Exploiting 3D Memory for Accelerated In-Network Processing of Hash Joins in Distributed Databases *

Johannes Wirth[1[0000−0002−5211−2544]], Jaco A. Hofmann[1], Lasse Thostrup[2], Andreas Koch[1[0000−0002−1164−3082]], and Carsten Binnig[2]

[1] Embedded Systems and Applications Group TU Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany
{wirth,hofmann,koch}@esa.tu-darmstadt.de
https://www.esa.informatik.tu-darmstadt.de/

[2] Data Management Lab TU Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany
{lasse.thostrup,carsten.binnig}@cs.tu-darmstadt.de

Abstract. The computing potential of programmable switches with multi-Tbit/s throughput is of increasing interest to the research community and industry alike. Such systems have already been employed in a wide spectrum of applications, including statistics gathering, in-network consensus protocols, or application data caching. Despite their high throughput, most architectures for programmable switches have practical limitations, e.g., with regard to stateful operations.

FPGAs, on the other hand, can be used to flexibly realize switch architectures for far more complex processing operations. Recently, FPGAs have become available that feature 3D-memory, such as HBM stacks, that is tightly integrated with their logic element fabrics. In this paper, we examine the impact of exploiting such HBM to accelerate an inter-server join operation at the switch-level between the servers of a distributed database system. As the hash-join algorithm used for high performance needs to maintain a large state, it would overtax the capabilities of conventional software-programmable switches.

The paper shows that across eight 10G Ethernet ports, the single HBM-FPGA in our prototype can not only keep up with the demands of over 60 Gbit/s of network throughput, but it also beats distributed-join implementations that do not exploit in-network processing.

Keywords: HBM and FPGA and Hash Join and INP and In-Network Processing

* 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.

1 Introduction

Distributed database systems are commonly used to cope with ever-increasing volumes of information. They allow to store and process massive amounts of data on multiple servers in parallel. This works especially well for operations which require no communication between the involved servers. For operations which *do* require communication - such as non-colocated SQL joins - sending data between servers can quickly become the performance bottleneck. For these operations, a distributed setup does not always achieve performance improvements, as shown in [11].

Recent papers have suggested the use of In-Network Processing (INP) [2,5,12] to accelerate distributed computations. INP employs programmable switches to offload processing across multiple servers into the network itself, which reduces the volume of inter-server data transfers. However, the current generation of programmable switches still have some limitations in this scenario. For instance, many current INP solutions are restricted to mostly stateless operations, as they lack large memories. This limits the applicability of these switches for INP, as the aforementioned joins, e.g., cannot be implemented without keeping large state.

As a solution, [8] proposes a new INP-capable switch architecture based on an FPGA, which can be integrated into the Data Processing Interface (DPI) [6] programming framework for INP applications. This architecture provides much more flexibility compared to software-programmable switches and, in addition, is well suited for memory-intensive operations. FPGA based architectures have been show to support hundreds of Gbit/s of network throughput [14], but for stateful operations, such as a database join, memory bandwidth is *still* the limiting factor [8].

Recently, FPGAs using 3D-memory, such as High-Bandwidth-Memory (HBM), have become available. This new memory type allows to perform multiple memory accesses in parallel, resulting in a huge increase in performance compared to traditional DDR memory. However, because multiple parallel accesses are *required* to achieve a performance advantage, the user logic must be adapted in order to actually exploit the potential performance gains.

Our main contribution is to adapt an FPGA-based INP switch architecture [8] to use HBM efficiently. To achieve this, we compare the performance of HBM for different configurations to determine the best solution for our architecture. Finally, in our evaluation we show that our HBM-based version can achieve *more than three times* the throughput of the older DDR3-SDRAM based INP accelerator, and easily outperforms a conventional eight server distributed database setup not using INP.

The remainder of this paper is structured as follows. In Section 2 we introduce the organization of HBM on Xilinx FPGAs and analyze its performance for different configurations. Afterwards, Section 3 introduces the hash join operation which is used as an example INP operation for our proposed architecture. In Section 4 we present a new HBM-based architecture for INP-capable switches. Finally, we report our experimental results in Section 5, and discuss some limitations of the current implementation with possible refinements, in Section 6.

2 HBM Performance

This section introduces the HBM organization on current Xilinx FPGAs, afterwards the HBM performance for different configurations is analyzed.

2.1 HBM Organization

Selected Xilinx FPGAs include HBM, offering a range of number of logic cells and the available amount of HBM. For the currently available devices, this amount ranges from 4 GB to 16 GB. Independent of the size, the organization of the HBM does *not* differ: The HBM on these devices is split into two *stacks*, which are attached at the bottom edge of the FPGA matrix. Memory access is realized via eight memory channels per stack - each providing two pseudo channels, resulting in a total of 32 pseudo channels over both stacks. Each pseudo channel may only access its associated *section* of memory (1/32 of the available total memory). The programmable logic can access the HBM via 32 AXI3 slave ports. By default, each AXI3 slave port is directly connected to one of the pseudo channels, so each AXI3 port can only access one memory section. Alternatively, an optional AXI *crossbar* can be activated, which allows each AXI3 port to access the *entire* memory space - but at a cost in latency and throughput. In this work, we do *not* use the optional crossbar. Figure 1 shows the organization of one HBM stack.

The 32 AXI3 slave ports are distributed over the entire width of the bottom edge of the FPGA matrix. Each port has a data width of 256 bit and can run at a clock frequency of up to 450 MHz. This results in a theoretical aggregate memory bandwidth of 460 GB/s.

2.2 HBM Performance

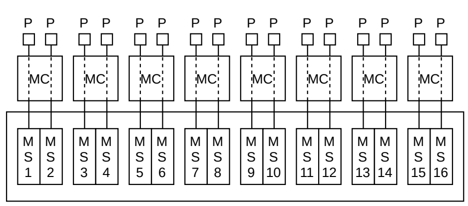


Fig. 1. Organization of a single HBM stack, showing the memory channels (MC), memory sections (MS), and AXI3 slave ports (P). Each line between an AXI3 port and a memory section indicates a pseudo channel. The optional AXI crossbar is not shown.

In many cases, it will not be possible to run the Processing Element (PE) at 450 MHz. Thus, it is either necessary to run the HBM slave ports synchronously at a lower clock frequency, or alternatively, to perform clock domain conversion (CD) by inserting a SmartConnect IP between the PE and HBM. In the latter case, it is also possible to use *different* data-widths and protocols on the PE-side, and also let the SmartConnect perform data-width (DW) and protocol conversion (PV), as required. To assess the performance impact of the different options, we analyze four configurations, ranging from no SmartConnect, to a SmartConnect which performs all three conversions (CD+DW+PV). The four configurations are shown in Figure 2. Other configurations - for example a SmartConnect IP with clock domain and protocol conversion - are also possible, but yield no additional performance data as they are covered by the other four options and are thus omitted.

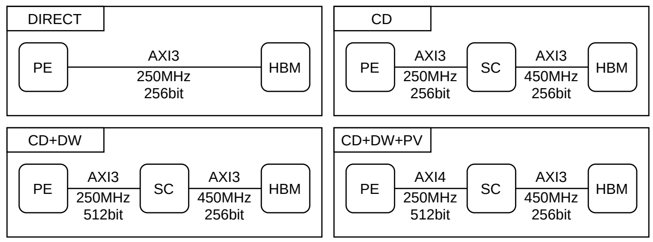


Fig. 2. The four analyzed configurations for connecting the PE with the HBM. The lines represent AXI connections, and are annotated with the protocol version, clock frequency and data width. The configurations are named based on the conversions (CD: clock-domain-, DW: data-width-, PV: protocol-version-) employed.

The rest of this section will compare the performance in various criteria of these four configurations. The goal of this evaluation is to find the best configuration for our specific application scenario. It does *not* strive to be a general evaluation of HBM performance. Thus we focus this evaluation on the *random* access performance, as this is the access pattern used by our hash-join architecture (see Section 4). All results shown use only one AXI3 slave port. As we do *not* use the AXI crossbar, the HBM ports are completely independent, and the performance scales *linearly* when using multiple ports in parallel.

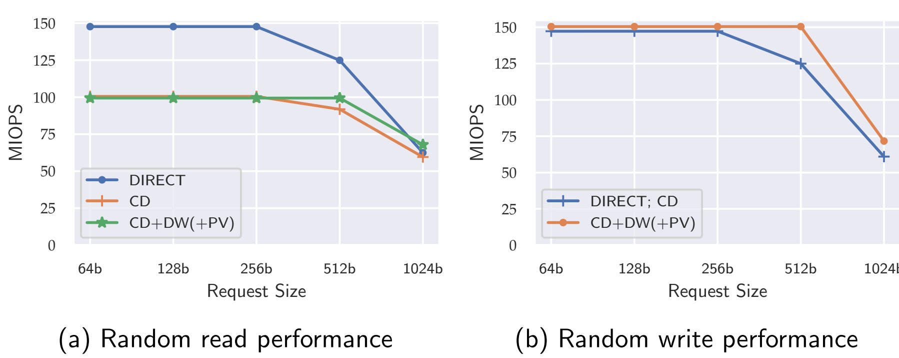


Fig. 3. HBM random access performance in I/O operations per second for the different configurations at a clock frequency of 250 MHz.

Figure 3 shows the random access performance for the different configurations. The results indicate a large performance benefit for small read accesses (up to 256 bit) for the `DIRECT` configuration, compared to all other configurations. This is likely caused by the additional latency introduced by the SmartConnect as shown in Figure 4. For small write accesses, all configurations achieve a similar performance, as these are not affected by this additional latency. For wider memory accesses the performance depends less on the latency, but more on the maximum throughput of the AXI connection. Thus, the configurations `DIRECT` and `CD` perform worse because in both of the cases where the PE-side has a data-width of 256 bit, and is running at only 250 MHz. Therefore, the maximum throughput is lower than the theoretical maximum of the HBM slave ports.

This shows that for our application, the best solution is omitting the Smart-Connect (`DIRECT`), as we only use memory accesses with a size of 256 bit (see Section 4).

Finally, Figure 5 shows that for this scenario of small random accesses, the peak performance is reached around 200 MHz. Thus, for our design (see Section 4), a clock frequency of 250 MHz suffices to achieve maximum memory performance.

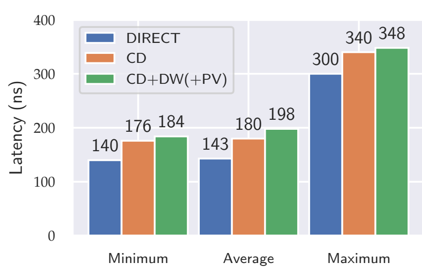


Fig. 4. Read access latency for the different configurations. The addition of an AXI SmartConnect for CD/DW/PV increases the latency considerably.

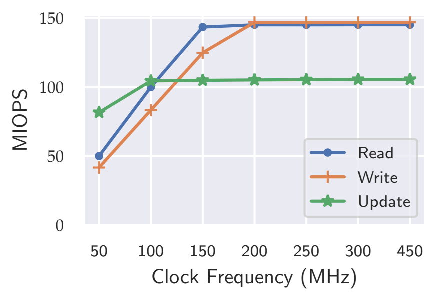


Fig. 5. Random access performance for clock frequencies ranging from 50 MHz to 450 MHz using the `DIRECT` configuration and 256 bit wide memory accesses.

3 Hash Join

The join operation is common in relational databases for analytical processing, and is frequently used in data warehouses and data centers [3, 4]. Its purpose is the merging of two relations into one, based on a shared key (Equi-Join). A database query can include multiple join operations to combine information from multiple tables. Such queries are especially prevalent in Data Warehousing [9], where typical queries join a *fact table* with multiple *dimension tables* in order to compute analytical aggregations. The fact table holds very fine-grained data (e.g., each entry could correspond to one order in a retail store data warehouse), which causes the fact table to typically be orders of magnitude *larger* than the dimension tables. The dimension tables include descriptive attributes of the fact table, and as such need to be joined with the fact table in order to answer analytical queries. While many join implementations exist, we chose to focus on the *no-partition hash join* [1], as this is a commonly used and well understood parallel hash join implementation.

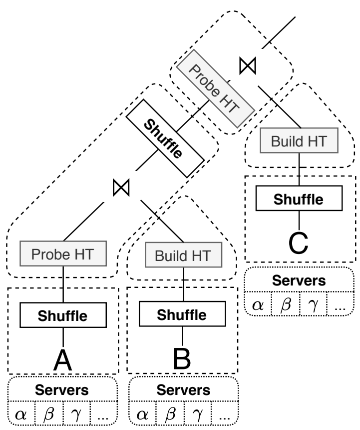


Fig. 6. Traditional distributed database join of the tables `A ⋈ B ⋈ C` distributed across multiple servers, requiring shuffles

Fundamentally, the join operation consists of two steps: (1) building a hash table on the join key of the smaller *dimension table*, and (2) probing that hash table with the join key of the larger *fact table*. In a query with multiple joins, a hash table is built on each dimension table such that the fact table can be probed into each of the hash tables to produce the query result. On a single-node system this approach works well if the main memory is sufficiently large to hold the database tables, hash tables and intermediate join result. However, for large databases (such as data warehouses), where tables are partitioned across multiple servers in a cluster, the join cannot simply be processed in parallel on each server without network transfers.

Traditionally, for processing such a distributed join, it must be ensured that tuples which share the same join-key from two tables are processed on the *same* server. This step is referred to as *shuffling* (or re-partitioning), and shown in Fig. 6 for three tables partitioned across a number of servers $\alpha, \beta, \gamma, \ldots$. Given the high chance that two tuples with the same join-key do *not* reside on the same server, distributed joins often incur heavy network communication for shuffling, which typically dominates the overall runtime of the join query. The bandwidth requirements for shuffling increase further when the distribution of tuples in the shuffling step is not uniform, as some servers then receive considerably more data than the others [8]. This *skewed* scenario leads to high ingress congestion, which results in low overall system performance and utilization.

As an alternative, we propose to execute the hash join following the *In-Network Processing* (INP) paradigm, for which we realize a high-performance hardware-accelerated INP-capable switch that is able to perform the two steps of the hash join *directly* on data flowing to the switch, without the need for shuffles.

Figure 7 gives a complete example of both the hash join algorithm itself, as well as the INP realization. It also foreshadows some of the design decisions for

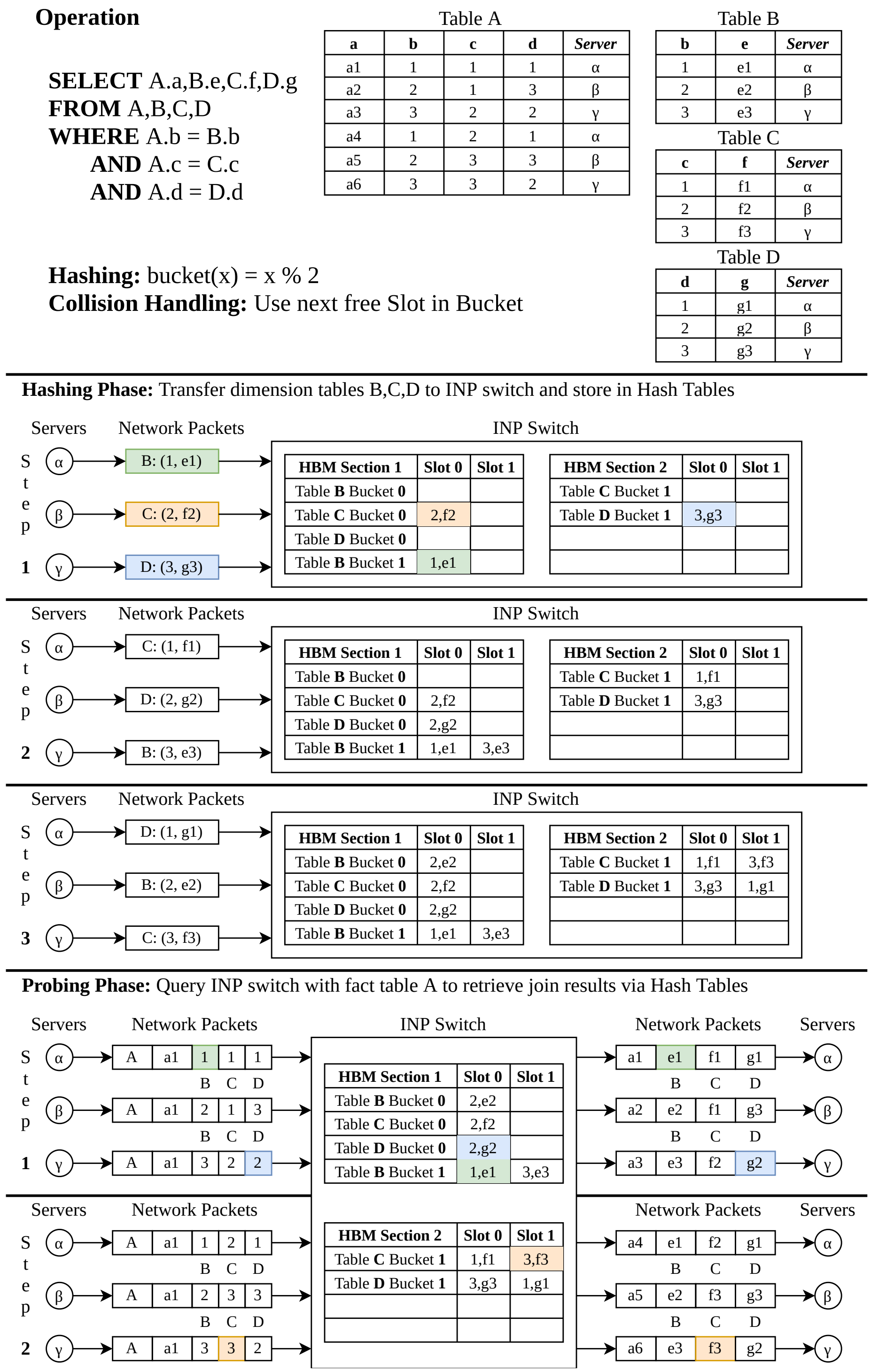


Fig. 7. Sample INP-style hash join over four tables, with data distributed over three servers.

the microarchitecture of the INP switch, e.g., the use of HBM to hold the hash tables, further described in Section 4. Note that, for clarity, we use the foreign keys `A.b`, `A.c`, `A.d` here explicitly in an SQL `WHERE` clause, instead of employing the `JOIN` keyword. In this 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 of the fact table `A`.

In the *Hashing Phase*, the three servers α, β, γ transfer the required contents of the tables to-be joined 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. During this transfer, the hash tables are built: For each key value, the hash function determines a *bucket* where that tuple is to be stored in. In the example, the hash function distributes the tuples across just two buckets per database table. The buckets for each table are spread out across multiple memories for better scaling and parallelism (further explained in Section 4). Hash collisions are resolved by having multiple *slots* within a bucket, and using the first available slot to hold the incoming tuple. Running out of slots in a bucket is an indicator that the switch memory is insufficient to perform the specific join in INP mode, and a fallback to the traditional join has to be used instead. The issue of these bucket overflows is further discussed in Section 4.2 and Section 6. In Figure 7, the buckets and slots used for the tuples incoming over the network are highlighted in the same colors.

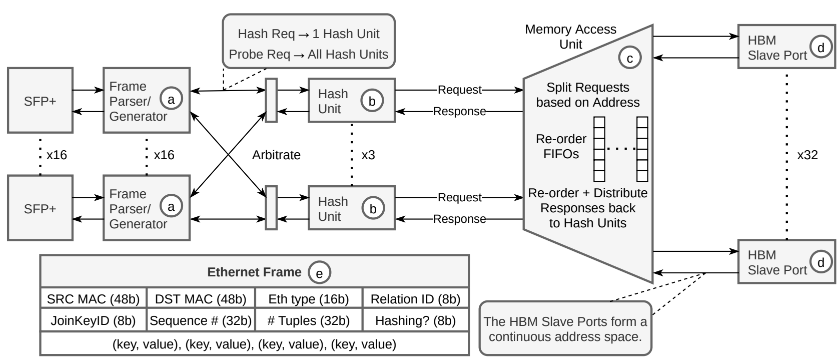


Fig. 8. Full system overview of the proposed INP Hash Join implementation. Hash and probe requests from the 10G Ethernet ports (a) are forwarded to one of the Hash Units (b). Each Hash Unit is responsible for creating one Hash Table. The Hash Units have to access the 32 HBM channels (d), which is realized through specialized arbitration units (c) that ensure routing. The auxiliary units are not shown. For the contents of the Ethernet frame (e), only the relevant parts of the header are shown.

After the hash table has been built in the switch from just the columns of the dimension tables `B`, `C`, `D` required for this specific join, the larger fact table `A` is streamed in. As it, too, has been distributed over all of the servers $\alpha \dots \gamma$, this 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 hash table (bucket and slot), and returned by network 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.

4 Architecture

The architecture presented here is a new design based on an earlier prototype, which was implemented on a Virtex 7 FPGA, attached to two DDR3 SDRAM channels [8]. That older architecture is capable of handling 20 Gbit/s of hash join traffic, but is not able to scale beyond that, as the dual-channel memory can not keep up with the random access demands. We present here a completely new implementation of the same basic concepts, but redesigned to target a newer UltraScale+ FPGA and fully exploit HBM. Figure 8 shows a high-level overview of the system.

The new design can be broken down into four separate parts, which deal with different aspects of the processing flow. Network operations are performed decentralized in network frame parsers (Figure 8a). The incoming frames are parsed into hash and probe requests, which are then forwarded to the actual hash units (Figure 8b), where each unit is responsible for managing one hash table. Next in the processing chain is the memory access unit (Figure 8c), which coordinates the memory accesses coming from the hash units and distributes them to the attached HBMs (Figure 8d). The results of memory accesses, namely the join tuples in the Probing Phase of the algorithm, are collected, encapsulated into network frames (8a), and returned to the server responsible for that part of the distributed join result.

The Ethernet interfaces and frame parsers (Figure 8a) run at a clock frequency of 156.25 MHz to achieve 10 Gbit/s line rate per port, while the rest of our design runs at 250 MHz. As discussed in Section 2 this is sufficient to achieve the maximum performance for the HBM in our scenario.

4.1 Network Packet Processing

The BittWare XUP-VVH card used in our experiments, contains four QSFP28 connectors, which can provide up to 100 Gbit/s each through four 25 Gbit/s links in one connector. We use these links independently in 10 Gbit/s mode, allowing in a maximum of 16 10 Gbit/s Ethernet connections. The data from each link is transferred over an AXI Stream interface as 64 bit per clock cycle of a 156.25 MHz reference clock. However, the receiving unit must not block. Hence, the units processing the Ethernet data have to be *lock-free*.

To keep things simple in the FPGA, we directly use Ethernet frames and omit higher-level protocols. The frame parsers (Figure 8a) can parse the frames sequentially without needing any inter-frame state. Each frame starts with the required Ethernet information, source and destination MAC, as well as the

Ethernet type, followed by a hash-join specific, custom header. This header contains information to identify the purpose of the frame (hash or probe), the relation the frame belongs to, the number of tuples in the frame, and a sequence number. The body of the frame contains a number of tuples. For a join with four relations `A`, `B`, `C` and `D`, each tuple contains eight 32 bit numbers. In the hashing case, the tuple contains only the key and value of the tuple to hash. In the probing case, the first four values indicate the primary key (PK) of `A`, followed by the foreign keys (FK) of the other relations (three 32b words in total, one for each dimension tables used in this example). The following four values are the corresponding values, which are actually used only in Probe replies. This common structure for both requests and replies, with place holders for the actual result data, was chosen since it keeps the ingress and egress data volumes balanced.

The frame parser retrieves the tuples from the network port and places them in queues to be forwarded to the hashing units. A frame is dropped if the queue is not sufficiently large to hold the entire frame. This avoids a situation where the input channel would block. However, no data is actually lost: The accelerator recognizes this case and explicitly *re-requests* the dropped data from the server, identified by the sequence number and relation ID. When using eight 10G Ethernet connections, about 0.08% of all requests are re-requested in this manner.

In the other direction, probe results are accumulated in a queue and sent back to the source of the request in the same format as described before.

4.2 Hash Table

The hash tables are the core of the algorithm. To keep up with the demands of the network, the hash units (Figure 8b) are highly throughput optimized. Accordingly, the goal of this implementation is to keep the HBM slave ports as busy as possible, as the random access performance is the limiting factor.

To clarify this, the following section presents a brief introduction to the hash table algorithm employed. The `insert` algorithm requires four steps: (1) find the bucket corresponding to the requested key, (2) request the bucket, (3) update the bucket if an empty slot exists, and (4) write back the updated bucket. Hence, every `insert` requires one bucket read and one bucket write. If the bucket is already full, a bucket overflow occurs, and an extra resolution step is necessary to avoid dropping the data (see below).

The `probe` step is simpler and requires only steps (1) and (2) of the `insert` algorithm, with an additional `retrieve` step which selects the correct value from the slots inside the bucket.

Performance-wise, step (1) is of paramount importance. A poorly chosen mapping between the keys and the values results in lower performance, as either the collision rate increases, or the HBM slave ports are not utilized fully. Fortunately, latency is not critical at this point allowing a fully pipelined design.

Like any hash table implementation, this implementation has to deal with collisions. As each HBM slave port is relatively wide (256 bit for HBM, see Section 2) compared to the key and value tuples (32 bit each in our case), the natural collision handling method uses multiple slots per bucket. The number of slots per bucket can be chosen relatively freely, but as the benchmarks performed in Section 2 show, a bucket size corresponding to the width of one HBM slave port is optimal.

For the specific case of database joins, it turns out that complicated bucket overflow resolution strategies are not necessary, as the *nature* of the data itself can be exploited instead: The keys used for accessing the hash table are actually the database primary and foreign keys. This means that the keys are usually just numbers ranging from 0 to $N-1$, with N being the size of the relation. This is especially common in read-only analytical database systems. Accordingly, the values can simply be distributed across the available buckets by using a simple modulo with the hash table size as hash function. For other applications, where this is not the case, and the keys span a larger space, tools such as [13] can be used to generate integer hash functions with good distribution behavior. In our scenario, buckets will only overflow when the entire hash table is already full anyway, resulting in an out-of-memory condition for this specific join.

The hash tables are placed interleaved in memory (see Figure 7) to let every hash table use all of the available HBM slave ports. The design uses one Hash Unit per hash table in the system, leading to three Hash Units in the system for the proposed evaluation in Section 5.

4.3 Memory Access

However, the throughput-optimized spreading of data across all available memory leads to the next problem: The memory accesses from the different hash units have to be routed to the corresponding HBM slave ports (Figure 8d). Connecting the 32 available HBM slave ports requires additional logic. The naive approach of using the vendor-provided interconnect solutions, which use a crossbar internally, is not feasible. First of all, this approach would *serialize* the majority of requests, making it impossible to fully utilize the HBM slave ports. Secondly, the interconnect has to connect the three master interfaces of the Hash Units with the 32 slave interfaces of the HBM, which results in a large and very spread-out layout, due to the crossbar design of that interconnect. Such a design fails to be routed at the high clock frequencies required to achieve optimal HBM random access performance. Hence, a special unit is needed that arbitrates between the masters, and allows efficient access to all HBM slave ports, while remaining routable.

The proposed design, shown in Figure 8c, takes memory requests from the hash units and places them in separate queues. The following memory selection step requires two stages to allow routing on the FPGA even with 32 slave ports: The first step splits the incoming requests onto four queues based on their address. Each splitter section handles the access to a sequential section of memory, which corresponds to the placement of the HBM slave ports on the FPGA. The second step then selects a request from the incoming queues using a fair arbiter, and forwards the request to the corresponding HBM slave port.

For the return direction, the out-of-order answers from the HBMs have to be re-ordered, as the Hash Units expect the answers to be in-order with its requests.

This is done by keeping track of the order in which requests have been forwarded to the HBM slave ports. When answers arrive, this queue is then used to ensure that the answers are returned in the same order as they have been requested in.

5 Evaluation

The evaluation of the system focuses on two aspects: (1) The scaling of the system itself across a number of ports, and (2) the performance compared to a classical distributed database hash join system with multiple worker servers.

5.1 System Performance

The proposed system, hereafter referred to as NetJoin, is evaluated using the TaPaSCo [7] FPGA middleware and SoC composition framework to generate bitstreams for the BittWare XUP-VVH board. The Xilinx UltraScale+ VU37P device on the board features 8 GB of HBM and up to 2.8 million "Logic Elements" in a three chiplet design.

Overall, we use only a fraction of the available resources on the VU37P. The biggest contributor is our PE with about 17% of the available CLBs. A detailed list of the resource usage of the PE, HBM, and the SFP+ connections is shown in Table 1.

Table 1. Resource Usage for the PE, HBM and SFP+ controllers. The percentage of the total available resources of this type on the VU37P FPGA is given in parentheses.

	LUTs	Registers	CLBs	BRAMs
NetJoin PE	135k (10.39%)	197k (7.56%)	28k (17.03%)	30 (0.33%)
HBM Controller	1.5k (0.12%)	1.6k (0.06%)	0.5k (0.33%)	0
SFP+ Interface	19k (1.48%)	31k (1.19%)	4.98k (3.05%)	0

The following benchmarks use a three table join scenario, where one fact table `A` is joined with three dimension tables `B`, `C` and `D`. The first benchmark compares the scaling of the system when varying the number of network interfaces. The size of the dimension tables is kept at 100×10^6 elements, while the fact table has 1.0×10^9 elements. The results, presented in Figure 9, show that the system scales linearly up to six ports, and slightly less than linearly for seven ports. This indicates that up to six ports, the system is able to fully keep up with the line rate of 60 Gbit/s. Figure 10 shows the scaling for the phases (Hashing and Probing) separately.

5.2 Baseline Comparison

The software baseline not using INP is executed on eight servers, each fitted with an Intel Xeon Gold 5120 CPU @ 2.2 GHz and 384 GB of RAM. The servers are

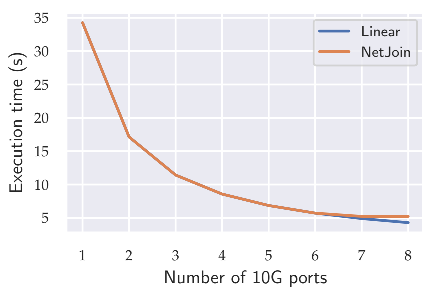


Fig. 9. Scaling of NetJoin performing three joins with table `A` having 1.0×10^9 and `B`,`C`,`D` having 100×10^6 elements. The system scales linearly up to six ports, and slightly slower up to seven ports, where the memory links become saturated and no further scaling is observed.

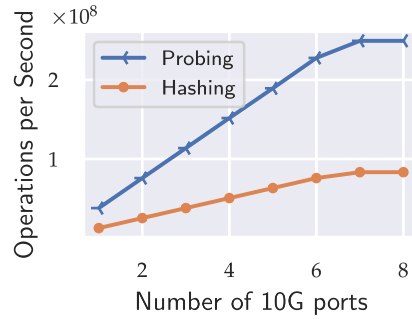
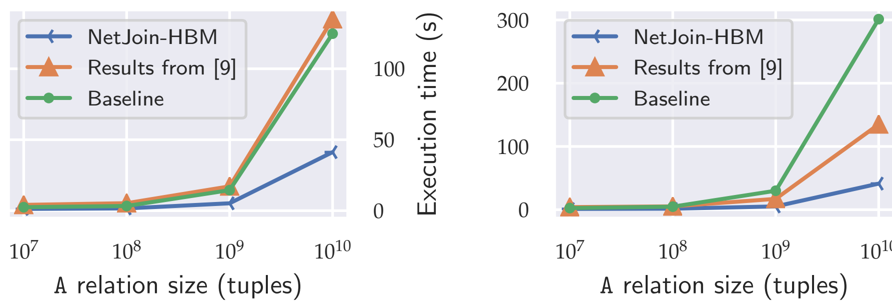


Fig. 10. Hashes and probes per second in NetJoin-HBM for the given number of ports. Unsurprisingly, the probe operation is up to three times faster, as it requires only one memory read to perform. Hashing on the other hand, requires one read and one write per operation.

connected via 10G BASE-T using CAT 6 RJ45 cables to a Stordis BF2556X-1T programmable switch. Furthermore, a master server with the same hardware is used to control execution. The baseline system operates with eight SFP+ ports on the switch, for a total maximum bandwidth of 80 Gbit/s.

As before, all the experiments join a fact table `A` with three dimension tables `B`, `C` and `D`. The first experiment compares the scaling behavior of both approaches for different fact table sizes with fixed dimension tables. The results in Figure 11a show that even for small sizes of the relation `A`, the NetJoin approach is already *two times faster* than the software baseline. As the intermediate results are relatively small here, the shuffling step does not incur a big overhead in the software baseline, but is nonetheless noticeable. For larger sizes, however, the advantage of avoiding the shuffling steps by INP becomes more pronounced. In the tested range, NetJoin-HBM is already *three times faster* than the baseline, and its execution time grows slower than the non-INP baseline. Note that these results represent the *optimal* scenario for the software baseline, as the keys are *equally* distributed across all servers. Also, remember that the older DDR3-based INP design from [8] cannot even keep up with the current software baseline, which now uses considerably more powerful hardware than available in the earlier work.

In the skewed case, where only a few servers have to do the majority of the processing, the advantage of the INP approach over the baseline grows, even on the older INP accelerator. In this scenario, one server receives 34.6% of the data, a second server receives 23.1% of the data and the rest receives 15.4%, 10.2%, 6.9%, 4.6%, 3% and 2% respectively. As described in Section 3, this leads to high ingress congestion and much sequential processing. The INP approach, on the



(a) Keys are *equally* distributed across all servers.

(b) Key distribution is *skewed*.

Fig. 11. Comparison of NetJoin-HBM, the baseline running in software on eight servers, and the older DDR3-based architecture from [8] which runs at 20 Gbit/s. Three joins are performed with tables `B`, `C` and `D` at a static size of 100×10^6 tuples. The size of relation `A` is varied.

other hand, does not suffer from this issue, and is able to process the data in the same way as before. The results in Figure 11b confirm this observation. For very small sizes of `A`, the NetJoin-HBM approach remains about two times faster. Increasing the size of `A` shows that the skewed case is handled poorly by the software baseline. At the highest tested table size, NetJoin-HBM is already 7.2× faster than the baseline for the skewed scenario. For comparison, the unskewed baseline is about 2.3× faster than its skewed counterpart.

6 Conclusion and Future Work

This work is motivated by the observation that the performance of an existing FPGA-based INP switch architecture is mainly limited by the memory access performance. To overcome this bottleneck, we propose an enhanced architecture, which uses HBM instead of DDR memory. We show that by exploiting HBM, we achieve more than *three times* the throughput of an older DDR3 SDRAM-based INP accelerator for joins [8].

The design is currently limited by the size of the available HBM memory. As most HBM-FPGA boards also still include DDR4-SDRAM, it would be possible to increase the amount of available memory by using *both* memory types. However, this would be a more complex architecture, due to the access differences between HBM and DDR, and the necessity to partition the data between HBM and DDR.

Another issue of our current architecture is the possibility for write after read errors. Considering the average response time of the HBM is 35.75 cycles, there can be cases where a *second* read to the same bucket occurs *before* the earlier write has been completed. The probability of this happening can be calculated based on the probability of a collision happening depending on the number of outstanding requests, which is about 5.45×10^{-6} [10]. For many data-analytics applications, this error rate will be acceptable. However, in applications where this chance for error can not be tolerated, a possible solution is the introduction a cache-like approach for keeping track of the in-flight requests and handling these avoiding the hazards (similar to MSHRs in caches).

References

1. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core cpus. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011. pp. 37–48. ACM (2011)
2. Blöcher, M., Ziegler, T., Binnig, C., Eugster, P.: Boosting scalable data analytics with modern programmable networks. In: Proceedings of the 14th International Workshop on Data Management on New Hardware. DAMON '18, Association for Computing Machinery, New York, NY, USA (2018)
3. DeWitt, D.J., Katz, R.H., et al.: Implementation techniques for main memory database systems. SIGMOD Rec. **14**(2), 1–8 (Jun 1984)
4. Dreseler, M., Boissier, M., Rabl, T., Uflacker, M.: Quantifying TPC-H choke points and their optimizations. Proc. VLDB Endow. **13**(8), 1206–1220 (2020)
5. Firestone, D., Putnam, A., et al.: Azure accelerated networking: Smartnics in the public cloud. In: Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation. p. 51–64. NSDI'18, USENIX Association, USA (2018)
6. Gustavo, A., Binnig, C., et al.: Dpi: the data processing interface for modern networks. Proceedings of CIDR 2019 (2019)
7. Heinz, C., Hofmann, J., Korinth, J., Sommer, L., Weber, L., Koch, A.: The tapasco open-source toolflow. In: Journal of Signal Processing Systems (2021)
8. Hofmann, J., Thostrup, L., Ziegler, T., Binnig, C., Koch, A.: 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)
9. Kimball, R., Ross, M.: The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling. John Wiley & Sons, Inc., USA, 2nd edn. (2002)
10. Preshing, J.: Hash collision probabilities. `https://preshing.com/20110504/hash-collision-probabilities/` (2011)
11. Rödiger, W., Mühlbauer, T., Kemper, A., Neumann, T.: High-speed query processing over high-speed networks. Proc. VLDB Endow. **9**(4), 228–239 (Dec 2015)
12. Sapio, A., Abdelaziz, I., et al.: In-network computation is a dumb idea whose time has come. In: Proceedings of the 16th ACM Workshop on Hot Topics in Networks. p. 150–156. HotNets-XVI, Association for Computing Machinery, New York, NY, USA (2017)
13. Wellons, C.: Hash function prospector. `https://github.com/skeeto/hash-prospector` (2020)
14. Zilberman, N., Audzevich, Y., Covington, G.A., Moore, A.W.: Netfpga sume: Toward 100 gbps as research commodity. IEEE micro **34**(5), 32–41 (2014)