep track of all in-flight requests to the memory. This is necessary to prevent RAW data hazards when a specific bucket is accessed before a modification of the same bucket is completed. Our solution is similar to the *miss status holding registers* (MSHRs) used in cache architectures. We also experimented with moving this request-tracking functionality to near the HBM ports. However, we did not find a solution ensuring correctness, while achieving the required throughput with acceptable area efficiency.

Secondly, our new hash units are now able to process *multiple* requests per cycle, matching the degree of parallelism. This change allows us to improve the overall performance beyond what was possible in previous work. The degree of parallelism also affects the outside connections of the hash units. Namely, it implies an equal number of *channels* from each hash unit to the network and memory access parts.

As the hash units are the central part of our design, we provide more details about their implementation in Section IV.

C. Memory Access Unit

The *Memory Access Unit* is responsible for connecting the hash units with the available memory, namely the HBM and off-chip DDR4-DRAM.

As explained earlier, each hash unit has multiple *channels* to the memory access unit, matching the degree of parallelism. However, it is not necessary to connect all *channels* of all hash units to each HBM slave port: The available HBM is partitioned evenly across all of these connections, thus each connection only needs access to the HBM slave ports of its own partition. With, e.g., two hash units and a degree of parallelism of two for each unit, the 32 HBM slave ports are thus partitioned evenly into 8 HBM slave ports per connection.

The memory access unit works as follows: It receives memory accesses from the hash units, and forwards them to the correct HBM port, depending on the address, and returns the responses to the requester. The memory accesses from the hash units also include a special *Cache Line Index*, which needs to be preserved by the memory access unit, and then returned with the response. The index is used to allow correct out-of-order processing of the responses in the hash units.

While this functionality is straightforward, the implementation complexity of the memory access unit mainly stems from two aspects: First, it is challenging to route the connections to up to 32 HBM memories, while keeping the resource usage reasonable and still provide good throughput. Thus it is of paramount importance to keep the individual parts as small and simple as possible. Second, the requirements on the memory access unit depend on the degree of parallelism and the number of hash units, as together these determine the number of *channels* coming from the hash units, and thus also how many HBMs ports need to be connected to each channel. Therefore, the memory access unit is not a static component, but is highly parameterized to be automatically generated matching the current requirements. This also requires corner-case handling, e.g., when the number of HBMs cannot be equally partitioned across the channels. To this end, our implementation is also able to share a HBM between two channels, if required.

IV. Hash Units

A. Hash Table

The available HBM is divided into HU memory ranges, each holding one of the hash tables. Each hash table contains a number of *buckets* with four *slots* per bucket. The slots are used for collision handling: if the first slot of a bucket is already occupied, the data can be stored in the second slot. Each slot can store a pair containing a $32\,\mathrm{bit}$ key and $32\,\mathrm{bit}$ value. The number of four slots per bucket was chosen so that the total width of a bucket matches the $256\,\mathrm{bit}$ access width of a HBM port. To indicate empty slots we use the reserved value -1. Our architecture will automatically initialize the memory correctly before executing the join. The time required for this initialization is *not* included in our evaluation, as it is negligible compared to the total runtime.

If more than four tuples must be stored in a single bucket due to collisions, a *bucket overflow* occurs. In this case, we move the full bucket from the HBM to the DDR4-DRAM, and thus have an empty bucket in the HBM again. In the probing phase, we then also need to check the DDR4-DRAM for the matching entry if it was not found in the HBM. The maximum required size of one hash table is determined by the number of entries in the dimension table and the size of a single hash table entry. As our architecture uses $32\,\mathrm{bit}$-wide keys and values, the dimension table size is capped at 2^{32} entries. Each entry is $64\,\mathrm{bit}$ wide, yielding a maximum of $2^{32} * 8B = 32GiB$ per hash table. As long as the off-chip DDR4-DRAM is sufficiently large to fully store the required number of hash tables, this approach ensures that no data is lost and we always produce correct results.

However, due to the performance gap between HBM and off-chip DDR4-DRAM, extensively spilling buckets from HBM to the DDR4-DRAM will severely reduce the processing speed of our design.

The number of buckets per hash table depends on the available HBM memory and the number of Hash Units in the design. It can be computed by

$$\text{\#BUCKETS} = \frac{\text{HBMSIZE}}{256\,\mathrm{bit} \cdot \text{HASHUNITS}} \tag{1}$$

For 3 hash units and $8\,\mathrm{GB}$ of HBM, this results in a number of around 90 million buckets per hash table. More details of our hash table operations are explained in Section II-B.

B. Hash Unit Structure

Figure 4 shows the internal structure of one Hash Unit. Incoming requests from the network ports are first forwarded to the *Hasher* (1) which calculates the bucket for the request as well as the HBM address of this bucket. There is a configuration option use hash function which determines how the hasher actually computes the bucket: If it is activated, the bucket is computed as

$$\text{bucket} = hash(\text{key}) \bmod \text{\#BUCKETS}$$

If it is deactivated, the hashing is skipped and only the modulo operation is applied. The hash function can be skipped without adverse effects when dealing with unique keys - which is typically the case in database environments. Afterwards, an entry in the *Completion Buffer* (2) is reserved. This is only done for probing requests, as hashing requests do not require a response to the sender. This reserve operation will return a *token*, which is added to the request information. It will later be used by the completion buffer to reorder the responses according to the original request order.

The following steps require the requests to be grouped by the most-significant bits of the HBM address. Up to this point, they were grouped by their originating *Network* group. Thus, the requests are first routed to FIFOs (3) depending on their most-significant address bits, and then redistributed (4) to the respective *Request Cache* (5). The *token* generated by the completion buffer also contains information about the originating network group, allowing the completion buffer to return the result to the correct group after completion.

In contrast to traditional caches, the *Request Cache* does *not* store data read from the memory for later accesses, but is only used to keep track of the in-flight requests. However, it works similarly to the *miss status holding registers* (MSHRs) found in traditional caches. Thus we also use the well-known cache-terminology where applicable.

Instead, the main task of the Cache is to prevent data hazards when, during the hashing phase, a bucket is accessed again before an earlier access has completed. The cache prevents these hazards by storing the in-flight memory accesses in *Cache Lines* (5.3), indexed by the lower bits of the bucket number. As each cache line can only hold one entry, this will stall subsequent requests to the same bucket until the first request completes. The contents of a cache line include a valid bit, the bucket, the key, and either the value (in the case of a hash operation), or the completion buffer token (in the case of a probe operation).

The request cache must be able to handle two events: Incoming requests from the FIFOs (4), and responses from the memory (6). The pseudo-code of these two operations can be found in Listing 1.

Note the DDR4-DRAM accesses in lines 22 and 33, for which a separate request cache exists. There is always only one instance of this DDR request cache, independently of the parallelism in the rest of the design. It operates similarly to the caches discussed here, which handle the in-flight HBM requests. Also, note that single a bucket may be spilled to DDR (l. 22) multiple times during hashing, resulting in multiple *instances* of this bucket in the DDR. Thus, *queryDDR* in line 33 can actually result in multiple DDR reads: The DDR request cache will go through *all* instances of this bucket, until a matching pair is found.

Early experiments have shown that the number of cache lines must be in the order of several thousands to achieve a reasonable performance. Combined with the fact that it also needs to be able to execute *both* functions shown in Listing 1 in each cycle, this gives rise to an implementation challenge: Typically, the relatively small amount of memory for the request cache would best be implemented in fast on-chip BRAM. But

Fig. 4. Internal Structure of one Hash Unit with a degree of parallelism of two. The extra connection for DDR accesses is omitted in this figure.

```
1  function insertRequest(request) {
2    line_index = getCacheLineIndex(request.bucket)
3    if (isCacheLineEmpty(line_index)) {
4      occupyCacheLine(line_index)
5      fillCacheLine(line_index, request)
6      // fetch bucket content from memory
7      // line_index will be preserved with the access
8      readHBM(request.bucket, line_index)
9    } else {
10     // do nothing
11   }
12 }
13 function handleReadResponse(data, line_index){
14   // extract original request from cache line
15   request = readCacheLine(line_index)
16   if (request.isHashing) {
17     // add (key, value) to empty slot in bucket
18     addPair(data, request.keyValuePair)
19     // write updated bucket to memory
20     if (isBucketFull(data)) {
21         // spill full bucket out to DDR
22         writeDDR(request.bucket, data)
23         // clear bucket in HBM
24         writeHBM(request.bucket, emptyBucket)
25     } else{
26         writeHBM(request.bucket, data)
27     }
28   } else {
29     // find matching pair in bucket
30     pair = find(data, request.key)
31     if (pair == null) {
32       // no match found
33       queryDDR(request)
34     } else {
35       // forward found pair to completion buffer
36       completeOutput(request.token, pair)
37     }
38   }
39   clearCacheLine(line_index)
40 }
```

Listing 1. Pseudocode for request cache operations.

two issues arise from this: First, when accessing BRAM, the response - including the data when reading - is only available in the *following* clock cycle. Secondly, BRAM at most allows *two* memory accesses per cycle. This is not sufficient here: When inserting a request, we need to read a cache line to check if it is empty (l. 3), and write a cache line (l. 4,5). When handling the memory response, we need to read the cache line (l. 15), and clear it (l. 39).

To overcome these issues, we store the valid bit of the Cache Lines *separately* using the Xilinx LUTRAM primitives, while the rest is stored in BRAM. The LUTRAM primitive we use stores up to 64 1-bit entries, allows one write operation and up to *three* read operations per cycle. This means we can, e.g., use just 32 LUTRAMs to store the valid bits of 2048 cache lines. Furthermore, when reading from the LUTRAM, the data is available in the *same* cycle. This means the operations accessing the valid bit (l. 3,4,39) need to access only the LUTRAM, while the operations accessing the rest (l. 5,15) access only the BRAM. With this approach, we require only one read and one write per cycle to the BRAM, thus the two available ports of the BRAM suffice. However, the valid bit in the LUTRAM still needs to be written twice (l. 4,39) per cycle, while a LUTRAM primitive only allows a single write per cycle. In most cases this is actually not relevant, as the two parallel write accesses will often target two *different* LUTRAM instances. In the rare case where both writes need access to the same instance, we need to delay one of them. It is *not* possible to delay occupying a cache line (l. 4), because this could result in incorrect behavior, as the cache line would be seen as empty in the next cycle. However, it *is* possible to delay *clearing* a cache line (l. 39), as this does not affect correctness.

Another issue arises, when the cache line, to which a new request is mapped, is already occupied: In this case, the new request is blocked until the previous request in this cache line is completed (l. 10) - unfortunately this would by default stall *all* following requests as well due to head-of-line blocking, and thus significantly reduce the performance. To overcome this, incoming requests are not stored in a single FIFO (which causes the head-of-line blocking), but rather multiple *Bucket Group FIFOs* (5.1). Each of these FIFOs is responsible for a range of cache lines, and incoming requests are distributed to the FIFOs accordingly. In each cycle, all FIFOs check if the cache line of their first entry is currently empty - if so, they indicate this to an *Arbiter* (5.2). This arbiter then uses a round-robin approach to decide which *Bucket Group FIFO* is chosen in each cycle, while giving priority to FIFOs that are more than half full.

V. Evaluation

For the evaluation of our architecture, we focus on two aspects: First, we show the performance and scaling of our system itself, comparing various configuration options with regard to the degree of parallelism, the number of Hash Units, and the effect of using the DDR4-DRAM as secondary memory for handling HBM spills. We also examine the resource usage for these different configurations. Afterwards, we compare our proposed system to a classical distributed database setup using eight servers, as well as the prior work from [9, 13].

For the evaluation of our architecture, we use special hardware *Benchmark Units* which generate the network traffic containing the requests for our design on the FPGA. These benchmark units can be fine-tuned to generate the exact load required, thus allowing us to more accurately measure the performance of our system than when using software-generated workloads., which often suffer from OS and PCIe bus interference. In a real-world setup, the benchmark units are just replaced by the connections to the physical network ports, without any changes to our architecture itself. The benchmark units are controlled via PCIe. Their programming options include the number of active benchmark units and the sizes of the dimension tables and the fact table. Each active benchmark unit will then be assigned a portion of all dimension tables and the fact table, and will then generate the hash and probe requests from this partition in a random order.

For our evaluation we use the Bittware XUP-VVH board with a Xilinx UltraScale+ VU37P FPGA having 8 GB of HBM. Our board carries two 128 GB DDR4-DIMMs which we use as secondary memory. For the external connections - including HBM, DDR4 and the network ports - we rely on the TaPaSCo [8] FPGA middleware and SoC composition framework.

A. Absolute System Performance

The performance of our system depends on the configuration options. First, we explore the impact of the degree of parallelism and number of Hash Units on the peak performance of our architecture. For this experiment, the dimension table size is fixed to 100×10^6 entries, which can be stored completely in HBM.

TABLE I
Peak Performance of our Architecture for different Parallelism and Hash Unit configurations, as million operations per second. The numbers in parentheses are the minimum number of 10G ports required to achieve the performance. The performance of [13] for HU=3 is given for reference.

Parallelism	#HU=2		#HU=3		#HU=4	
P = 2	474	(10)	462	(14)	470	(16)
P = 4	782	(20)	746	(20)	703	(20)
[13]			248			

This allows us to measure the peak performance our architecture can achieve, since it avoids the overhead for spilling buckets to DDR4-DRAM. Table I shows the maximum operations per second our architecture achieves for the probing phase. We focus on the probing phase here, as it dominates the runtime of the join in a typical setup with smaller dimension tables, and a large fact table. However, the performance of the hashing phase scales similarly.

Two key observations can be made: First, our proposed architecture improves over [13] by a factor of 1.86x or 3x, for a degree of parallelism of two or four, respectively. The slower speedup scaling when increasing the parallelism from two to four is due to the limited memory bandwidth: As the number of HBMs on the chip is fixed, and the number of *channels* between the hash units and the memory access unit *grows*, fewer HBMs are available *per channel*. Thus, each individual channel achieves a lower performance, as it is constrained by its fraction of the total available memory bandwidth. Secondly, the minimum number of 10G ports required to fully load our architecture with requests increases with the number of hash units. This is explained by the fact, that a probing request contains one key for the fact table and one key for *each* dimension table. Thus, the size of each probing request *grows* with more hash units, resulting in *fewer* transferred requests per second for a fixed network bandwidth, in turn needing more 10G ports to reach the peak rate of requests per second.

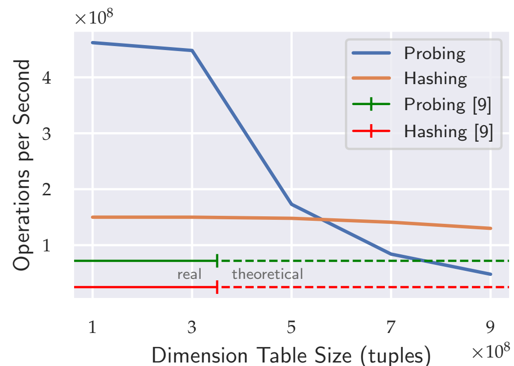


Fig. 5. Performance impact of using DDR4-DRAM for spilling buckets. With a growing dimension table size, the number of spilled buckets increases, putting more load on the DDR4-DRAM subsystem. The performance values from [9] are given as reference.

In a second experiment, we measure the performance impact when the off-chip DDR4-DRAM is used for spilling hash buckets. We measured the performance of both phases individually, using an architecture configured with three hash units and a parallelism of two. The results are shown in Figure 5, which also gives the performance of the DDR-only design of [9] for reference. Note that [9] reported their performance at 50×10^6 elements per dimension table, which is is the value we use in Figure 5. Furthermore, [9] uses only 8 GB of memory, and can thus support at most 350×10^6 elements. For easier comparison, the curves for [9] extend beyond this point.

As expected the performance of the probing phase drops significantly as soon as more buckets need to be spilled to DDR4-DRAM, starting around a dimension table size of 300×10^6 elements. Starting at 700×10^6 elements, some buckets have received more than eight values during *hashing*,

resulting in *multiple* spilled instances of this specific bucket. In turn, multiple DDR reads for finding a matching tuple might be necessary during *probing* to iterate over all of the spilled bucket instances. Performance will drop further with every spilled bucket, bounded only when reaching the maximum dimension table size introduced by the width of the unique keys. On the other hand, the performance of the *hashing* phase drops only slightly. This is due to the hash table being built completely in the fast HBM, with the delays for intermittent spills from HBM to DDR4 being mostly hidden by the normal HBM-based operation.

Overall, though, our new architecture is a significant improvement over prior DDR-only work: When mainly operating in HBM, probing is up to 6x faster for a parallelism of two, and 10x faster for a parallelism of four, than its DDR-only predecessor. As explained before, the performance degrades gracefully when HBM capacity is exhausted, at some point even below the *theoretical* performance of [9]. We write *theoretical*, because these numbers can *not* be directly compared: The performance of [9] is given at 50×10^6 elements per dimension table, while our architecture only drops below this level at 700×10^6 elements. The performance of [9] does not drop significantly when increasing the number of elements, but it can handle at most 350×10^6 elements due to the limited memory capacity.

TABLE II
RESOURCE USAGE OF OUR ARCHITECTURE FOR DIFFERENT CONFIGURATIONS. THE PERCENTAGE OF THE AVAILABLE RESOURCES ON THE VU37P FPGA IS GIVEN IN PARENTHESES.

	LUTs	Registers	CLBs	BRAMs
2HU2P	156k (11.9%)	157k (6.0%)	34k (21.1%)	143 (7.1%)
2HU4P	273k (21.0%)	230k (8.8%)	54k (32.9%)	307 (15.2%)
3HU2P	194k (14.9%)	191k (7.3%)	43k (26.2%)	174 (8.6%)
3HU4P	367k (28.1%)	283k (10.8%)	74k (45.6%)	363 (18.0%)
4HU2P	218k (16.7%)	206k (7.9%)	46k (28.5%)	220 (10.9%)
4HU4P	436k (33.4%)	307k (11.8%)	82k (50.6%)	467 (23.2%)

Finally, we discuss the resource usage for the different configuration options of our architecture in Table II. As expected, the resource usage grows when increasing the number of hash units or the degree of parallelism. The key resource is CLBs, of which 50% of the VU37P device are required for the largest design with HU=P=4. From measurements taken at the on-board power management chip, the join accelerator board draws 47 W when idle (most of it going to keeping the network connections and memories up), and just up to 47 W when running.

B. *Performance Comparison with Prior Work*

In the second part of our evaluation, we compare the performance of our system to a software baseline not using INP, and the previous work [9, 13]. The software baseline is executed on eight workers, each using an Intel Xeon Gold 5120 CPU @ 2.2 GHz using 384 GB of RAM. The servers are connected via 10G BASE-T using CAT6 RJ45 cables. Furthermore, a master server with the same hardware is used to control execution. We use eight threads to execute the software baseline for our experiments. Further increasing the number of threads does *not* improve the performance as the bottleneck is – depending on the phase – either the 10G network connection or the DRAM.

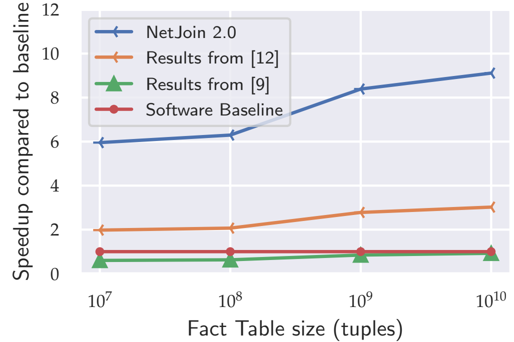


Fig. 6. Speedup of our architecture over the *best-case* software baseline using eight workers. The prior work [9, 13] is shown for comparison. This experiment uses three joins with a dimension table size of 100×10^6 tuples.

The scenario used for this comparison is joining a fact table `A` with three dimension tables `B`, `C` and `D`. The dimension table size is fixed at 100×10^6, while the fact table size is varied. Figure 6 shows the speedup of our proposed architecture and that of the prior work [9, 13] over the software baseline. Note that for this experiment, the keys are distributed *equally* across the eight workers, which is the *best-case* scenario for the software baseline. It can be seen that our new architecture performs at 6x. . . 9x of the performance of the software baseline and the DDR-only accelerator [9], and 3x that of the HBM-only unit of [13]. Note that for the chosen dimension table size of 100×10^6 tuples, no spilling to DDR occurs. As discussed in the previous section, spilling significantly reduces the performance of the new architecture. However, we cannot show this drop in relation to earlier work, as neither of the two prior INP solutions can actually handle joins which exceed a memory capacity of 8 GB, and the slowdown in our new approach becomes pronounced only for the larger joins our new architecture is capable of, as indicated in Figure 5.

VI. CONCLUSION AND FUTURE WORK

This work is motivated by two limitations of existing INP join accelerators: First, prior work did not scale up to larger dimension table sizes, and additionally risked incorrect results due to data hazards. We propose an improved architecture, which uses a cache-like approach to prevent the incorrect results, and uses off-chip DDR4-DRAM as secondary memory to support larger dimension tables. Furthermore, our improved architecture also improves the performance by a factor of 3x compared to previous work. Note that we used the *best-case* scenario for software as a baseline.

Even with the achieved advances, several options for future improvements exist. These include integrating our proposed architecture into a full-scale FPGA-based switch to enable the use in a real-world distributed database setup, as well as schemes for faster lookups of spilled data during probing.

REFERENCES

[1] AXI High Bandwidth Memory Controller. https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf, Product Guide, Xilinx, July 2020.

[2] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 37–48. ACM, 2011. DOI: 10.1145/1989323.1989328. URL: https://doi.org/10.1145/1989323.1989328.

[3] M. Blöcher, T. Ziegler, C. Binnig, and P. Eugster. Boosting scalable data analytics with modern programmable networks. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, DAMON '18, Houston, Texas. Association for Computing Machinery, 2018. ISBN: 9781450358538. DOI: 10.1145/3211922.3211923. URL: https://doi.org/10.1145/3211922.3211923.

[4] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, June 1984. ISSN: 0163-5808. DOI: 10.1145/971697.602261. URL: https://doi.org/10.1145/971697.602261.

[5] M. Dreseler, M. Boissier, T. Rabl, and M. Uflacker. Quantifying TPC-H choke points and their optimizations. *Proc. VLDB Endow.*, 13(8):1206–1220, 2020. URL: http://www.vldb.org/pvldb/vol13/p1206-dreseler.pdf.

[6] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: smartnics in the public cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 51–64, Renton, WA, USA. USENIX Association, 2018. ISBN: 9781931971430.

[7] A. Gustavo, C. Binnig, I. Pandis, K. Salem, J. Skrzypczak, R. Stutsman, L. Thostrup, T. Wang, Z. Wang, and T. Ziegler. Dpi: the data processing interface for modern networks. *Proceedings of CIDR 2019*, 2019.

[8] C. Heinz, J. Hofmann, J. Korinth, L. Sommer, L. Weber, and A. Koch. The tapasco open-source toolflow. In *Journal of Signal Processing Systems*, 2021.

[9] J. Hofmann, L. Thostrup, T. Ziegler, C. Binnig, and A. Koch. High-performance in-network data processing. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2019, Los Angeles, United States.* 2019.

[10] S. Mühlbach, M. Brunner, C. Roblee, and A. Koch. Malcobox: designing a 10 gb/s malware collection honeypot using reconfigurable technology. In *IEEE Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*. IEEE, 2010.

[11] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *Proc. VLDB Endow.*, 9(4):228–239, Dec. 2015. ISSN: 2150-8097. DOI: 10.14778/2856318.2856319. URL: https://doi.org/10.14778/2856318.2856319.

[12] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 150–156, Palo Alto, CA, USA. Association for Computing Machinery, 2017. ISBN: 9781450355698. DOI: 10.1145/3152434.3152461. URL: https://doi.org/10.1145/3152434.3152461.

[13] J. Wirth, J. Hofmann, L. Thostrup, A. Koch, and C. Binnig. Exploiting 3d memory for accelerated in-network processing of hash joins in distributed databases. In *International Symposium on Applied Reconfigurable Computing (ARC)*, 2021.

[14] M. Wissolik, D. Zacher, A. Torza, and B. Day. Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance. https://www.xilinx.com/support/documentation/white_papers/wp485-hbm.pdf, White Paper, Xilinx, July 2019.