Overview on Hardware Optimizations for Database Engines

Annett Ungethüm[1], Dirk Habich[1], Tomas Karnagel[1], Sebastian Haas[2], Eric Mier[1],
Gerhard Fettweis[2], Wolfgang Lehner[1]

Abstract: The key objective of database systems is to efficiently manage an always increasing amount of data. Thereby, a high query throughput and a low query latency are core requirements. To satisfy these requirements, database engines are highly adapted to the given hardware by using all features of modern processors. Apart from this software optimization, even tailor-made processing circuits running on FGPAs are built to run mostly stateless query plans with a high throughput. A similar approach, which was already investigated three decades ago, is to build customized hardware like a database processor. Tailor-made hardware allows to achieve performance numbers that cannot be reached with software running on general-purpose CPUs, while at the same time, addressing the dark silicon problem. The main disadvantage of custom hardware is the high development cost that comes with designing and verifying a new processor, as well as building respective drivers and the software stack. However, there is actually no need to build a fully-fledged processor from scratch. In this paper, we present our conducted as well as our ongoing research efforts in the direction of customizing hardware for databases. In detail, we illustrate the potential of instruction set extensions of processors as well as of optimizing memory access by offloading logic to the main memory controller.

1 Introduction

The relationship of hardware and software in general is at an inflection point. While software benefited from higher clock cycles of modern CPUs for many years, nowadays, hardware components are advancing at an incredible speed providing a rich bouquet of novel techniques. While this trend is opening up many opportunities for software solutions, they also pose significant challenges and risks. Specifically for database systems, we identify relevant advances in three different areas as well as their interplay as depicted in Figure 1: (1) *processing elements*, (2) *main memory*, and (3) *network*. On the area of *processing elements*, the core count increased and the internal techniques like advanced vector extensions, pre-fetching or branch-prediction improved within modern CPUs. Furthermore, alternative *processing element* approaches like GPUs, FPGAs, or any combination of them provide a wide field of opportunities to adopt database engine architectures and their implementation [He09, MTA09b]. On the area of *main memory*, the capacities increased over the last years allowing to keep the full (transactional) database in main memory [Fa12], while recent advances in the context of non-volatile RAM may be considered a disruptive technology for database systems with impact on data structure design, recovery mechanisms etc. [APD15]. On the *network* side, the network becomes complex due to an increasing number of cores. For example, in large NUMA (non-uniform memory access) multiprocessor systems, each

[1] Technische Universität Dresden, Database Systems Group, 01069 Dresden, <fistname>.<lastname>@tu-dresden.de

[2] Technische Universität Dresden, Vodafone Chair Mobile Communications Group, 01069 Dresden, <fistname>.<lastname>@tu-dresden.de

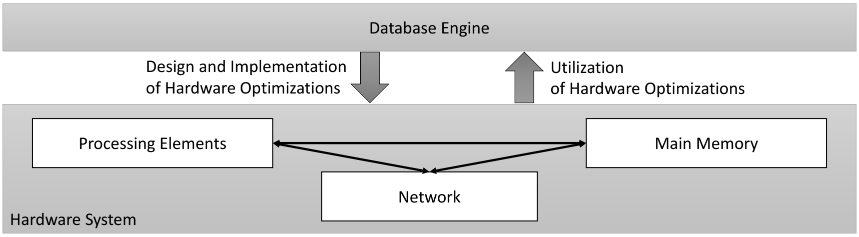

Database Engine
Design and Implementation of Hardware Optimizations
Utilization of Hardware Optimizations
Processing Elements
Main Memory
Network
Hardware System

Fig. 1: Overview of Hardware/Software Co-Design for Database Engines.

multiprocessor has its own local main memory that is accessible by other multiprocessors via a communication network. This NUMA behavior may cause significant performance penalties when not considered correctly [KSL13], while Infiniband [Me07] or OmniPath [Bi15] provide bandwidth figures for communication between nodes close to performance characteristics of node-internal communication (e.g. via QPI [In08]). Considering higher latency in case of RDMA (remote direct memory access) [We15] for accessing memory of remote hosts, memory access of remote and local memory access (in case of multi-hop) are by now already exhibiting similar characteristics blurring node boundaries.

In general, all software components of database engines are highly specialized for the underlying hardware to satisfy the high performance requirements. Therefore, database engine developers usually need a deep understanding of all hardware features. Apart from this software optimization, hardware/software co-design approaches gain in importance, because tailor-made hardware allows to achieve performance numbers that cannot be reached with software running on general-purpose CPUs, while at the same time, addressing the dark silicon problem. Within this paper, we give an overview of our currently ongoing as well as already conducted research work to support database systems by customizing hardware components. Although we tackle this research field from different angles, we will focus on the perspective of processing elements, discuss optimization opportunities as well as consider the relationship between processing elements and main memory.

Our focus on customizing processing elements for data management solutions is mainly driven by two observations: (1) *thermal effect* and (2) *massive on-chip / on-socket parallelism*. First, the notion of „dark silicon“ [Es11] represents the consequence of thermal problems in traditional chip design. While silicon structures are getting denser, the complete die can no longer be supplied with sufficient power due to an increasing power density. As an effect, some chip area may end up unpowered („dark silicon“). These areas can be used for specialized circuits which can be activated on demand as an alternative to chip space housing generic functionality. While different application domains already exploit the dark silicon effect for special-purpose extensions (e.g. bio-inspired application, image processing or cryptographic support), the database community does not yet heavily consider this opportunity. Second, since data transfer very quickly becomes the bottleneck in data-driven applications, massive parallelism in combination with tight memory coupling techniques might be well suited for database systems to pre-process data sets before forwarding the data to more complex operators. However, thermal constraints again play a crucial role resulting in a need of extremely low-power processing elements.

The remainder of the paper is organized as follows: We begin with a short introduction of the Tomahawk architecture—reflecting the foundation of our work—in Section 2. Then, we outline some of our developed database-specific optimization for the processing elements in Section 3. In particular, we want to illustrate the potential of instruction set extensions of the core processors. Afterwards, we sketch preliminary research results in optimizing memory access by offloading typical memory access logic of data structures (e.g. filtering or search in trees) to the memory controller in Section 4. The objective of this offloading is to reduce the traffic on the chip's internal network and thus optimizing the latency of core database access primitives. Finally, we conclude the paper with related work and a summary in Section 5 and 6.

2 Tomahawk Architecture

The hardware foundation of our hardware/software co-design approach is the Tomahawk platform [Ar14c]. This platform is a heterogeneous multiprocessor system-on-a-chip (MPSoC) and has been developed at our university, whereby the primary focus was on supporting mobile communication applications. Nevertheless, the platform aims to be able to adapt for highly specialized tasks while being very energy efficient. Next, we briefly describe the overall MPSoC architecture, before we introduce more details about our enhancements.

2.1 Overview

Generally, the Tomahawk platform consists of two subsystems called control-plane and data-plane [Ar14c] as illustrated in Fig. 2. The control-plane subsystem comprises a CPU (traditional fat core), a global memory and peripherals, whereas this CPU is also called application core (App-Core). The App-Core is responsible for executing the application control-flow. The data-plane subsystem consists of a number of processing elements (PEs), each equipped with a local program and data memory. PEs are not able to access the global memory directly, instead a data locality approach is exploited using scratchpad local memory. That means, the PEs are explicitly isolated from the control-plane subsystem. The data-plane subsystem is used as an accelerator for the control-plane. Therefore, this subsystem can be seen as slave unit in the overall system architecture. Both subsystems are

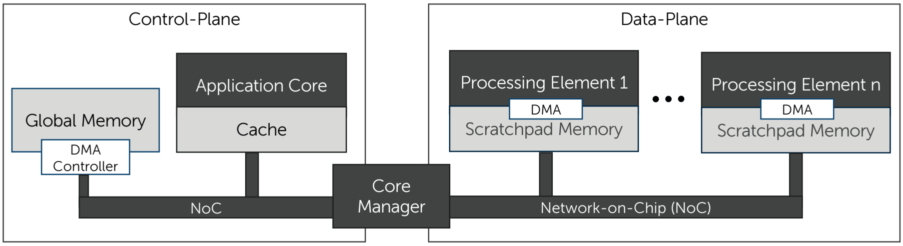


Fig. 2: An overview of the Tomahawk architecture.

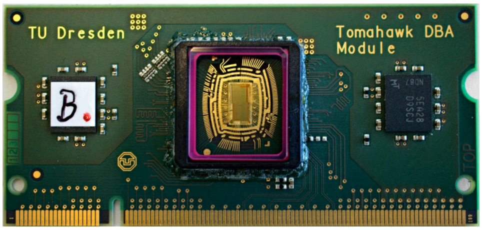


Fig. 3: The Tomahawk DBA module including the processor and two 64 MB SDRAM modules.

decoupled logically and linked together through a controller called Core Manager (CM). The CM is responsible for the task scheduling of the PEs, the PE allocation, and data transfers from global memory to the PEs and vice versa. Additionally, the CM can perform frequency scaling of the PE cores to minimize the power consumption.

Based on the overall collaborative setting at our university, we are able to tailor this Tomahawk platform for our database requirements. A number of modifications has been recently added to enhance data processing. This includes minor and uncritical changes like an enlarged scratchpad memory of the PEs as well as more sophisticated enhancements like specialized instruction set extensions for the PEs (see Section 3). To further increase the performance, we have also enhanced the DMA controller of the global memory to push down data intensive operations, like filtering or pointer chasing (see Section 4). The objective of this DMA enhancement (or intelligent DMA, iDMA) is to reduce the necessary data transfer between the control and data-plane which is clearly a bottleneck for processing large amounts of data. We have focused on the DMA controller of the global memory for several reasons. Nevertheless, the iDMA concept should be also beneficial for the PEs.

The overall development is conducted in cycles and the Tomahawk is currently in its fourth revision. The most recent available version is the third revision, also called the *Tomahawk Database Accelerator (DBA)* (Fig. 3) featuring 4 PEs. It is the first Tomahawk version that contains a specialized instruction set for database operations, which is further explained in Section 3. In the following, we describe the foundations for the processing elements and the DMA controller in more detail.

2.2 Processing Element Foundation

To not start from scratch, we use a Tensilica LX5 Reduced Instruction Set Computer (RISC) processor[3] for our processing elements offering a basic 32-bit ISA (instruction set architecture) with a base instruction set of 80 RISC instructions and one load-store unit. The LX5 is fully programmable with high-level programming languages, e.g. C/C++. Like every other synchronous microprocessor, it processes instructions in four steps: (1) Fetch Instruction, (2) Decode Instruction and Fetch Operands, (3) Execute, and (4) Store Results.

In step (1), the current instruction is *fetched* from the main memory into the *local instruction memory* and from the *local instruction memory* into the instruction *register*. In step (2), the

[3] Tensilica LX5 Datasheet - `http://ip.cadence.com/uploads/132/lx5-pdf`

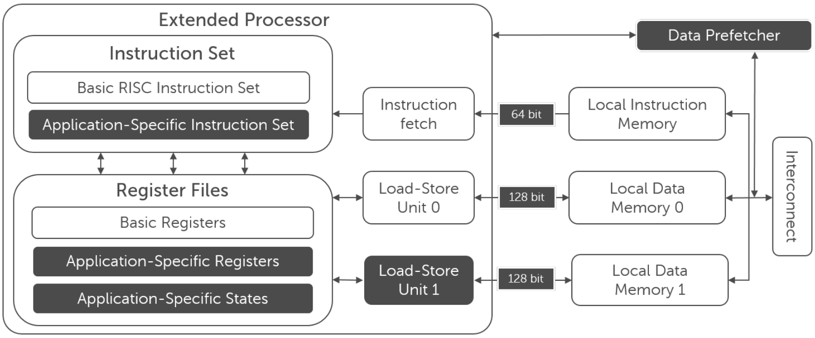


Fig. 4: The Tensilica LX5 Processor and the surrounding components have been extended by the features highlighted by a gray box.

instruction is decoded. This instruction belongs to the *instruction set* of the architecture. A result of the decoding is the determination of the necessary operands, which are then loaded from the *local data memory* into the corresponding *registers*. A *Load-Store Unit* is responsible for this step. In step (3), the actual computation is done. The results are stored in the corresponding *registers*. Step (4) finishes the instruction execution by storing the results in two phases. First, they are written back to the *local data memory*. Again, the *Load-Store Unit* is responsible for this phase. Then in the second phase, the result is copied from the *local data memory* into the main memory.

Besides the basic functionality, the Tensilica LX5 processor is configurable allowing us to modify and extend it according to the requirements of a database engine. A central extension which we added are an instruction set extension for database operators. Instruction set extensions are a common way for making frequent tasks faster, more energy efficient, and easier to use. Because of the different components which are involved in the four steps of the instruction execution, an instruction set extension does not only extend the machine code by plain instructions but also by architectural aspects like registers, data types, or memory modules if this is necessary. A well known example for instruction set extensions is SSE[4] for Single Instruction Multiple Data (SIMD) operations in x86 CPUs. In the second SSE revision, 144 new instruction were introduced together with enlarged registers from 64 bit to 128 bit, such that the values for two floating-point operations fit into one register enabling a faster double-precision processing[5].

For our database instruction set extension, we have added and modified a number of units of the Tensilica LX5 processor in general. An overview of the changed processor is shown in Fig. 4. In addition to the basic core, we have added application specific registers, states, and an application specific instruction set. These extensions vary for every developed extension, i.e. a bitmap compression extension and an extension for hashing have different instruction sets and registers. However, some modifications are shared by all instruction set extensions. In detail, the shared components are:

[4] Streaming SIMD Extensions

[5] Source: `http://www.intel.eu/content/www/eu/en/support/processors/000005779.html`

Time	Description	Latency in ns
t_{RCD}	Row activate to columns access delay	15
t_{RP}	Bank precharge to row activate delay	15
t_{CL}	Read column access to first data delay	15
t_{RAS}	Row activate to the next bank precharge delay	40

Tab. 1: Timing Constraints of the Micron DDR2 SDRAM (400 MHz)

Second Load-Store Unit We introduced a second load-store unit for being able to access two local memories simultaneously.

Data Prefetcher The data prefetcher is no part of the processor but operates next to it. It preloads data from an external memory into the local memories. This way it alleviates the memory access bottleneck.

Bus Width Extension The instruction bus is extended from 32 bit to 64 bit enabling us to introduce longer instruction words. The memory bus is extended from 32 bit to 128 bit for a faster memory access.

2.3 Main Memory Foundation

The global memory of our Tomahawk platform is of type DDR SDRAM (Double data rate synchronous dynamic random-access memory) - in detail, a Micron DDR2 SDRAM[6]. SDRAM devices use memory cells to store data, thereby the memory cells are arranged in two-dimensional arrays (matrices), also called banks. A typical SDRAM device has 4 or 8 internal banks. Therefore, bank index, row index and column index are necessary to locate a memory block in an SDRAM device. Furthermore, memory cells need to be periodically refreshed in order to keep the data due to the leakage currency of the capacity. This is known as refreshing, which can be simply done by a dummy read without output. According to the standard, every bit of an SDRAM device has to be refreshed every 64ms or less, which is periodically done by a refreshing logic.

Accessing an SDRAM device requires three SDRAM bus transactions besides the data transfers: *bank precharge*, *row activate* and *column access*. A *bank precharge* prepares the selected bank. A *row activate* command activates the word line selected by row index and copies the entire row in a row cache. Then one or more column accesses can select the specified column data using column index and output it through I/O gate logic. To access a different row of the same bank, the bank needs to be *precharged* again, followed by a new row activate and column access. Furthermore, SDRAM devices have a set of timing constraints that must be met for all transactions. For example, after a *bank precharge*, no transaction can be performed on this bank until t_{PR} (bank precharge to row activate delay time) has lapsed. Similarly, t_{RCD} is the minimal time interval between a row activate and a column access. For a read, the data will appear on the data bus t_{CL} (read column access to first data delay) cycles after the column access. Tab. 1 shows the timing constraints of our Micron DDR2 SDRAM.

[6] Micron DDR2 SDRAM - `https://www.micron.com/products/dram/ddr2-sdram`

SDRAM devices usually support burst mode allowing multiple sequential data within the same row to be accessed without additional column access transactions. The number sequential data in the burst mode is known as burst length. For DD2, the burst length should be set to 4 or 8. Because main memory accesses are cache misses, memory accesses always require entire cache lines. The burst length should be selected such that an entire cache line can be transferred in one burst. For an instance, for a system that has a 64-bit memory bus and 64B L2 cache line size, the burst length should be set to 8.

Generally, main memory requests issued by the CPU contain the address of requested memory block, the size of the block and type of the request (read or write). SDRAM devices can not process these requests directly. Thus SDRAM controllers, also known as memory controllers, are used to manage the flow of data going to and from the memory. In our case, we use a Synopsys DWC DDR2 memory controller[7]. Those controllers contain the logic necessary to read and write SDRAM devices. To locate a memory block in the SDRAM space, an SDRAM controller translates the address of requested memory block into the SDRAM address, which is composed of channel, rank, bank, row, and column index. With the considerations of timing constraints and possible bus contentions, the controller generates transactions to access the SDRAM device. For reads, the SDRAM controller returns the requested data to the CPU; for writes, it updates the requested memory block in the SDRAM device with the new data. Also, the SDRAM controller periodically refreshes the SDRAM devices to prevent data loss.

To summarize, SDRAM controllers and SDRAM devices compose our global main memory. SDRAM devices are organized to create an SDRAM address space. SDRAM controllers provide a memory interface through which the CPU or other devices (i.e. DMA controllers) can issue memory requests and receive responses. The DMA controller (direct memory access) is an additional feature that allows certain hardware subsystems e.g. processing elements to access main memory independently of the CPU making them an ideal candidate for customizing with appropriate logic. Without a DMA controller, the CPU is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work.

3 Processing Element Extensions for Database Primitives

To better support database engines from a hardware perspective, we developed certain instruction set extensions for database primitives like hashing [Ar14a], bitmap compression [Ha16c] and sorted set operations [Ar14b] over the years. In this section, we want to describe design questions, present instruction insides, and highlight an example instruction.

3.1 Design Questions

On the hardware level, the logic of an operation is implemented as a circuit consisting of one or more logic gates. For instance, an AND operation is realized with a single AND gate

[7] Memory Controller - `https://www.synopsys.com/dw/ipdir.php?ds=dwc_ddr2-lite_phy`

while a full adder uses two AND gates, two XOR gates, and one OR gate. It is also possible to merge several instructions into a new instruction. The input signals of such circuits are voltages which can be interpreted as *0* or *1*. If the input signal causes the circuit to close, there is a voltage at the output. This is interpreted as *1*. If the circuit is not closed, there is no voltage at the output. This is interpreted as *0*. The time it takes between switching on the input signal and getting an output signal is a hard constraint for the frequency of the circuit and therefore for the frequency of the whole core. The longer a circuit becomes, the longer it takes for getting an output signal. This results in a lower core frequency. Furthermore, we always have to consider the worst-case scenario, i.e. the longest path through a circuit. This scenario is called the *critical path*. Additionally, a complex circuit uses a larger chip area than a simple one. Thus, we have to find a compromise between the complexity of the circuit, i.e. the instruction, and the core frequency. This requires balancing different demands:

Universality: A common way for implementing a specialized instruction is to merge two or more basic instructions. If the resulting instruction is heavily used, this is a gain for the application because it can execute the instruction in one cycle instead of multiple cycles. But if this instruction merging is exaggerated to a point where the instruction is so specialized that it is rarely used, it wastes space on the processor and might decrease the clock frequency because of the potentially long *critical path*.

Technical feasibility: There are operations which are well suited for a hardware implementation while the corresponding software implementation needs many instructions, e.g. bit manipulations. For other operations, the tradeoff between a relatively slow software implementation and the increased required area and lowered clock frequency caused by a hardware implementation, has to be considered very carefully. For instance, an SIMD instruction for adding two vectors requires more space on the processor the larger the vectors get. Therefore the possible length of the vectors has to be limited.

For database operations this means that an instruction set extension has to cover the basic operators rather than a whole query. It might be possible to implement a whole TPCH query in hardware. It is very likely, that this quey would be processed much faster than the software implementation. However, this extension can not be used for anything else than the exact implemented query, while it occupies a significant amount of the area on the processor and has a longer *critical path* than a basic operator.

Furthermore, operators like a join or an intersection are by default no atomic operations. They can be split into several small steps. Each of these steps can be implemented separately resulting in a dedicated instruction for each of them. In the most extreme case, these instructions are a subset of the processor's basic instruction set and would therefore not result in a performance gain. Vice versa, packing all steps into one single instruction would clearly violate the demand for a certain universality. For this reason, we investigated the database operations, which we planned to implement. Additionally we identified and classified the working steps they have in common (Sec. 3.2). This way, we simplify the decision whether a step should get an instruction on its own or be merged with another step.

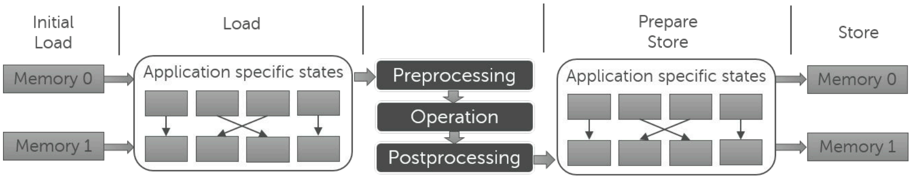


Fig. 5: Execution of a database operation using our instruction set extension

3.2 Instruction Insides

An instruction does not only consist of the logical operation itself, e.g. comparing two records, but also contains load and store operations and some pre- and post-processing. Therefore, our database operations contain the following steps (see also Fig. 5):

Initial Load Each load-store unit loads up to 128 bits from the memory into the intermediate states of the processor.

Load If necessary, the data is (partially) reordered and moved to a second internal state, e.g. for getting a 128-bit aligned line.

Pre-processing During the pre-processing phase all values for the actual operation are computed, e.g. the address of a bucket or the branch which will be chosen.

Operation The actual computation is done.

Post-processing Information for storing the data is collected, e.g. if a previous result will be overwritten.

Prepare Store If necessary, the results are shuffled. Afterwards they are stored into the dedicated states of the processor.

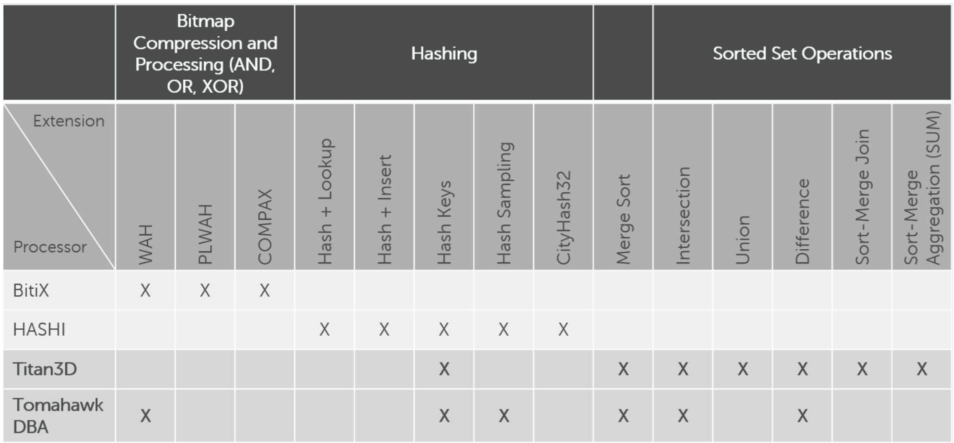
Store The results are written back to the local memory.

Following the demand for universality and a short *critical path*, the load operations are decoupled from the actual operation and store phase. Instead they are implemented as separate instructions. This way they can be reused on operations which request the same input data types. Since we have two memory modules and load-store-units, we can finish two load instructions within one cycle.

The operation and store phases are either separated or combined into a single instruction depending on the database primitive. Because of the complexity of some database primitives, the corresponding instructions are often scheduled over multiple cycles for obtaining the best compromise between execution time and latency. The number of the necessary cycles is determined empirically.

3.3 Example Extension

We realized several extensions implementing the introduced steps for different database primitives and added these extensions to the PEs of the Tomahawk. The natural decision was to start with some basic operators every database system needs to implement. For this purpose, a Merge-Sort has been developed as well as some operators which work



	Bitmap Compression and Processing (AND, OR, XOR)			Hashing						Sorted Set Operations				
Extension / Processor	WAH	PLWAH	COMPAX	Hash + Lookup	Hash + Insert	Hash Keys	Hash Sampling	CityHash32	Merge Sort	Intersection	Union	Difference	Sort-Merge Join	Sort-Merge Aggregation (SUM)
BitiX	X	X	X											
HASHI				X	X	X	X	X						
Titan3D						X			X	X	X	X	X	X
Tomahawk DBA	X					X	X		X	X		X		

Fig. 6: Overview of our extensions and the processors they are implemented on [Ar14b, Ar14a, Ha16c]. The processors which have already been manufactured at this point are highlighted.

on the produced sorted sets, i.e. Intersection, Difference, Union and Join [Ar14b]. Two processor designs featuring instruction set extensions for hashing [Ar14a] and compressed bitmap processing followed [Ha16c]. The first manufactured processor containing a subset of these extensions is the *Titan3D*, which was successfully tested [Ha16b]. Finally, the *Tomahawk DBA* was built with four extended cores [Ha16a]. A summary of the developed extensions can be found in Fig. 6.

One of the extensions which are available on real hardware is the processing of compressed bitmaps. Bitmaps can be used for a fast evaluation of query results. This is especially useful when the number of distinct values is small. Then, a match between the rows and an attribute can be stored as bitmap with a set bit indicating a match. An operation then only has to work on the bits and not the attributes themselves. We developed extensions for the compression and decompression of bitmaps using the *Word-Aligned Hybrid (WAH)* code and for basic operations like *AND* and *OR* on the compressed bitmaps [WOS04]. Furthermore, we implemented the same operations for two additional bitmap encodings, namely PLWAH [DP10] and Compax [FSV10].

The WAH code is used to encode bitmaps with run length encoding (RLE). It does not use signal words but encodes the necessary identifiers within every word [WOS04]. A WAH compressed bitmap can contain an uncompressed word (literal) or RLE compressed words (fills). The distinction between literal and fill is done by reading the first bit. A 0 indicates that this word contains a literal, a 1 indicates that it contains a fill. If it contains a literal, the remainder of the word represents the uncompressed word. If the word contains a fill, the second bit shows if set or unset bits are compressed. The remainder of the word is used to store the number of compressed chunks. For instance, the compressed word 11000010 expands to 1111111 1111111. The first bit indicates that this word is a fill. The second bit shows that the fills are set bits. The rest of the word shows that there are two chunks encoded. In the original work and in our implementation, a word is 32 bits long and a chunk contains 31 bits.

While operations on uncompressed bitmaps are memory bound, operations on compressed bitmaps are more complex, while having less data, therefore, these operations tend to be computation bound. Both cases have been tackled by our solution following the steps of the previous section:

Initial Load Since a second load-store unit has been introduced and the bus width has been extended to 128 bits, four 32 bit words of two input bitmaps can be loaded simultaneously from the local memory into the intermediate states of the processor.

Load For ensuring a continuous availability, the data is reordered because the memory access is aligned to 128 bit lines.

Pre-processing The first bit of the first input word of every input bitmap shows if it is a fill or a literal. If it is a literal, the word is preserved. If it is a fill, the input words are overwritten with the corresponding set or unset bits.

Operation For literals a bit-wise operation is performed. Fills only need to compare one bit. We implemented the operations AND, OR and XOR. This is done 4 times since there are 4 code words in a 128 bit line.

Post-processing The result of the operation is written to the output bitmap. If the result is a literal, it is appended to the output. If it is a fill, the previous word is overwritten to represent one more chunk.

Prepare Store To use the 128 bit memory bus optimally, four 32 bit words are buffered in a dedicated register before writing them back to the local memory.

Store The four buffered words are written to the local memory.

Because there are two load-store units, two load instructions can be executed simultaneously. They finish within one cycle. The remaining steps are scheduled over 4 cycles. For this purpose, an internal pipeline has been added. The main loop of the operation takes the majority of these four cycles. This is why the operation and the store process have been merged into one instruction. Hence, a code doing an AND comparison between two bitmaps is depicted in Fig. 7.

resetInst();	*Initialize states*
WUR_opID(0);	*Write into user register defining the operation,* *0 indicates an AND operation*
...	*Setting pointers and auxiliary variables*

do{
    ldXstream();
    ldYstream();
	
	Load next 128 bits of bitmaps,
	Together both load operations need 1 cycle

WAHinst();
    WAHinst();
    WAHinst();
	Do bitwise operation 4 times since there are four
	32 bit words in the state,
	Each operation uses 1 cycle

}**while**(WAHinst());	*Returns 0 if bitlist has reached the end,* *Check for condition uses one more cycle*
...	*Any left over words in state? If yes, process them.*

Fig. 7: Code snippet which calls and AND operation on two WAH compressed bitmaps

resetInst();	*initialize states*
doWAHinst(0, 1024);	*perform AND operation*

Fig. 8: Reduced code snippet using our library.

The cycle used for checking the condition at the end of the while loop can be avoided if the number of loops is precalculated and the while loop is replaced by a for loop. For simplifying the usage of the extensions, there is also a C-library hiding these instructions behind handy functions. Using this library, the example reduces to the code snippet as shown in Fig. 8.

The first parameter of `doWAHinst` in Fig. 8 is an `unsigned int` defining the operation and the second parameter is an `unsigned int` containing the size of a bitmap in bits. The pointers to the bitmaps are globally defined.

3.4 Experiments

To demonstrate the gains of our hardware implementation, we compare it to the Tensilica LX5 core which we used as the base for our PEs and to an Intel i7-6500U, which is a current low-power CPU. For the sake of comparability, we measure the single thread performance on the i7 since the LX5 has only one core. All test cases are written in C and compiled with the -O2 flag. The ISA extensions, which have not been manufactured, run in a cycle accurate simulator provided by Tensilica.

Compression: Fig. 9 shows the execution times of operations on differently compressed bitmaps. Additionally the compression and decompression times for the WAH compression are pictured. The values are averaged over different bit densities ranging from $1/2^0$ to $1/2^{16}$. Each input set has a size of 2500 elements. For every test case, the comparison shows that there is a speedup of two orders of magnitude if the extensions are used compared to not using them. The extensions even outperform the i7 significantly. The latter is even more surprising considering that the i7 draws between 7.5 and 25 W while the LX5 does not draw more than 0.15 W in any of our test cases. The exact performance gain varies

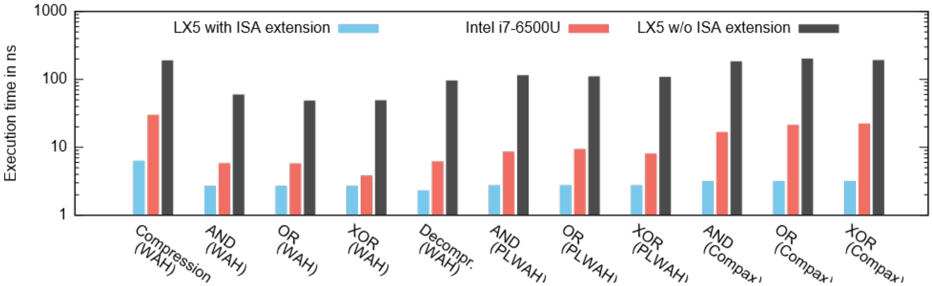


Fig. 9: Execution times for different bitmap compression techniques and operations on the compressed bitmaps

among the compression methods. For instance, the operations on WAH-compressed bitmaps are only twice as fast using our extension compared to the i7. However, using the more compute-intensive Compax algorithms, we can see speedups of 500% and more.

Further Primitives: Fig. 10 shows the throughput of some other selected operators, i.e. a merge-sort, some sorted set operations, and hashing. The input set size we used for the sorting and the sorted set operations is 5000 32-bit elements. That means, a set still fits into the local memory of a PE. The selectivity is set to 50%. In our experiments, we found the merge-sort to be performing worst compared to the i7. There, our extension does not reach the throughput of the i7 but stays within the same order of magnitude, while drawing only a fraction of the power of the i7. The merge-sort is the operation with the most accesses to the local memory since intermediate results have to be stored. These accesses have been accelerated by introducing the second load-store unit and increasing the data bus width, however, the possibilities for further optimization in this direction are more restricted than for the algorithm implementation itself. On the other hand, for operations on the resulting sorted sets, the memory accesses can be reduced and serialized. Hence, they outperform the i7 while still only draining 0.15 W at the maximum. The highest performance gain is reached with the hashing operators. There is a 100x speedup compared to the single thread performance on the i7 and even a 1000x speedup compared to the C-implementation on the LX5 without the ISA extensions. Hashing is also the application with the highest optimization potential starting as the lowest performing operation and becoming the most efficient operation when the respective extensions are used.

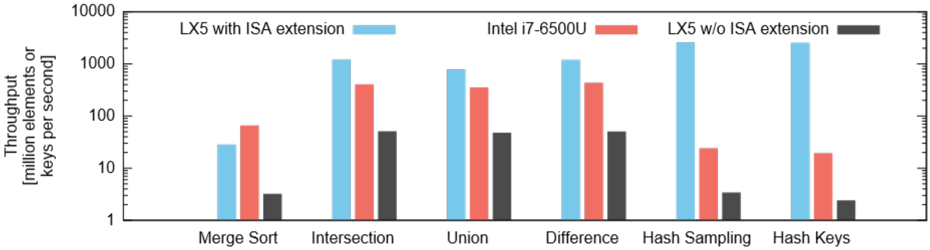
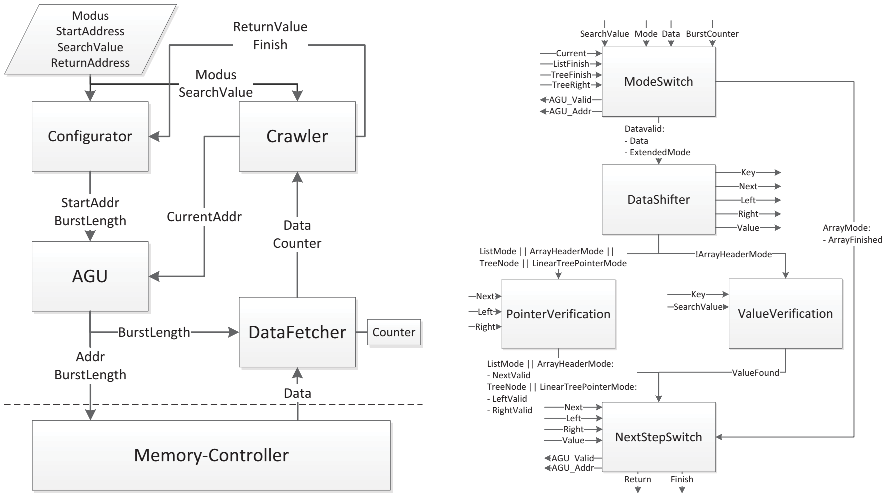


Fig. 10: Throughput for different operations.

4 Intelligent Main Memory Controller

In addition to the hardware optimization by means of special instruction set extensions for the PEs, access to the data in the global main memory can also be made more efficient. Instead of transferring massive amounts of data to the PEs for processing, e.g. filtering out data or searching a specific object in a data structure, we want to push-down the appropriate logic to the DMA controller. In this case, our intelligent DMA controller (iDMA) will be responsible to conduct fundamental data-intensive tasks like filtering or searching and the final result is only transferred to the PE.

In detail, we designed a first version of our iDMA approach which is able to answer point queries on various dynamic data structures. The simplest dynamic data structure is the linked list. With the help of pointers, appropriate objects are coupled one after the other. To extract a particular object (point query), the list must be searched from beginning to end using the pointers. This evaluation is usually done on a PE which requires the transfer of all necessary memory blocks. By dynamically inserting and removing objects, the objects usually do not necessarily co-locate in main memory, which increases the number of transferred memory blocks from the global memory to a PE. Instead of extracting the object at the PE, our iDMA takes over this task and transfers only the desired object to the PE which reduces the data transfer massively. In addition to the linked list, our iDMA also supports hash tables with overflow chains, linked arrays, and the binary tree. For all of these dynamic data structures, our iDMA offers an enhanced point query interface.



(a) iDMA block diagram

(b) Submodules of the *Crawler*

Fig. 11: Enhanced DMA controller.

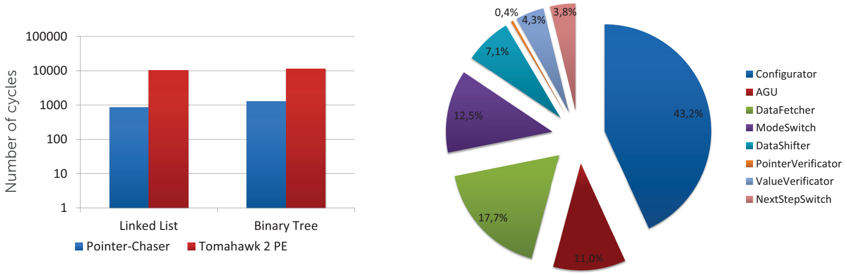
To support point queries in our iDMA, the corresponding search logic for the different data structures is translated into hardware. Fig. 11(a) shows the block diagram of our enhanced DMA controller. The *Configurator* accepts requests and analyzes them to make a decision which actions must be performed. Thereby, different requests at the same time are possible. For the processing of point queries on dynamic data structures, the *Configurator* needs the following input: (i) the data structure to be processed, (ii) the address of the first object of the data structure, (iii) the search object (point query), and (iv) an address to which the result of the search is written. In the following steps, the enhanced DMA controller answers the point query on the corresponding data structure by scanning over the data as done in the software. In order to do that, additional components are required to load the necessary data by directly addressing the memory. One component is the *Address Generation Unit (AGU)*, which generates the necessary memory instructions and passes them on to the memory controller. If the delays of the memory controller and the memory are greater than the burst length, further burst requests must be pre-loaded for the some dynamic data structures. The *AGU* receives the calculated prefetch distance from the *Configurator* so that the preload can be executed.

Then, the *DataFetcher* receives the data from the memory and counts the number of burst request replies. In addition, the *DataFetcher* compares the storage tags with the stored tags to assign the data obtained to the requests as multiple requests are possible. Then, the data with the assigned request is transferred to the subsequent *Crawler* module. This *Crawler* is responsible for complete data processing. For this, the crawler is divided into several submodules as illustrated in Fig. 11(b). On software or application level, an object is usually not processed until it is fully loaded. However, since the bandwidth of the memory is limited, an object can usually not be loaded in one clock cycle. But, there is the possibility to partially process the objects by prior knowledge of the dynamic data structure as done by our iDMA.

The *ModeSwitch* as depicted in Fig. 11(b) selects per clock the mode for the appropriate data structure and outputs it to the other submodules of the *Crawler*. Additionally, the *ModeSwitch* sends burst requests to the *AGU* for large data structures (e.g., linked arrays), since often a burst is not sufficient to load the entire object. For that, the *ModeSwitch* receives the necessary number of burst requests from the *Configurator* and the current burst position of the *DataFetcher*. The burst requests are scheduled in an appropriate way, so that the queue does not overflow in memory. Furthermore, the scheduling considers the time constraints as introduced in Section 2.3. Then, the *DataShifter* receives the data of the memory which was obtained by the *DataFetcher* and divides it into blocks of the size of 32 bits. We currently support only 32bit objects. Afterwards, the *ValueVerificaton* and the *PointerVerification* get the data. Here, the object is compared with the point query (search criterium) and the pointers are checked for validity. All signals are then sent to the *NextStepSwitch*, where they are evaluated before the next step is initiated. For example, the *NextStepSwitch* starts the transfer of a new object of the data structure or sets the finish signals so that the Configurator can terminate the search.

To summarize, our iDMA controller answers point queries on dynamic data structures (linked list, hash tables with overflow chains, linked arrays, and the binary tree) using a hardware-implemented pointer-chasing approach. To evaluate our approach, we integrated the iDMA in the Tensilica cycle accurate simulator, so that we are able to compare the iDMA approach with a software-approach running on a PE. Fig. 12(a) shows the results for a linked list and a binary tree, both are holding 840 32-bit objects. The SDRAM properties and timing constraints are described in Section 2.3. As we can see, our iDMA (pointer-chaser) outperforms the point query processing on the PE by a factor of 10 (in terms of number of cycles).

Fig. 12(b) depicts the percentage areas of the partial modules of our iDMA based on the developed hardware model. The *Configurator* takes about 43% of the area because the parameters of the search (search value and return address), the data structures, as well as other signals must be stored in registers. In addition, arithmetic units are added for the calculation of the required number of bursts. The *AGU* requires approximately 11% of the area, since the *AGU* needs registers to store the current memory tag which must be sent with each memory request so that the responses can be assigned. Since the *DataFetcher* has to assign the replies of the memory to the queries by matching their tags, the *DataFetcher* needs a FIFO for buffering all requests from the *AGU* resulting in 11% of the area. Like the *AGU*, the *ModeSwitch* of the *Crawler* requires several registers and arithmetic units for processing. This results in the required area of approximately 12.5%. The remaining four submodules of the *Crawler* only process signals and make decisions so that no data is stored or arithmetic is executed. Therefore, their surface areas are not so large.



(a) Execution times in number of clocks of iDMA with pointer chaser and a processor with an SDRAM

(b) Percentage area fraction of the submodules of the pointer chaser component

Fig. 12: Evaluation of Pointer Chaser Component.

5 Related Work

Fundamentally, hardware/software co-design can be done for all hardware components of (a) the processing elements, (b) the memory, and (c) the network. In this section, we want to discuss different approaches:

Processing Elements A common technique for the extension of processing elements—which is widely used in commercial hardware systems—is the extension of the basic instruction set of standard CPUs with use-case specific frequently used instructions. Apart from the widely known SSE extensions, Intel® also developed extensions for other use-cases, e.g. the Software Guard Extensions (SGX) [An13] or the Advanced Vector Extensions (AVX) [Fi08]. While AVX is basically an enhancement of SSE4, SGX serves a completely different domain. It introduces protected software containers, called enclaves, for securing sensitive content. The access to these enclaves is protected by hardware control mechanisms. Thus, not performance but security is the optimization goal for the SGX extension. This is especially useful in distributed applications and client-provider-scenarios, and holds the potential for paradigm changes on the networking layer (c). There are also extensions developed by ARM®, e.g. THUMB which is now a de-facto standard extension for embedded ARM® cores or NEON which is an extension for SIMD processing.

A way for bringing very specialized functionality into hardware is using FPGAs, which contain a programmable logic circuit instead of a hard-wired one. Such FPGAs can serve as processing elements for very specialized jobs. Several works have used FPGAs for database applications [WTA13, MTA09a]. The challenges in programming an FPGA and developing an instruction set extension1 are similar in many aspects. For instance, a decision has to be made which functionality is implemented in software and which is applied to the FPGA, e.g. for boolean expressions, Woods et al. compute a truth table in software and send it to the FPGA, where it serves as a lookup table for the query evaluation [WTA13]. However, the setup of a system using an FPGA is different from ours. While we enhance the hardware components themselves, i.e. the processors and the memory, an FPGA is an additional component which has to be connected to the host system. This introduces a data transfer overhead which has to be regarded when optimizing the query execution. Additionally a hard-wired circuit is usually much smaller than the programmed equivalent. This comes at the cost of reconfigurability. An implementation for an FPGA can be changed later whereas an instruction set extension is fixed once it is built. When it comes to the costs, an FPGA is expensive once when it is purchased while changing the configuration comes at almost zero cost. A processor or memory chip with additional hard-wired functionality becomes more affordable the more pieces are manufactured. This is because the initial modification of the manufacturing plant is the most expensive task, e.g. manufacturing the wafer. Hence, an FPGA is better suited for very specialized tasks which do not require a large quantity of chips whereas our approach is suitable for the mass.

Memory The bandwidth of memory buses and the capacity of (intermediate) memory is limited. In the memory hierarchy, bandwidth is growing more the closer the memory is to the processing element. But at the same time the capacity decreases. In data intensive applications this leads to high latencies which make the memory access a bottleneck in query processing. There are several approaches to overcome these architectural limitations.

One trend is to reduce the data that has to be sent to the processing elements. This is done by moving the functionality, which filters the data, closer to the memory. An early proposal using this concept is the *Intelligent Disk* (IDISK)[Ke98]. It combines a processor, DRAM and a fast network, on one chip (IRAM) which is put on top of the disk. This enables simple operations on the data before it is sent to the actual database engine. This way the data can be filtered before sending it to a smaller intermediate memory close to the main processing elements. Additionally the computing load for doing this filtering is taken away from the main processing elements. Instead of a relatively slow multi purpose processor, an FPGA can also be used as the component between the memory and the database engine. This has been implemented in IBM® *Netezza* [Fr11]. Here, an FPGA is installed between the memory and the processing nodes. It is responsible for filtering and transformation before the data is sent to the node. Finally, the already mentioned work by Woods et al. introduces a system called *Ibex* [WTA13], which uses an FPGA not as a usual processing element but as a filter between an SSD and the database engine.

Another trend is to decouple the dataflow from the actual processing in the processing elements. An approach for this direction is the SGI *global reference unit* (GRU). In the SGI Altix UV series, the GRU is a part of the UV Hub which connects the processors, the local SDRAM and the network interface. A UV Hub contains 2 GRUs which connect two processor sockets with the NUMAlink5 interconnect [SG10]. This way the whole system can be used as a single logical system including the memory modules. Since the GRU is programmable, it can be set up to execute instructions, e.g. remote copy of data, while the actual user application is doing something else.

However, the mentioned approaches focus on the interconnect between different sockets and their attached RAM or the database engine and the permanent memory, not on the connection between the RAM and the according processing element.

All Layers One step further than focusing on improving selected elements of existing systems, is to develop a completely revised architecture which can include changes on all 3 layers of the system design. On the large scale this includes heterogeneous architectures and changes in the system layout, e.g. the transfer of the memory controller from the north bridge to the CPU. On the small scale it addresses features like the out-of-order processing implementation. For instance, Hyperion-Core develops a core which is able to switch between different execution modes at runtime including a programmable loop-acceleration, out-of-order, asynchronous and the classical RISC/VLWI method[8]. These extensions and architectures address a wider area of usage than database systems. Hardware which is specialized in database applications could enhance the performance and energy efficiency even further.

Commercial Database Systems There have also been commercial approaches to hardware-software co-design for database systems. Oracle introduced *Software in Silicon* in their Spark M7 processors [Or15, Or16]. In detail, three aspects are optimized using custom made hardware: (1) application data integrity, (2) memory decompression, and (3) query processing performance. The first is meant to detect buffer overflows and memory access violations in main memory. Thus, it addresses the memory system layer (b). The second and the third aspect are incorporated in *Database Acceleration Engines*, which decompress data, filter values, and format results, all in one hardware accelerator. A database acceleration engine can be seen as a specialized processing element. This approach needs a hypervisor to queue work to a specific accelerator while the actual CPU cores submit the work asynchronously. On the other hand, our instruction approach extends each single core, which can execute each custom-made instruction in one cycle. Therefore, work is not asynchronously submitted and the core stays in control of the process.

6 Summary and Future Work

In this paper, we have given an overview of our work to support database engines by customizing the Tomahawk processor. In particular, we summarized our database-specific instruction set extensions for the processing elements as well as our iDMA approach for optimizing the main memory access by offloading appropriate logic to the DMA controller. Our experiments are done either using real hardware or using a cycle accurate simulator. Both evaluations show the high potential of our research work regarding performance and energy consumption, especially compared to a general-purpose Intel i7-6500U processor.

Generally, we envision a customized storage and data pre-processing hardware system for database engines. So far, our conducted research work is a first step in this direction. Next, we want to generalize the creation of database-specific instruction set extensions, so that we are able to transfer more database primitives into hardware extensions. After that, we want

[8] Hyperion Core - `http://hyperion-core.com`

to switch the view from a single PE to the complete processor. At this level, the interesting questions are: (i) which PE should contain which instruction set extensions (a single PE cannot offer all extensions due to thermal and space constraints) , (ii) how often does an extension have to be available at all, (iii) which extensions can be best combined, e.g. to reduce the space or co-locate frequently combined extensions. There are a lot of open research questions. Furthermore, the influence on query processing has to be examined, that means, how to do the query processing if we now have a large number of PEs, each offering different functionalities. From our point of view, this introduces a new dimension into query optimization.

Aside from the processing elements, we also want to extend our work on the iDMA controller approach. In addition to point queries, we also want to support range queries for different data structures. Furthermore, we want to investigate what we will be able to outsource to the iDMA, e.g. can we compress or decompress data on the iDMA controller. Nevertheless, this requires balancing our demands of universality and technical feasibility as described in Section 3.1. Furthermore, space and thermal constraints have to be considered as well. Afterwards, the influence on the query processing has to be investigated.

References

[An13] Anati, Ittai; Gueron, Shay; Johnson, Simon; Scarlata, Vincent: Innovative technology for CPU based attestation and sealing. In: HASP. volume 13, 2013.

[APD15] Arulraj, Joy; Pavlo, Andrew; Dulloor, Subramanya R.: Let's Talk About Storage - Recovery Methods for Non-Volatile Memory Database Systems. In: SIGMOD. pp. 707–722, 2015.

[Ar14a] Arnold, Oliver; Haas, Sebastian; Fettweis, Gerhard; Schlegel, Benjamin; Kissinger, Thomas; Karnagel, Tomas; Lehner, Wolfgang: HASHI: An Application Specific Instruction Set Extension for Hashing. In: ADMS@VLDB. pp. 25–33, 2014.

[Ar14b] Arnold, Oliver; Haas, Sebastian; Fettweis, Gerhard; Schlegel, Benjamin; Kissinger, Thomas; Lehner, Wolfgang: An application-specific instruction set for accelerating set-oriented database primitives. In: SIGMOD. pp. 767–778, 2014.

[Ar14c] Arnold, Oliver; Matus, Emil; Noethen, Benedikt; Winter, Markus; Limberg, Torsten; Fettweis, Gerhard: Tomahawk: Parallelism and heterogeneity in communications signal processing MPSoCs. TECS, 13(3s):107, 2014.

[Bi15] Birrittella, Mark; Debbage, Mark; Huggahalli, Ram; Kunz, James; Lovett, Tom; Rimmer, Todd; Underwood, Keith D.; Zak, Robert C.: Intel Omni-Path Architecture, Enabling Scalable, High Performance Fabrics. Technical report, Intel Corporation, 2015.

[DP10] Deliège, François; Pedersen, Torben Bach: Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps. In: EDBT. 2010.

[Es11] Esmaeilzadeh, Hadi; Blem, Emily; St. Amant, Renee; Sankaralingam, Karthikeyan; Burger, Doug: Dark Silicon and the End of Multicore Scaling. In: ISCA. 2011.

[Fa12] Farber, Franz; Cha, Sang Kyun; Primsch, Jurgen; Bornhovd, Christof; Sigg, Stefan; Lehner, Wolfgang: SAP HANA Database: Data Management for Modern Business Applications. SIGMOD Rec., 40(4):45–51, 2012.

[Fi08] Firasta, Nadeem; Buxton, Mark; Jinbo, Paula; Nasri, Kaveh; Kuo, Shihjong: Intel AVX: New frontiers in performance improvements and energy efficiency. Intel white paper, 2008.

[Fr11] Francisco, Phil et al.: The Netezza data appliance architecture: A platform for high performance data warehousing and analytics. IBM Redbooks, 3, 2011.

[FSV10] Fusco, Francesco; Stoecklin, Marc Ph.; Vlachos, Michail: NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic. PVLDB, 2010.

[Ha16a] Haas, Sebastian; Arnold, Oliver; Nöthen, Benedikt; Scholze, Stefan; Ellguth, Georg; Dixius, Andreas; Höppner, Sebastian; Schiefer, Stefan; Hartmann, Stephan; Henker, Stephan et al.: An MPSoC for energy-efficient database query processing. In: DAC. p. 112, 2016.

[Ha16b] Haas, Sebastian; Arnold, Oliver; Scholze, Stefan; Höppner, Sebastian; Ellguth, Georg; Dixius, Andreas; Ungethüm, Annett; Mier, Eric; Nöthen, Benedikt; Matus, Emil Matus; Schiefer, Stefan; Cederstroem, Love; Pilz, Fabian; Mayr, Christian; Schüffny, René; Lehner, Wolfgang; Fettweis, Gerhard: A Database Accelerator for Energy-Efficient Query Processing and Optimization. In: NORCAS. 2016. to appear.

[Ha16c] Haas, Sebastian; Karnagel, Tomas; Arnold, Oliver; Laux, Erik; Schlegel, Benjamin; Fettweis, Gerhard; Lehner, Wolfgang: HW/SW-Database-CoDesign for Compressed Bitmap Index Processing. In: ASAP. 2016.

[He09] He, Bingsheng; Lu, Mian; Yang, Ke; Fang, Rui; Govindaraju, Naga K.; Luo, Qiong; Sander, Pedro V.: Relational Query Coprocessing on Graphics Processors. ACM Trans. Database Syst., 34(4), 2009.

[In08] Intel Corporation: Intel QickPath Architecture. Technical report, 2008.

[Ke98] Keeton, Kimberly; Patterson, D; Hellerstein, Joseph; Kubiatowicz, John; Yelick, Katherine: The intelligent disk (IDISK): A revolutionary approach to database computing infrastructure. Database, 9(6 S 5), 1998.

[KSL13] Kiefer, Tim; Schlegel, Benjamin; Lehner, Wolfgang: Experimental Evaluation of NUMA Effects on Database Management Systems. In: BTW. pp. 185–204, 2013.

[Me07] Mellanox Technologies: InfiniBand Software and Protocols Enable Seamless Off-the-shelf Applications Deployment. Technical report, 2007.

[MTA09a] Mueller, Rene; Teubner, Jens; Alonso, Gustavo: Data processing on FPGAs. PVLDB, 2(1):910–921, 2009.

[MTA09b] Mueller, Rene; Teubner, Jens; Alonso, Gustavo: Streams on Wires: A Query Compiler for FPGAs. PVLDB, 2(1):229–240, 2009.

[Or15] Oracle Software in Silicon. Oracle Open World Presentation, 2015.

[Or16] Oracle's SPARC T7 and SPARC M7 Server Architecture. White Paper, 2016.

[SG10] SGI® Altix® UV GRU Development Kit Programmer's Guide, 2010.

[We15] Wei, Xingda; Shi, Jiaxin; Chen, Yanzhe; Chen, Rong; Chen, Haibo: Fast In-memory Transaction Processing Using RDMA and HTM. In: SOSP. pp. 87–104, 2015.

[WOS04] Wu, Kesheng; Otoo, Ekow J; Shoshani, Arie: An efficient compression scheme for bitmap indices. Lawrence Berkeley National Laboratory, 2004.

[WTA13] Woods, Louis; Teubner, Jens; Alonso, Gustavo: Less watts, more performance: an intelligent storage engine for data appliances. In: SIGMOD. pp. 1073–1076, 2013.