Energy-Efficient Hash Join Implementations in Hardware-Accelerated MPSoCs

Sebastian Haas, Gerhard Fettweis
Vodafone Chair Mobile Communications Systems
Center for Advancing Electronics Dresden (cfaed)
Technische Universität Dresden, Germany
sebastian.haas@tu-dresden.de, gerhard.fettweis@tu-dresden.de

ABSTRACT

Join is one of the most important operators in database query processing. Its research progressively focuses on hardware-conscious implementations since single-threaded performance improvements of general-purpose processors will slow down in the next years. SIMD extensions, multi-threading as well as multi-core processors may further lead to performance advantages. Besides that, multiprocessor system-on-chips (MPSoCs) are a suitable platform to keep up with high-performance processors while providing an up to three orders of magnitude lower power consumption.

In this paper, we study the implementation of hash join algorithms on MPSoCs and exemplarily employ the *Tomahawk4* chip. Tomahawk4 integrates four processing modules each equipped with tightly-coupled SRAM as well as an instruction set extension tailored to hashing algorithms. An external DRAM serves as shared main memory and can be accessed by DMA transfers. We aim to best exploit the architecture and to adapt the algorithms to the MPSoC. Hence, we compare two hash table designs according to their memory accesses and investigate the performance impact of the additional hashing instructions. Furthermore, the MPSoC platform allows power measurements with different clock frequencies and supply voltages to find the configuration with highest energy savings. Our experiments on the MPSoC show that four database-specific cores outperform a standard RISC CPU by up to factor 5 while consuming less than 200 mW.

1. INTRODUCTION

Nowadays, tuning database algorithms to the underlying hardware is a frequently studied research topic. Although processor performance has dropped from about 50 % to about 20 % per year [15], improvements originate from hardware-specific features such as multi-threading and SIMD towards many-core systems. General-purpose processors like Intel Xeon, AMD Opteron or Sun UltraSPARC deployed in modern database systems provide the required throughputs by exploiting the mentioned hardware features. Besides these high-performance CPUs, GPUs are also well suitable for query processing due to high parallelism and superior memory bandwidths [18]. They only suffer from the limited amount of main memory and time-consuming copy processes between CPU and GPU.

Besides the performance requirements, power consumption becomes important as well, especially when considering applications such as Mobile Edge Computing [17] where data processing is shifted closer to the user and thus away from data centers which provide (almost) unlimited power supply. The high energy efficiency of multiprocessor system-on-chips (MPSoCs) makes them an interesting platform to accelerate database operators. MPSoCs comprise domain-specific hardware accelerators such as ASICs and ASIPs with high instruction and data level parallelism. These cores usually run at clock frequencies below 1 GHz to provide low power consumptions. In this work, we are heading towards database-specific accelerators integrated in an MPSoC which are tailored for energy-efficient hash joins. Initially, MPSoCs require higher development costs than CPUs but benefit from flexibility regarding core-to-core communication, latency, and reliability. However, the memory bottleneck is still a challenge as we will report in this paper.

We choose the join operator since it is one of the most important operators used in relational query processing. Applications such as data mining [9] and information retrieval [28] benefit from efficient join implementations. In this paper, we focus on the hash join algorithm for two main reasons: 1) Prior work [21] has shown on a set of data analytic workloads that hash-table-based indexing operations employed in hash joins are the largest single contributor to the overall execution time. 2) In our previous work [1], we reported that the underlying hashing algorithms are worthwhile for hardware acceleration and will improve the hash join performance. In particular, our contributions can be summarized as follows:

- We describe two hash join algorithms which combine state-of-the-art solutions regarding partitioning of input data and hash table design.
- We adapt the algorithms to best exploit our MPSoC architecture which includes hardware-accelerated processing elements.
- We provide performance and power measurements of the algorithms on the Tomahawk4 MPSoC. We are then able to compare the selected algorithms by investigating the effect of number of cores and additional hashing instructions.

The remainder of the paper is organized as follows: Section 2 provides the necessary background of the join operator and discusses state-of-the-art implementations and solutions. Section 3 presents the MPSoC platform used for evaluation and explains the hash join implementations. Afterwards, the experimental results are shown in Section 4. Finally, we conclude the paper in Section 5.

2. BACKGROUND AND RELATED WORK

This section provides the necessary background information about join algorithms and summarizes state-of-the-art implementations.

2.1 Join Operator

The join operator is a common operation used in SQL queries. In this paper, we focus on the implementation of the equi-join or inner join. The query can be expressed as

```
SELECT *
FROM R, S
WHERE R.A = S.B
```

It returns all matching tuples of relations R and S where the respective values of attributes A and B are identical. Basically, the three most known implementations are 1) nested-loop join, 2) sort-merge join, and 3) hash join. Since the nested-loop join simply iterates over the two relations by using two nested loops, it has a time complexity of $O(|R||S|)$. Even though partitioning and parallelization is trivial, it always results in a poor performance. The sort-merge join can be computed by a single run over both sorted relations. The sorting problem defines the complexity which is at least $O(n \log n)$ for a merge-sort algorithm [20].

Differently, the hash-based join can operate on unsorted data. In a first build phase, the smaller relation, say R, is mapped into a hash table by using a hash function. The actual join is performed in a subsequent probe phase where tuples from S are compared with the records from R stored in the hash table. An optimal hash function is adapted to the data set and should reduce the number of collisions to provide short access times on the indexed tuples. A collision in the hash table occurs when two different keys obtain the same hash value and, thus, are mapped to the same bucket. Several works already studied the efficiency of different hash functions [4, 27]. However, we want to focus on the overall performance constraint by the system and, hence, use the simple bit selection described in Section 3.2. The hash join may outperform the sort-merge join in a cache based multi-core system as reported by [5, 19, 25]. However, the authors also conclude that their sort-merge join implementation executes faster with higher SIMD widths, i.e., with more appropriate hardware features as well as limited memory bandwidths.

Our MPSoC evaluation platform provides only small local memories which are tightly coupled to the processors (more details in Section 3.1). Hence, the main memory access time significantly impacts the overall execution time. Even though, we decided to focus on the implementation of hash joins since the cores support hardware-accelerated hashing instructions.

2.2 State-of-the-Art

A lot of research has been done on the implementation of hash joins in multi-core systems. Most of the work exploit general-purpose CPUs and adapt the hash join algorithm according to the underlying processor architectures and memory subsystems [6, 8, 22]. Especially the authors in [6] conclude that hardware-conscious algorithms perform significantly better than the hardware-oblivious counterparts.

Efficient partitioning methods are provided by [10, 24, 29] and minimize cache misses when accessing the hash table. For instance, Manegold et al. [24] show a radix-clustering algorithm to partition the input relations which we also adapted to parallelize the hash join on the MPSoC. However, Blanas et al. [8] report that simple hash joins without partitioned input relations are very competitive to other more complex implementations. Note that for our work, it is essential to partition the data and use distributed hash tables without synchronization since the high-latency main memory access in the MPSoC platform prevents the usage of latched hash table buckets. Instead, we took another idea of [8] where bucket entries are connected by linked lists which makes the algorithm robust for an unexpected high ratio of collisions in the hash table.

The aforementioned approaches do not incorporate hardware extensions such as SIMD or instruction parallelism in multi-core systems. Nevertheless, Balkesen et al. [5] and Kim et al. [19] run their hash join and sort-merge join algorithms on multi-core Intel processors supporting simultaneous multi-threading while the sorting step was advanced by 128-bit SSE and 256-bit AVX instructions. Although the sort-merge approach obtains an advantage by additional instructions, their results reveal that hash-based join algorithms can still be around 2× faster than sort-merge joins.

The introduced massive parallelism in graphics processors can be easily applied for GPU-based joins [18, 23, 31]. Amongst other join algorithms, [31] implements a hash join which contains a partitioning step followed by a join step. The approach is similar to the one used in our work, where partitioning is based on hash values and a histogram is used to obtain sorted tuples in the input relations.

Besides the broad range of join implementations on general-purpose platforms, the attention to low-power chips which integrate application-specific processing elements seems to grow as well. For instance, the authors in [30] consider basic database operators such as sorting and aggregation on the Tomahawk2 MPSoC. They only use general-purpose processing elements but conclude that emerging hardware features will improve the performance of query operators. The work in [12] takes this idea and integrates database-specific processing elements into an MPSoC. Compared to state-of-the-art platforms, their manufactured Tomahawk3 chip achieves an 96× energy improvement for a scan benchmark. In this paper, we also follow this approach and especially study the hash join on the next Tomahawk generation – Tomahawk4.

Gedik et al. [11] developed the so called CellJoin which focuses on a windowed stream join operator executed on the Cell processor. The Cell processor is similar to our used MPSoC platform and also includes processing elements (Synergistic PEs – SPEs) equipped with a local memory. In contrast to the special hashing instructions provided by Tomahawk4, the SPEs only support general-purpose instructions on 128-bit SIMD registers. Even though, the CellJoin outperforms an SSE implementation on a dual-core Intel Xeon processor by factor 8.3.

In summary, this paper provides the next steps to exploit state-of-the-art algorithms (in particular hash join) as given by the literature and to apply them to low-power MPSoCs with application-specific hardware accelerators. The findings are essential to build an energy-efficient system for database query processing which is capable to offload data-intensive operations from high-performance processors.

3. IMPLEMENTATION

After reviewing state-of-the-art join realizations in Section 2, this section presents our hash join implementation. Therefore, we first discuss the evaluation platform to understand the design restrictions. This is followed by detailed descriptions of the hash join algorithms.

3.1 MPSoC Platform

Before studying the actual hash join implementation, it is helpful to understand the requirements and constraints given by the underlying hardware. Hence, we first present the MPSoC platform which is used for all evaluations. The Tomahawk4 MPSoC [14] is a heterogeneous multi-core platform combining SDR capabilities and database processing. It relies on the Tomahawk concept [3] which provides a fully programmable many-core system with runtime management. Thereby, the control flow of the application is mapped on the system. The actual computation is performed by multiple processing modules (PMs) which may include one or more processing elements (PEs). A PE might be a common RISC, DSPs, or specialized hardware accelerators (ASIC, ASIP). In addition, each PM has local SRAM which is tightly coupled to the PEs.

In particular, Tomahawk4 contains four PMs each comprising an ARM Cortex-M4 and a Tensilica Xtensa LX5 core, both sharing a local memory with 64 kB for data and instructions each. The concept is aimed for the isolated execution of single tasks in the PMs by using the local memory. However, a shared off-chip DRAM is connected by an LPDDR2 interface. This memory access is only possible by using the direct memory access (DMA) controller integrated in each PM. Furthermore, the Tensilica processor is tailored to the energy-efficient execution of basic hashing operators. We refer to the LX5 as the database accelerator (DBA) which is explained in more detail in Section 3.2.

Task scheduling and power management is provided by a control subsystem including an application core (APP) and the central scheduling unit called *CoreManager* (CM). The APP is a general-purpose CPU (Tensilica 570T) with full MMU which uses a 32 kB cache and the DRAM as main memory. It usually executes the application code which is split into multiple tasks. The CM is an Xtensa LX5 RISC and is responsible for analyzing these tasks according to data dependencies in order to map them to the PMs. This also includes initiating data transfers as well as power management. For our purposes, the CM directly maps the query execution plan to the DBAs (i.e. the hash join algorithm) whereas the APP only allows to compare the parallelized hash join with a single-core cache-based processor.

In Tomahawk4, the power management concept is based on a dynamic voltage and frequency scaling [16]. All four PMs can be individually connected to one of three power supply rails with preset voltages in the range of 0.6 - 1.1 V and predefined clock frequencies between 100 and 500 MHz. An integrated power management controller enables a fast switching between the supply rails. The CM can control the power management by adapting the performance according to the system's workload.

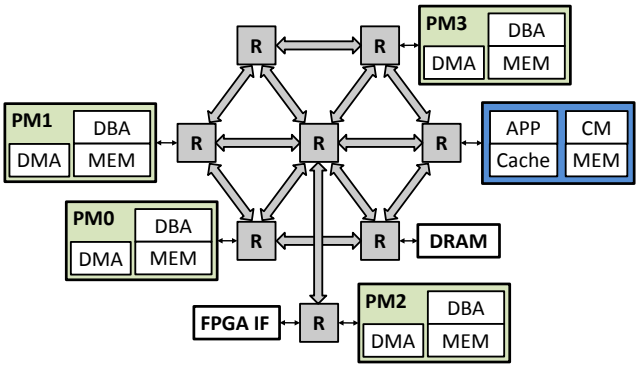

Figure 1: Simplified Tomahawk4 MPSoC platform with 8 routers connected in a hexagonal structure.

All components are connected by a packet-switched hexagonal network-on-chip (NoC). One packet transfers 64 bit of payload per cycle at 500 MHz resulting in a link throughput of 32 Gbit/s. In comparison to a rectangular mesh NoC, it provides higher reliability when NoC links are heavily occupied or faulty [26]. Furthermore, the NoC is built upon a GALS approach (globally asynchronous, locally synchronous) to enable individual clock domains of each module. This requires buffers placed on the interfaces between the modules and the NoC. The FPGA interface connects an FPGA board to access a host PC and to enable chip-to-chip interconnections. Figure 1 depicts the MPSoC focusing on the main components used for database processing.

In summary, the MPSoC provides a complete system including application-specific processing modules to study the performance of different hash join implementations according to processing speeds, memory bandwidths as well as power consumption.

3.2 Database Accelerator

The DBA is built upon an ASIP approach, i.e., the basic LX5 RISC is adapted by 1) an extended instruction set, 2) two load/store units (LSUs) for parallel access to two local data memories, and 3) a 128-bit wide memory interface width for SIMD capabilities. Figure 2 depicts a PM with the DBA, DMA, and the local memory. The concept was already evaluated for a wide variety of database operations [1, 2, 13]. In this paper, we focus on the extended instruction set for hashing algorithms published in [1]. The following paragraphs briefly explain the hashing algorithm and specific instructions.

We choose a hash function which selects b bits out of a 32-bit integer value representing the key in a key-payload tuple. A hash mask denotes the position of appropriate bits. The selected bits from the key are shifted to the right to obtain a b-bit wide hash value. The resulting hash table now contains 2^b buckets. For example, with a given hash mask of 1011_b and a key value of 12 (1100_b), we obtain a hash value of 4 (0100_b). In the following, we assume $b \leq 16$ to limit the size of the hash values and the hash table, respectively.

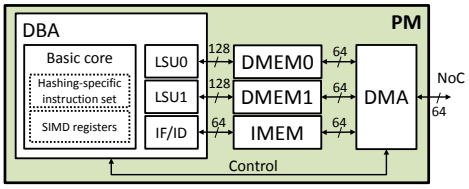

Figure 2: Simplified structure of processing module with database accelerator.

The LX5 core operates on 32-bit general-purpose registers and supports typical arithmetic and logical instructions. The described integer hash function requires the extensive usage of bitwise operators which consume a large number of cycles. To speedup the hash function, we use a developed instruction which combines all bitwise operators and enables to hash a key within one clock cycle. This instruction as well as specific load/store instructions are applied to 128-bit SIMD registers which contain four keys. For an n-fold SIMD approach, speed increases linearly with n and area grows with n^2. Hence, 128-bit wide registers and interfaces are an acceptable trade-off between area and performance. Moreover, the two data memory interfaces allow to load and process on two SIMD registers simultaneously. This again doubles the overall performance. Compared to the pure RISC execution, a final speedup of 1055× can be achieved. We refer to [1] for more details.

3.3 Hash Join

As explained in Section 2.1, we cover the equi-join on two relations R and S. We further assume that both relations with cardinalities $|R|$ and $|S|$ consist of unsorted tuples of 32-bit keys and 32-bit payloads represented as unsigned integers where the keys are used for comparison for the join. Initially, all tuples of R and S reside in DRAM. As mentioned in Section 3.1, the PMs of the MPSoC operate on their local memory and data from DRAM is transfered by using the DMA integrated in each PM. To ensure a continuous execution, we apply the concept of a ping-pong memory. The local memory is split into two arrays: one is used to load chunks from R and S, respectively, while the processor performs on the other one. The purpose of both arrays is switched for the next chunk to establish a pipelined execution.

We presume $|R| \leq |S|$ so that R is target to be mapped into the hash table. Consequently, the hash table has at least $|R|$ bucket entries and will be placed in DRAM as well. This leads to multiple implementation challenges which we have to address.

DRAM access latency: Compared to the local memories, DRAM accesses from the PMs are much more expensive in terms of read latency and throughput. Hence, it is impossible to use a shared hash table for multiple PMs since this would require locks for synchronization. We decide to focus on individual hash tables assigned to each PM.

Memory mapping: Furthermore, the DRAM is only accessible via single DMA transfers and, thus, cannot be mapped into a data memory region of the PMs. For this reason, the central CM assigns the DRAM addresses of hash tables for each PM before runtime.

Input data: We assume to have no prior knowledge about the input data as well as number of duplicates. This also leads to an undefined number of collisions in the hash table and requires an adjustable number of buckets and bucket sizes.

We want to find the concept which best exploits the given MPSoC architectures and fulfills the aforementioned requirements. Hence, we investigate and compare two hash join implementations which basically differ from the hash table structure: 1) a partitioned hash join which uses a hash table build with linked lists, and 2) a hash join based on radix-partitioning with a histogram based hash table. The following subsections explain the implementations in more detail.

3.3.1 Hash Table with Linked Lists

The first hash join approach is similar to [8] and uses a hash table where bucket entries are connected by a linked list (`LL-HT`). A bucket entry stores the key and the pointer (index) to the next tuple of this bucket. This index is used as reference to establish a linked list. Information about the position of the first and last key as well as the number of bucket entries is stored locally in each PM.

The following hash join is described for the execution with p processors (DBAs). The complete build input R is transferred to all p processor. Each processor i ($i = \{1, ..., p\}$) then performs the following steps (cf. Figure 3):

(1) **Hash R:** Iterate over R and calculate the hash values $h(R)$. Chunks of R which fit into the local memories are successively loaded from DRAM. Loading and processing is alternately executed on two arrays in a pipelined fashion as described earlier.
(2) **Insert R to hash table:** Check each tuple in R if it belongs to an assigned range of hash values $h_{i-1} \leq h(R_i) < h_i$. R_i and $h(R_i)$ are now denoted as the applicable tuples and hash values of processor i, respectively. Insert keys of R_i into hash table i which belongs to processor i. This generates p independent hash tables. When inserting the keys, always use a new entry in the hash table. If a tuple is already assigned to the determined bucket, add the current entry's index to the previous inserted tuple of this bucket.
(3) **Hash S:** Iterate over S and calculate the hash values $h(S)$ in the same manner as for R in step (1).
(4) **Join:** Access the hash table with the hash values $h(S_i)$ assigned for processor i, compare the keys in S_i with the found keys from R_i, and store matching tuples. Continuously write back the result to DRAM.

The described partitioning is similar to the radix join from [24] while the radix bits are now selected by the hash function. This ensures that no data dependencies occur between the partitions. Furthermore, the solution is highly scalable due to the non-blocking execution, i.e., the hash table is built in a single run over R. Furthermore, the hash table stores all tuples successively in memory. Except for collisions in the hash table, this results in a single predictable write access per tuple.

Figure 3 also illustrates a sample procedure to fill the hash table with one partition R_i of processor i. The keys of each tuple are hashed by the hash function as described in Section 3.2. In this example, we assume that all hash values are in the range between $h_{i-1} = 0$ and $h_i = 16$ otherwise the corresponding tuples would be omitted from the hash table. The chosen hash mask leads to a collision in the hash table

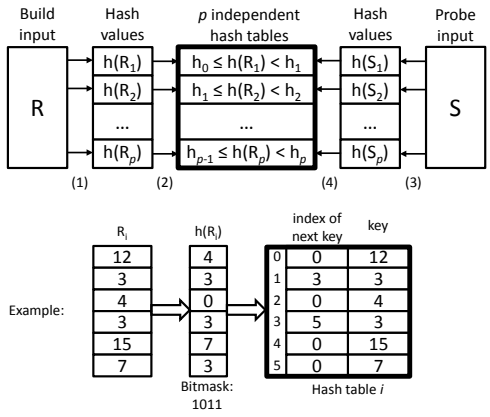

Figure 3: Hash join with linked list in hash table (`LL-HT`) executed on p processors. The example below shows the hashing (1) and the insert (2) step of processor i.

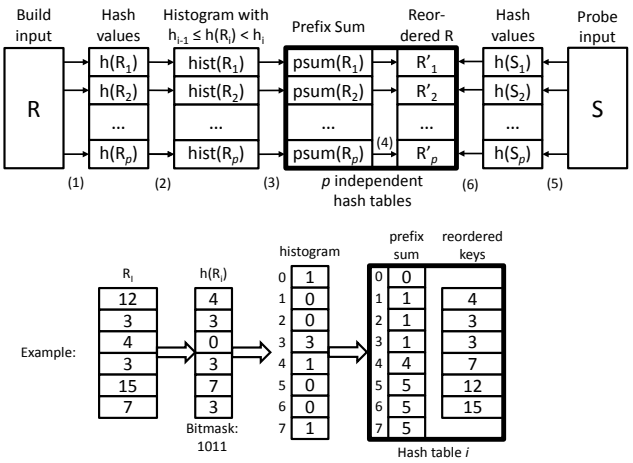

Figure 4: Hash join with histogram-based hash table (`Hist-HT`). The example below shows all steps from hashing (1) to insert (4).

since the keys 3 and 7 are mapped to the same bucket due to identical hash values. Hence in the sample hash table, the bucket with key value 3 at position 1 is linked to the next bucket entry with index 3 to indicate that the next key of this bucket is on position 3.

3.3.2 Histogram-based Hash Table

Our second hash join implementation relies on a preceding partitioning based on radix clustering proposed by the authors in [24] and builds a histogram-based hash table (`Hist-HT`) and a prefix sum similar to [7] with a reordered relation R as presented in [19]. In contrast to our first approach where the number of DRAM accesses are minimized, this hash join implementation requires known input cardinalities before runtime as well as a second run over R. Again, processor i receives all tuples of R from DRAM and performs the following steps (cf. Figure 4):

(1) **Hash R:** Identical to step (1) of the `LL-HT` hash join implementation.
(2) **Determine histogram from R:** Scan all hash values $h(R)$ and select tuples R_i which belong to the assigned range of hash values $h_{i-1} \leq h(R_i) < h_i$ to obtain histogram i.
(3) **Determine prefix sum from histogram:** Iterate over histogram i and obtain the prefix sum for R_i.
(4) **Insert R to hash table:** Again load R and obtain the hash values similar to step (1). Insert tuples of R_i which are in the specified range of hash values. The position in the hash table is determined by the prefix sum calculated for this tuple. All tuples which belong to the same bucket are now successively stored in the hash table. There are now p independent hash tables for all p processors.
(5) **Hash S:** Iterate over S and calculate the hash values $h(S)$ in the same manner as R in step (1).
(6) **Join:** Identical to step (4) of the `LL-HT` hash join implementation.

As can be seen from the algorithm, R is loaded and hashed twice from DRAM. However, due to the additional hashing instructions, determining the hash values is expected to even have the smallest impact on execution time. Additionally, all tuples of one bucket in the hash table are successively located in memory. In comparison to the hash table with linked lists, we expect to obtain a faster join phase when using the histogram-based hash table.

Figure 4 depicts an example hash table and the mapping phase for one partition R_i of one processor. After hashing, the histogram is obtained from the hash values, i.e., hash value 4 appears once, 3 exists three times, etc. The prefix sum determines the position of the keys in the hash table, e.g., key value 12 is mapped to index 4 in the hash table which is selected by the prefix sum at index 4. We again assume that all hash values belong to the applicable range for this partition i.

4. EXPERIMENTS

This section presents our experimental results. After introducing the configured set-up, we show performance as well as power measurements.

4.1 Configuration

We evaluate all hash join implementations explained in Section 3.3 with the previously described Tomahawk4 platform. The DBAs execute the actual algorithm and the CM is responsible for data initialization in DRAM as well as starting the cores. In the following, we again refer to the hash joins as `LL-HT` and `Hist-HT` for the hash table built with linked lists and the histogram, respectively. For comparison, we also execute both hash joins on the APP core. In contrast to the DBAs, the code of the APP only omits partitioning the input relations according to the hash values due to a single-core execution. Moreover, the APP benefits from a direct DRAM address mapping as well as a data and an instruction cache of 16 kB each. Unless otherwise specified, APP, CM, and the DBAs run at 500 MHz with a supply voltage of 1.05 V. For the DRAM, we measured a maximum bandwidth of 12.5 Gbit/s. Our DRAM latency estimation yields about 300 cycles when one single core reads 8 byte without any further traffic on the NoC. This includes to configure the DMA from software (260 cycles) plus 40 cycles round-trip delay to cross the buffers between the modules and the NoC as well as to pass the LPDDR2 interface.

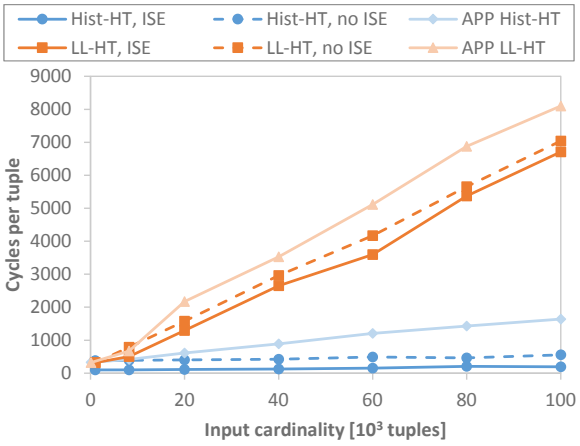

Figure 5: Cycles per tuple ($\frac{\text{exec. time}}{|\mathbf{R}|+|\mathbf{S}|}$) of hash join executed on four DBAs with different hash table implementations.

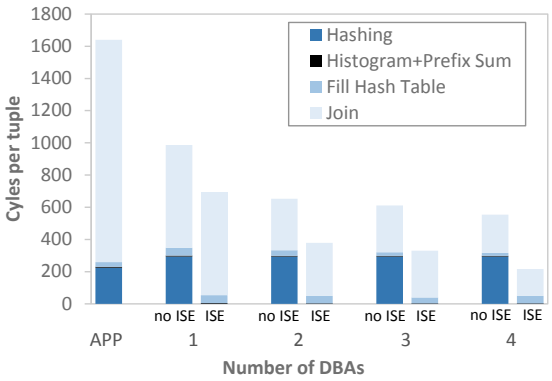

Figure 6: Cycles per tuple of hash join steps with histogram-based hash table (`Hist-HT`) for different number of cores ($|\mathbf{R}| = |\mathbf{S}| = 100{,}000$ tuple).

For the DRAM accesses during the join step, we mitigate the configuration delay by always loading a full NoC burst of 512 bytes and apply software pipelining to reduce the impact on NoC packet transfer time.

For the workload, we again use two input relations R and S with $|R| = |S|$ which consist of 64-bit key-payload pairs (tuples). The tuples of both relations are unsorted and stored in DRAM. The randomly generated 32-bit keys of R follow an uniform distribution within the domain $0, ..., 2^{32} - 1$. To reduce the output cardinality from at most $|R| * |S|$ to a predefined value, the keys of S are then chosen from the same domain as R, but in a way that the values with R overlap only by $\frac{1}{4}$.

We measure the execution time of the algorithms by cycle counters which are integrated in each processing module and the CM. Furthermore, we obtain the power consumption from analog-to-digital converters [14] which monitor voltage and current of the power supply rails.

4.2 Performance

In a first experiment, we run the algorithms on all four DBAs by varying the input cardinality. We set the upper limit to 100,000 tuples for each relation which is 50× what fits into one local data memory of a DBA. Hence, we ensure that initialization overhead in the DBA's software or the communication of the DBAs with the CM can be neglected. As depicted in Figure 5, we plot the number of cycles per input tuple which are calculated by the execution time in cycles divided by the total number of input tuples $|R| + |S|$. The curves show a linear slope at an increasing input cardinality. However, the hash join `LL-HT` increases faster than `Hist-HT`. Near 100,000 tuples, `Hist-HT` takes about 217 cycles/tuple which is around 30× better than the `LL-HT` hash join. Additionally, the algorithms perform on average ca. 1.6× (`LL-HT`) and 3.2× (`Hist-HT`) faster when using the instruction set extensions (ISE). We also plot the performance of the APP core for both hash joins as reference. It follows the same behavior and is on average 1.9× to 5.0× slower than the 4-core `LL-HT` and `Hist-HT` hash join, respectively.

To understand the difference of both algorithms, we run the hash join with the histogram-based hash table for a different number of cores and plot the number of cycles to process one tuple for the different hash join steps (see Figure 6). Note that the number of cycles for the hashing step includes hashing of all keys of both input relations and stays constant for different number of DBAs since every DBA hashes all tuples of R and S. Consequently for Figure 6, the histogram+prefix sum and the fill step can be assigned to the build phase while the join step represents the probe phase. Hence, when using the additional hashing instructions, we obtain on average a 3.8× and 1.6× performance improvement for the build phase and the probe phase, respectively. While the build phase only writes to DRAM, the probe phase needs to read tuples from the hash table. Due to high-latency DRAM reads, joining always takes at least 70% of the total execution time. The same applies for the APP core as can be seen in Figure 6.

The `Hist-HT` hash join stores all entries of a bucket successively in memory. Hence for each tuple in S, on average only one DRAM access at one single address is necessary. This even applies for a high number of collisions in the hash table. In contrast, bucket entries of the `LL-HT` hash join are scattered over the complete hash table. Assume that c collisions occur, also c accesses on different DRAM addresses are required, i.e., configuring the DMA controllers c times which finally results in an overall higher join time. We omit to plot Figure 6 again for the `LL-HT` hash join since the join step takes always more than 97% of the total execution time.

In conclusion, despite that the `Hist-HT` hash join runs through R twice, it has an performance advantage over the `LL-HT` since the probe phase dominates the total execution time. As mentioned earlier, configuring the DMA from the PM's software causes the major part of the DRAM read delay. Modifying the clock buffers between the NoC and the PMs to provide reduced round-trip delays would further improve the performance. We leave this idea open for future work.

We now want to study the particular performance improvements enabled by the instruction set extensions and multi-core processing. Figure 7 depicts the speedup for the `Hist-HT` hash join under different configurations. Every curve is related to the single-core execution of one DBA

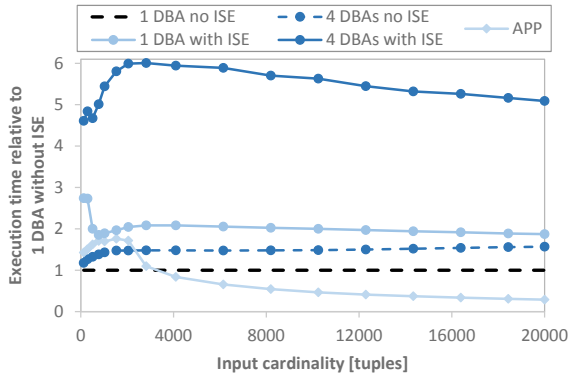

Figure 7: Performance improvements related to single DBA without using instruction set extensions for hash join with histogram-based hash table (`Hist-HT`).

without the usage of additional hashing instructions. We focus on smaller cardinalities to elaborate ranges where the input data size starts to exceed the size of the local memories and the cache, respectively. For the DBA, the data arrays in the local memory contain at maximum 4096 tuples. As explained in Section 3.3, one half is used to load data from DRAM while the DBA processes on the other part. As can be derived from Figure 6, the build phase takes at least 50 cycles/tuple. Knowing that the DRAM delivers 2.5 cycles/tuple and the NoC achieves 1 cycle/tuple, we conclude that the hash join is limited by the processing speed of the DBAs.

Furthermore, the results reveal the following statements:

- The performance advantage of the instruction set extensions decreases with higher cardinalities since joining dominates the total time as figured out previously.
- In comparison to the single-core execution, four cores increase the speedup. This shows that the system scales well with the number of cores as long as it is not memory I/O bound.
- When combining the benefits from the instruction set extensions and the parallelism on core-level, the speedup approaches a factor of 4 for input cardinalities above 20,000.
- The APP core follows a steadily increasing speedup until the input cardinality reaches the cache size of 16 kB. Hence at 2000 8-byte tuples, cache misses incur and the DRAM latency slows down the performance.

We do not explicitly report on the analysis for the `LL-HT` hash join since the behavior of these speedups are very similar to the `Hist-HT` implementation.

4.3 Power

We measure the power consumption of the Tomahawk4 MPSoC for both hash join realizations as well as with and without instruction set extensions. For that purpose, we run the algorithms in an infinite loop to monitor the power values. As a result, we found no significant difference between the implementations. Hence, we omit to distinguish between different implementations and decide to present all results only for the `Hist-HT` hash join which uses the additional hashing instructions.

Table 1 shows the total power consumption for different

PM Configuration	Number of DBAs 1	4	3	4
100 MHz, 0.90 V	60.3	70.7	78.0	82.7
200 MHz, 0.95 V	69.1	83.5	95.6	95.6
400 MHz, 1.00 V	88.6	105.8	123.0	133.1
500 MHz, 1.05 V	100.8	134.4	161.7	191.7

Table 1: Measured power consumption (in mW) of `Hist-HT` hash join at different frequencies and voltages. The power includes the PMs (DBA, DMA, local memory), CM, and the NoC.

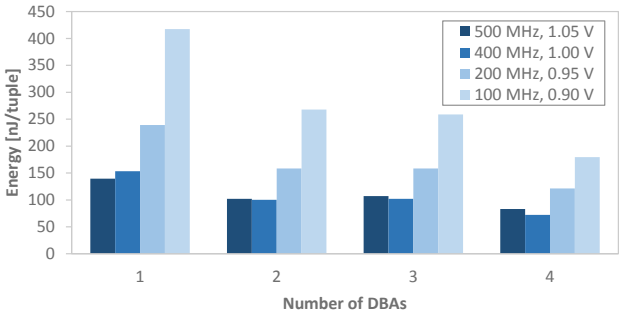

Figure 8: Energy consumption per tuple of `Hist-HT` hash join at different frequencies and voltages.

clock frequencies and supply voltages. Note that only the PMs are affected when frequencies and voltages change. The CM and the NoC still run at 500 MHz and 1.05 V and always consume about 51 mW. The algorithms achieve the highest throughput at the maximum clock speed of 500 MHz. In this case, the MPSoC including four PMs, the CM, and the NoC consume 191.7 mW. We further state that, e.g., for the 100 MHz configuration, the total power only increases by factor 1.4 when using four DBAs instead of one.

We select different clock frequencies and supply voltages to analyze the trade-off between power consumption and throughput. For this purpose, a suitable measure is the energy consumption which is calculated by dividing the dissipated power by the cycles spent to process one tuple. Figure 8 depicts the energy per tuple for a different number of cores when running the hash join with 100,000 tuples for each input relation. The overall lowest energy is required when four DBAs are used. Compared to the single-core execution, the performance improves by more than factor 4 (cf. Figure 7), but the power consumption only increases by factor 1.9 for the 500 MHz case. However, we found that the lowest possible energy consumption occurs with 72 nJ/tuple for four DBAs at the 400 MHz configuration. In this case, the trade-off between power and throughput is optimal. We conclude that the MPSoC achieves the highest energy-efficiency when using multiple cores which run at slightly reduced frequencies.

5. CONCLUSION

In this paper, we studied the implementation of hash join algorithms on the Tomahawk4 MPSoC. The chip integrates four database-specific accelerator cores each equipped with 128 kB of tightly-coupled SRAM. The cores are built upon an ASIP approach and support instructions to speedup an

integer hash function. An off-chip DRAM serves as a main memory and is shared by the database accelerators.

The results reveal that state-of-the-art hash join algorithms are well suited to be applied to MPSoC architectures. In particular, we found that partitioning the input data is essential to exploit the system parallelism. Furthermore, the size of the input relations and the hash table requires to include main memory accesses during execution. However, in order to hide the DRAM read latency, memory accesses should occur from subsequent addresses. By applying this approach to the hash table design, we obtain an up to 30× performance advantage. Moreover, the results provide the basis to establish low-power MPSoCs in database system which, e.g., may act as coprocessors next to high-performance CPUs to offload and accelerate basic database operations such as hash joins.

6. ACKNOWLEDGMENTS

This work has been supported in part by the state of Saxony under grant of the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (cfaed), and SFB912 – HAEC. This work has further received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 671566 ("Superfluidity") as well as from the ECSEL Joint Undertaking under grant agreement No. 692519 ("PRIME").

We also thank the Chair for Highly-Parallel VLSI-Systems and Neuro-Microelectronics of Technische Universität Dresden for backend design and PCB development of the Tomahawk4 chip. Furthermore, we would like to thank Synopsys and Cadence for providing software and IP.

7. REFERENCES

[1] O. Arnold, S. Haas, G. Fettweis, B. Schlegel, T. Kissinger, T. Karnagel, and W. Lehner. HASHI: An Application-Specific Instruction Set Extension for Hashing. In *ADMS*, pages 25–33, 2014.

[2] O. Arnold, S. Haas, G. Fettweis, B. Schlegel, T. Kissinger, and W. Lehner. An Application-Specific Instruction Set for Accelerating Set-Oriented Database Primitives. In *ACM SIGMOD*, pages 767–778, June 2014.

[3] O. Arnold, E. Matus, B. Noethen, M. Winter, T. Limberg, and G. Fettweis. Tomahawk: Parallelism and Heterogeneity in Communications Signal Processing MPSoCs. *ACM Trans. Embed. Comput. Syst.*, 13(3s):107:1–107:24, 2014.

[4] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced Allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999.

[5] C. Balkesen, G. Alonso, J. Teubner, and M. T. Ozsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1), 2013.

[6] C. Balkesen, J. Teubner, G. Alonso, and T. M. Özsu. Main-Memory Hash-Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. Technical Report 779, ETH Zürich, November 2012.

[7] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-Efficient Hash Joins. *Proc. VLDB Endow.*, 8(4):353–364, Dec. 2014.

[8] S. Blanas, Y. Li, and J. M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 37–48. ACM, 2011.

[9] M.-S. Chen, J. Han, and P. S. Yu. Data mining: An Overview from a Database Perspective. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):866–883, Dec 1996.

[10] J. Cieslewicz and K. A. Ross. Data Partitioning on Chip Multiprocessors. In *Proceedings of the 4th International Workshop on Data Management on New Hardware*, DaMoN '08, pages 25–34, New York, NY, USA, 2008. ACM.

[11] B. Gedik, R. R. Bordawekar, and P. S. Yu. CellJoin: A Parallel Stream Join Operator for the Cell Processor. *The VLDB Journal*, 18(2):501–519, Apr. 2009.

[12] S. Haas, O. Arnold, B. Nöthen, S. Scholze, G. Ellguth, et al. An MPSoC for Energy-efficient Database Query Processing. In *Proceedings of the 53rd Annual Design Automation Conference (DAC'16)*, pages 112:1–112:6, 2016.

[13] S. Haas, T. Karnagel, O. Arnold, E. Laux, B. Schlegel, G. Fettweis, and W. Lehner. HW/SW-Database-CoDesign for Compressed Bitmap Index Processing. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 50–57, July 2016.

[14] S. Haas, T. Seifert, B. Nöthen, S. Scholze, S. Höppner, et al. A Heterogeneous SDR MPSoC in 28nm CMOS for Low-Latency Wireless Applications. In *Proceedings of the 54th Annual Design Automation Conference (DAC'17)*, pages 47:1–47:6, 2017.

[15] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[16] S. Höppner, C. Shao, H. Eisenreich, G. Ellguth, M. Ander, and R. Schüffny. A Power Management Architecture for Fast Per-Core DVFS in Heterogeneous MPSoCs. In *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 261–264, May 2012.

[17] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young. Mobile Edge Computing-A key technology towards 5G. *ETSI White Paper No. 11*, September 2015.

[18] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU Join Processing Revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pages 55–62. ACM, 2012.

[19] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. S. J. Chhugani, A. D. Blas, and P. Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB*, 2009.

[20] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[21] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the Walkers: Accelerating Index Traversals for In-Memory Databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 468–479. ACM, 2013.

[22] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper. Massively Parallel NUMA-aware Hash Joins. In *INDM*, 2013.

[23] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A Fast Similarity Join Algorithm Using Graphics Processing Units. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1111–1120, April 2008.

[24] S. Manegold, P. Boncz, and M. Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709–730, Jul 2002.

[25] N. Mirzadeh, Y. O. Koçberber, B. Falsafi, and B. Grot. Sort vs. Hash Join Revisited for Near-Memory Execution. In *5th Workshop on Architectures and Systems for Big Data (ASBD 2015)*, 2015.

[26] S. Moriam and G. P. Fettweis. Fault Tolerant Deadlock-Free Adaptive Routing Algorithms for Hexagonal Networks-on-Chip. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 131–137, Aug 2016.

[27] R. Pagh and F. F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122 – 144, 2004.

[28] N. A. Rahman and T. S. Saad. Hash Join Algorithms Used in Text-Based Information Retrieval: Guidelines for Users. In *2008 International Symposium on Information Technology*, volume 2, pages 1–7, Aug 2008.

[29] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *In Proceedings of the 20th VLDB Conference*, pages 510–521. Morgan Kaufmann Publishers Inc, 1994.

[30] A. Ungethüm, D. Habich, T. Karnagel, W. Lehner, N. Asmussen, M. Völp, B. Nöthen, and G. Fettweis. Query Processing on Low-Energy Many-Core Processors. In *Proceedings of the International Workshop on Big Data Management on Emerging Hardware*, HardBD'15, 2015.

[31] M. Yabuta, A. Nguyen, S. Kato, M. Edahiro, and H. Kawashima. Relational Joins on GPUs: A Closer Look. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2017.