Oliver Arnold, Sebastian Haas, Gerhard Fettweis, Benjamin Schlegel, Thomas Kissinger, Wolfgang Lehner

An Application-Specific Instruction Set for Accelerating Set-Oriented Database Primitives

Erstveröffentlichung in / First published in:

SIGMOD/PODS'14: International Conference on Management of Data, Chicago 22. – 27.06.2017. ACM Digital Library, S. 767–778.

DOI: https://doi.org/10.1145/2588555.2593677

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 767–778, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2593677

An Application-Specific Instruction Set for Accelerating Set-Oriented Database Primitives

Oliver Arnold, Sebastian Haas, Gerhard Fettweis
Vodafone Chair Mobile Communications Systems
Technische Universität Dresden
Dresden, Germany

Benjamin Schlegel*, Thomas Kissinger, Wolfgang Lehner
Database Technology Group
Technische Universität Dresden
Dresden, Germany

ABSTRACT

The key task of database systems is to efficiently manage large amounts of data. A high query throughput and a low query latency are essential for the success of a database system. Lately, research focused on exploiting hardware features like superscalar execution units, SIMD, or multiple cores to speed up processing. Apart from these software optimizations for given hardware, even tailor-made processing circuits running on FPGAs are built to run mostly stateless query plans with incredibly high throughput. A similar idea, which was already considered three decades ago, is to build tailor-made hardware like a database processor. Despite their superior performance, such application-specific processors were not considered to be beneficial because general-purpose processors eventually always caught up so that the high development costs did not pay off. In this paper, we show that the development of a database processor is much more feasible nowadays through the availability of customizable processors. We illustrate exemplarily how to create an instruction set extension for set-oriented database primitives. The resulting application-specific processor provides not only a high performance but it also enables very energy-efficient processing. Our processor requires in various configurations more than 960x less energy than a high-end x86 processor while providing the same performance.

Categories and Subject Descriptors

C.1 [**PROCESSOR ARCHITECTURES**]: Other Architecture Styles;; H.2 [**DATABASE MANAGEMENT**]: Systems

Keywords

Hardware/Software-Codesign, Customizable Processors, Instruction Set Extensions

*This author is now at Oracle Labs, Belmont, CA, USA.

DOI: http://dx.doi.org/10.1145/2588555.2593677.

1. INTRODUCTION

In addition to many beneficial features of a database system, excellent query performance is still a decisive factor for many applications scenarios. Database systems are therefore optimized in many different directions; efficient index structures [32, 18], parallel sorting [6, 12], efficient query operators [38, 4], fast compression [36, 26] are only some examples to speed up query processing. The key idea of most of these optimizations is to exploit hardware features like multiple cores, SIMD, and multiple execution units or to adapt to hardware characteristics like cache hierarchies. Algorithms deployed in database systems are therefore highly tuned and very often either reach the processor's peak performance or are limited by some system characteristics like memory bandwidth or interconnect capacity.

To push the envelope in database performance even further, we may look into different directions. First, parallelism can be enhanced within a single system by adding more and more components and cores. Unfortunately, thermal restrictions of current processor designs are a limiting factor and force to build larger and more distributed systems with higher latencies between the different components. Second, processors themselves can be improved to allow for a higher query throughput and lower latency by using specialized hardware optimized for query processing functionality. Features like instruction sets, cache sizes, memory bus width, number of cores, among others can be adapted to meet query processing needs.

Within this paper, we tackle both challenges by providing a specialized instruction set for set-oriented database primitives in combination with a low-energy processor design. The extremely low-energy design enables us to put hundreds of chips on a single board without any thermal restrictions. This work also tries to spark the discussion on real HW/SW co-design and calls for specialized hardware optimized for query processing following this observation: It is well known that the single-threaded performance of processors has almost stopped increasing because the maximum core frequency is limited by physical constraints. Unfortunately, the current solution to put more and more homogeneous cores onto a single socket will also reach physical limitations. As the feature size in which processors are manufactured will go smaller and smaller, the number of transistors increases. However, not all of those transistors can be supplied with power at the same time. This *dark silicon* [10, 13] will soon cover large fractions of the chip space.

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 767–778, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2593677

Since placing more homogeneous cores on the chip does then obviously not help in terms of performance anymore, but specialized circuits in form of instruction set extensions or heterogeneous cores make a lot of sense [13].

As a proof point to demonstrate the feasibility of a specialized processor design, we tackle the challenge to optimize set-oriented database primitives. For example, sorting, set and string operations, indexing, compression, encryption and partitioning, are good candidates for being processed with specialized circuits. They are typically responsible for a large fraction of the overall query runtime and follow—besides different data types—often a predefined algorithm.

While there already exist instruction set extensions in general-purpose processors to speed up string operations and encryption (via SSE4.2 and AES), it is unlikely that more database-specific instructions will be integrated in such processors to keep their generality. In the opposite, database processors [5, 3, 28, 19, 2], which have been developed in the 1980s, include almost only functionality that is required for an efficient query processing in database systems. Despite their superior performance, they have not been considered as beneficial because (1) the I/O bandwidth of disks showed to be the bottleneck of query processing and (2) the high development costs did not pay off.

Since modern database architectures are mostly main-memory centric, the argument (1) against database processors does not hold any longer. In the opposite, modern database engines require the optimization of memory access patterns and efficient cache exploitation. Within this paper, we also want to argue against (2): Building a complete processor from scratch is obviously still expensive and therefore prohibitive. However, *customizable processors* considerably reduce the development effort by providing a configurable base processor, which can be extended with application-specific instruction sets. Therefore almost 30 years after the initial attempt to build database processors, we demonstrate the feasibility and show extremely encouraging results. The techniques for developing application-specific processors proposed in this paper can be easily reused to obtain instruction sets for other (and even more complex) database primitives and may trigger research for a second wave of database processors.

In this paper, we show how to build an instruction set extension for a customizable processor. As a show case, we picked set-oriented database primitives. These primitives can be used to speed up sorting, RID-list intersections, and union operations, which often amount for a significant fraction of the overall runtime of queries in database systems. Our key contributions can be summarized as follows:

(1) We discuss the components of processors, shortly introduce customizable processors, and show techniques for improving them. We further extensively review related work in the area of database processors.
(2) We propose the system model of a processor that is tailor-made for data-intensive tasks and provide a detailed description for obtaining such a processor based on the Tensilica customizable processor.
(3) We discuss a novel instruction set extension to speed up set-oriented database primitives. The instructions can be used, for example, for efficient sort algorithms or set operations on RID sets. We discuss each instruction in very detail–including their implementation in hardware and their usage within database engine code.
(4) We provide a broad evaluation of the system model and the instruction set extension. Besides a performance comparison, we evaluate the required chip space of the instructions, their impact on the processor's core frequency, and the processor's energy consumption. We further compare the performance of algorithms running on our processor with existing highly-optimized algorithms running on modern x86 processors.

The remainder of this paper is structured as follows: We describe the necessary background in Section 2. Section 3 provides details with respect to the development strategy and the design of our application-specific processor. In Section 4, we explain our instruction set extension in more detail whereas Section 5 provides results of our vast experimental evaluation. The paper concludes with an overall summary.

2. BACKGROUND

This section provides the necessary background of the paper. We review the processor components that are required to understand this work and discuss shortly available customizable processors. We further give an overview on (1) techniques for building application-specific instructions and (2) the database primitive being optimized.

2.1 Customizable Processors

Although there exists a wide variety of processors, all can be reduced to a simplified processor model (see Figure 1). In what follows, we describe the components of such a model.

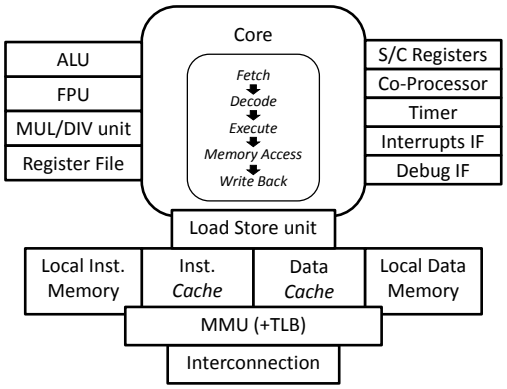

Figure 1: State-of-the-art processor model

A single *core* of a processor is responsible for executing instructions. The core itself consists of several components: The *arithmetic logic unit* (ALU) is used to perform instructions working on integer values including, among others, arithmetic, comparison, and shift instructions. The *floating-point unit* (FPU) has basically the same task as the ALU but processes only floating point values. It usually supports a different set of instructions since many floating point instructions (e.g., rounding) have no integer counterparts. Similarly, the *MUL/DIV unit* handles multiply and division instructions. Besides the units required for processing instructions, the *register file* is used to stage data between memory and these units.

Most instructions can only process data located in these registers. Hence, data must be loaded before an instruction can be performed. The *status registers* (S/C registers) hold flags that are set or unset by the instructions, e.g., the zero,

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 767–778, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2593677

carry, and overflow flag. The status register also influences the control flow. Additional *co-processors* may support certain specialized instruction sets. Finally, the *timer* provides a clock signal; the *interrupt* and *debug interface* allow the operating system to interact with the core.

A single or multiple load–store units link the core to the memory and potentially to other cores. The *instruction cache* and *data cache* are directly connected to these units and temporary cache instructions and data elements for a fast access. Multiple caches can be organized in a hierarchy and exhibit different access speeds. Some processors show a *local instruction memory* and *local data memory*[1], which have the same access speed as the caches but are maintained by the application program. The *memory management unit* (MMU) together with the *translation lookaside buffer* (TLB) are used for implementing virtual memory and to handle memory requests on the *interconnection network*.

Since building a full processor and its components from scratch is prohibitive, customizable processors are available to ease the development of specialized processors. Examples are CoWare LISATek [7], Tensilica LX4 [34], and ARC 750 [1]. All these processors provide a basic core, which can be extended by individual components, e.g., a floating point unit or an application-specific instruction set. Besides adding components, the basic cores can also be parametrized, e.g., the number of pipeline stages, the bus width, and the size of the local memory can be altered.

2.2 Optimization Techniques

Customizable processors could be improved in a large number of ways. In what follows, we outline techniques that are used in this paper and discuss their potential as well as their limitations.

The most promising way to improve processors is to extend them with application-specific instructions. These instructions often combine multiple existing instructions to a single instruction, which are repeatedly performed together within an application setting. A well-known example for such an *instruction merging* are the fused multiply-add instructions, which combine an add and a multiply instruction each and are widely available in modern processors.

However, customizable processors allow to build even more complex instructions. Calculating a CRC value, for example, requires shift, comparison, and XOR instructions, which can all be combined into a single instruction. The time for performing the CRC operation thus depends only on the latency of the single new instruction instead of the latency of the sequence of the core instructions.

Further and well-known opportunities also exist for bit manipulation instructions, because they are often simple to implement in hardware but require many shift and mask instructions when realized in software. For example, reversing the order of the bits in a 32-bit word is cheap in hardware whereas it requires dozens of instructions in software.

Unfortunately, there exist two problems for instruction merging. Firstly, the more instructions are merged into a single instruction, the more *specialized* (and less *general*) becomes the instruction and the less it may probably be used. Hence, the processor developer has to carefully select which instructions and how many of them should be merged because otherwise valuable chip space may be wasted with instructions that are rarely used. Secondly, the *critical path*, which is the longest path of any circuit representing an instruction, might be largely increased when many instructions are merged into a single one. The core frequency, however, is determined by the critical path because the time for a single cycle must be sufficient to allow a signal passing the complete circuit of any instruction. Therefore, instructions must be well designed to minimize the length of the critical path and thus allow a high core frequency.

Adding SIMD instructions based on heavily-used scalar instructions is an other way to improve customizable processors. SIMD instructions perform the same operation on multiple data elements. In what follows, we distinguish between two types of SIMD instructions. *Element-wise instructions* treat the elements of the vectors independently. A SIMD add instruction, for example, always adds the elements that have the same position in the input vectors; n addition circuits are used when two vectors with n elements are added. The area occupied by element-wise instructions thus grows linearly when the vector width is linearly increased. The main limitation of SIMD instruction is the bandwidth to main memory, which may not be arbitrarily increased. Alternatively, *intra-element wise instructions* act across the elements in the vectors being processed. For example, the all-to-all comparison instruction of SSE4.2 compares all elements of one vector with all elements of a second vector; its implementation in hardware uses n^2 comparison circuits when the vectors have length n. Depending on the operation, the area occupied by intra-element wise instructions grows more than linear (e.g., quadratic) when the vector length is linearly increased. Therefore, a tradeoff between the performance improvement through increasing the vector width and the area required for the instruction must be found. *In this paper, we combine both techniques and create novel instructions by merging existing instructions and building SIMD versions of them.*

In addition to build new instruction set extensions, the execution of instructions itself has to be optimized. *Pipelining* is a popular concept to reduce the critical path and thus increasing a processor's core frequency. In an n stage pipeline, the instructions respective circuits can be divided into n parts, which ideally divides the critical path by n. Pipelining, however, makes the process of building instructions more complex and not every application code benefits from it. *Superscalar pipelining* even provides multiple execution units in which instructions or parts of them could be processed in parallel. *In this paper, we use a form of superscalar pipelining where multiple load–store units are used by a single instruction at the same time.*

2.3 Sorting and Set Operations in Databases

As mentioned earlier, we build an instruction set extension that is used to speed up (1) sorting and (2) operations performed on sorted sets. We choose these operations because they are heavily used within database systems and typically amount to a large fraction of the overall time required for query processing [17, 6, 9, 31]. Sorting, for example, is used before sort-merge joins [4] or whenever the result set should be returned in a specific order. Set operations like intersection, difference, or union are often performed on RID sets [31], which are obtained from secondary indices when complex selection predicates within the `WHERE` clause are specified, or when a query contains a `INTERSECT`, `UNION`, or `DIFFERENCE` clause.

[1]This is often denoted as *scratchpad memory* or *local-store*.

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 767–778, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2593677

```
void merge(int* A, int* B, int l_a, int l_b, int* C) {
  int pos_a = 0, pos_b = 0, pos_c = 0;
  while( pos_a < l_a && pos_b < l_b ) {
    if( A[ pos_a ] < B[ pos_b ] )
      C[ pos_c++ ] = A[ pos_a++ ];
    else
      C[ pos_c++ ] = B[ pos_b++ ];
  }
  while( pos_a < l_a ) C[ pos_c++ ] = A[ pos_a++ ];
  while( pos_b < l_b ) C[ pos_c++ ] = B[ pos_b++ ];
}
```

Figure 2: Merge procedure of merge-sort

In the following, we discuss the merge-sort algorithm, which is among the most efficient sorting algorithms [6]. We also touch upon merge-based set-operations algorithms based on sorted sets. We will limit the discussion to only the algorithms' key aspects that are necessary to understand the main concepts shown in this paper.

The *merge-sort algorithm* [23] is a divide and conquer algorithm with $O(n \log n)$ time complexity. Its key idea is to repeatedly merge sorted sequences to obtain larger sorted sequences. The algorithm starts from sequences of length 1, which are inherently sorted. With each iteration, the length of the sorted sequences doubles until after $\log n$ iterations the input sequence is sorted completely. Merge-sort spends most of its time in the merge procedure so that its optimization is key to a high sorting performance.

Figure 2 depicts the C-code of an implementation of the merge procedure. The procedure takes two sorted sequences `A` and `B` with length `l_a` and `l_b`, respectively, and a reference to the result sequence as input. Although merge-sort can sort in-place, we depict an out-of-place implementation for ease of explanation. After initializing variables, the core loop is repeated until the end of one input sequence is reached. Within the core loop, both input sequences are stepwise traversed using two indices `pos_a` and `pos_b`; the smaller value of both sequences is written into the result set and only the index of the input sequence from which the value was taken is increased. After the core loop is left, remaining values of the set that is not fully traversed are written at the end of the result set.

The most expensive part of the merge procedure is the hardly predictable branch. It reduces the effect of pipelining and therefore degrades performance. To avoid branches in the core loop and exploit the SIMD capabilities of modern processors, Chhugani et al. [6] proposed to repeatedly use small sorting networks, which are implemented with SIMD instructions, to initially create and merge sorted sequences. Basically, up to 8 values are loaded into SIMD register; shuffle and min–max SIMD instructions are used to order the values before writing them back to memory. *In this paper, we reuse this idea but instead of issuing multiple instructions sequentially, we realize the sorting network in hardware and issue only two instructions to sort four values.*

Merge-based set-operations algorithms for intersection, difference, and union work very similar to the merge procedure of the merge-sort. The result sequence, however, contains the values of the performed set operation.

Figure 3 depicts the C-code for a function that intersects two sorted sets `A` and `B`. The parameter of the function as well as the variables and their initialization are the same as in the merge procedure (cf. Figure 2). The core loop, however, is slightly different and contains two branches. They are used to choose one of three different actions based on the comparison of the current two values taken from both input sets. The union and difference implementations differ only in the actions they perform and in the way remaining values of the input sets might be copied.

```
int intersect(int* A, int* B, int l_a, int l_b, int* C) {
  int pos_a = 0, pos_b = 0, pos_c = 0;
  while( pos_a < l_a && pos_b < l_b ) {
    if( A[ pos_a ] == B[ pos_b ] ) {
      C[ pos_c++ ] = A[ pos_a ];
      pos_a++;
      pos_b++;
    }
    else if( A[ pos_a ] < B[ pos_b ] )
      pos_a++;
    else
      pos_b++;
  }
}
```

Figure 3: Sorted-set intersection

Despite their simplicity, the merge-based algorithms are among the best algorithms for performing set operations. Only few optimizations exist for them—see Ding and König [9] for an overview. In earlier work [33], we proposed to use SIMD instructions to speed up the intersection operation. Basically, we use the aforementioned all-to-all comparison instruction on two vectors containing 8 values of each set. Although, several instructions are still issued per iteration in the core loop, the indices of at least one input set are increased by eight values instead of one. *Based on this idea, we build instructions for all three set-operations resulting in only two instructions per core-loop iteration.*

3. PROCESSOR DESIGN

In this section, we explain the tool chain employed to generate a processor and the respective development strategy used to obtain efficient instructions. We further describe the model of the processor in which we integrate our novel instruction set extension.

3.1 Tool Chain and Development Strategy

The development strategy is divided in two main parts: *simulation* to obtain the application-specific instruction set and *synthesis* of the processor to generate results for area, power, and timing. Figure 4 shows the respective tool flow, i.e., steps performed in both parts and how they interact.

The tool flow starts with a cycle-accurate profiling of an application to analyze its runtime behavior. The profiler unveils hotspots in the application's execution, i.e., frequently executed and computationally intensive parts. Based on the obtained results and their analysis, a new instruction set can be specified and attached to a customizable processor. Furthermore, extension units can be added to the core. In this work, for example, we attach an additional load–store unit to allow concurrent data transfers and potentially doubling the memory bandwidth of the processor. We will give further details on the extension units in Section 3.2.

After specifying the instruction set, a processor generator creates a cycle–accurate simulation model of the processor as well as a suitable compiler. The newly introduced instructions are made available by intrinsics. They replace certain parts of the original application code. An example is shown in Section 4.

After generating the processor model, the verification is performed, e.g., by applying unit tests, regression tests or

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 767–778, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2593677

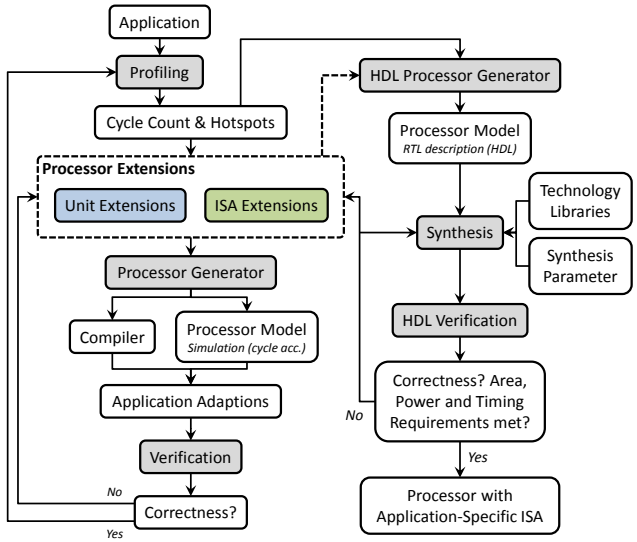

Figure 4: Tool flow for developing instruction set extensions

equivalence checks. In our work, we use a dedicated unit test for each newly introduced instruction. The unit tests compare output results with pre-specified values—especially considering corner cases.

If the correctness of the processor model is assured, further optimizations are done, i.e., another profiling and instruction set development iteration is performed. Due to the integration of the newly introduced instructions in the previous iteration cycle, the processor performance is improved. Thus, the bottlenecks of the application in this iteration cycle are either reduced or shifted to different parts of the application. Consequently, these parts are now in the focus of the instruction set extension development.

As soon as the improvement of the processor is exhausted, a hardware description language (HDL) processor generator creates a synthesizable model on register transfer level (RTL). It can be used for silicon prototypes. The iteration cycle of the HDL processor generator as well as the synthesis and verification cycle is several orders of magnitude more time consuming compared to the iteration cycle of the instruction set extension development. The latter is in the range of some minutes.

Finally, the HDL verification assures the correctness of the results of the synthesized processor model. If the area demands (die size), power consumption as well as timing requirements are met, the application-specific processor is ready for production. If not, either the newly developed instruction set or the synthesis needs to be adapted.

3.2 Processor Model

Our application-specific processor uses an Tensilica LX4 RISC processor as basis. The LX4 RISC processor can be extended with further units and a new instruction set. In our case, we add two units and database-specific instructions to the processor. Figure 6 illustrates the processor model of our processor with our modifications highlighted. In what follows, we describe (1) how instructions are created and (2) the employed additional units in more detail.

For our new instructions, we employ a VLIW instruction format called Flexible Length Instruction Xtension (FLIX) with the instruction width set to 64 bit. Additionally, we introduce two types of additional internal memories into the core: *TIE registers* and *TIE states*. Both can be varied in number of instances, size and bit width. Registers are used for generic memory access whereas states are typically used to store data for application-specific parts of the processor, i.e., predefined instructions access (read or write) these states in the same cycle the instructions are executed.

Figure 5 provides an example of a definition of a state, a register file, an instruction definition as well as the corresponding C-code. In this example, an 8-bit state with the name *state8* is firstly instantiated and initialized with zeros every time the processor is powered-on. The *add_read_write* statement allows to access the content of the state within the C-code of the application. Macros are automatically generated for read and write operations.

A user-defined register file is shown in part b) of the example. Eight 32-bit width registers are generated. All TIE instructions can use such a register file. In contrast to states, register allocation as well as load and store of data is managed by the compiler. In the latter case, it is the user's responsibility to manage the content of the states. Consequently, explicit load and store operations are required.

Figure 5 c) illustrates the newly integrated example instruction *add3_shift*. In the first line, the *operation* keyword starts the definition of an instruction, followed by its name. The register access is specified in the next line. The general-purpose Address Registers (AR) are part of the basic register file of the processor, which are used to store the result. The previously defined register file reg32 contains the input operands. Furthermore, the state *state8* is defined as an input argument as well. The keywords *in* and *out* specify the direction of the state and register file access of the operation. The last line defines the functional behavior. In this example, we use a TIE assignment statement: The three register values are added and the their sum is shifted by the values in state8. Note that the instruction is executed within only a single clock cycle.

The corresponding C-code for an application is shown in Figure 5 d). The new instruction uses three values of the new register file as input arguments and *state8* is set to 4. Register initialization is not shown. The instruction *add3_shift* itself has to be used with an automatically generated intrinsic. Finally, the result is stored in a user-defined integer value. In Section 4, we will introduce our instruction-set extension, which is obtained in a similar way.

```
// a) state definition
state state8 8 8'h0 add_read_write

// b)  register definition
regfile reg32 32 8 reg

// c) instruction definition
operation add3_shift
{out AR res, in reg32 in0, in reg32 in1, in reg32 in2}
{in state8}
{assign res = (in0 + in1 + in2) >> state8;}

// d) C-code
reg32 v0, v1, v2; WUR_state8(4);
int value = add3_shift(v0, v1, v2);
```

Figure 5: State-, register-, instruction-specification and C-code

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 767–778, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2593677

Besides the instructions, we integrate a second *load–store unit* and a *data prefetcher* (cf. Figure 6). The additional load–store unit is mainly used to improve memory latency and bandwidth. The data prefetcher is included to perform data transfers over the on–chip interconnection network. It contains a direct–memory access controller (DMAC) and a programmable finite state machine (FSM). The latter is used to control the DMAC. The programming of the FSM is done either by the processor itself or by another entity in the system. The data transfers of the *data prefetcher* and processor execution are performed concurrently. Consequently, it fetches data prior it is needed. Results are written back while the next operator has already started its execution. To allow this mode of operation, we include dual-port local memories, which support two memory accesses to occur at the same time. One port is connected to the processor, the other to the *data prefetcher* and the interconnection network. In contrast to caches, no cache-misses occur and the cache logic can be omitted. Hence, area is reduced but the processor in this work has no direct access to the interconnection network. It solely operates on the local instruction and data memory. By using this approach data locality is maximized and system performance may thus be enhanced.

The *data prefetcher* uses furthermore burst transfers, typically in the order of several KB to improve memory access patterns. Burst transfers can potentially improve the interconnection network usage and the access to the off–chip memory. Consequently, the observed bandwidth is increased and performance enhanced.

The following parts of the processor can be additionally adapted, but are not regarded in this work: TIE queues, TIE ports and TIE lookups. TIE queues read or write data from external queues. TIE input and output ports define a dedicated interface from the outside of the processor to internal states. TIE lookups request data from external devices.

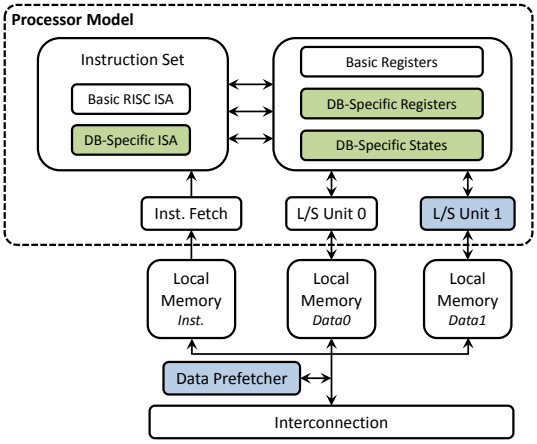

Figure 6: Processor model schematic

4. INSTRUCTION SET EXTENSION

In this section, we introduce our novel instruction set extension. Table 1 gives an overview of the instructions and a short summary of their functionality. All five instructions are executed within an iteration of our example set-based algorithms. In what follows, we explain (1) the instruction ordering as well as the instruction pipeline and (2) how the instructions are integrated into the implementation of the algorithms.

Inst.	Explanation
LD	This load instruction is available for each Load-Store Unit (LSU)—LD_0 for LSU0 and LD_1 for LSU1. For each LSU up to 128 bits are loaded from the local memory to the intermediate `Load` states of the processor.
LD_P	Is available for each LSU—LD_P_0 for LSU0 and LD_P_1 for LSU1. It (partially) reloads the words from the `Load` states to the `Word` states for each set. Hence, it is ensured that after each operation all `Word` states are fully filled with elements.
SOP	It performs the actual sorted-set operation based on an all-to-all comparison. It is applied on 4 elements of each set. All results are saved in the `Result` states.
ST_S	Reorders and shuffles the results of the SOP operation and save them to the `Store` states.
ST	Stores the results from the `Store` states to the local memory. Due to the word aligned memory access 128 bits are always written by the LSU.

Table 1: Instruction overview

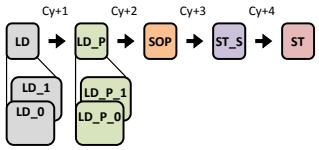

Figure 7: Instruction ordering

The overall instruction ordering is illustrated in Figure 7. Firstly, the LD instruction loads the data from the local memory to the processor's intermediate states. Afterwards, this data is partially or fully moved and reordered (LD_P) to a second internal state. Thus, the loading and the reordering is decoupled for a reduction of the critical path while loading data. The sorted-set operations (SOP) perform the actual execution of the database-specific tasks. The obtained results are shuffled to guarantee correct ordering of the resulting elements (ST_S). For this reason, we implement a first-in first-out like implementation. The store instruction (ST) saves the results in the local memory.

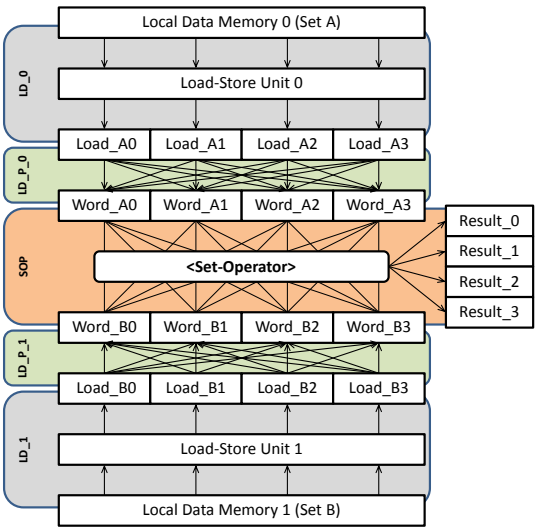

Figure 8: Load, pre-execution reordering, and execution phase

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 767–778, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2593677

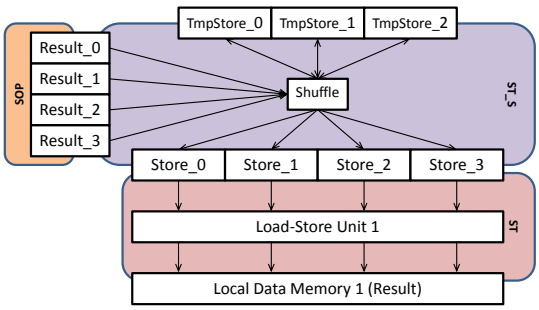

Figure 9: Pre-store reordering and store phase

An example data flow for 32-bit wide elements is depicted in Figure 8 and 9 for the load and execution as well as the store phase respectively. It uses partial loading and result reordering to store the data. Firstly, eight elements are loaded with two Load-Store Units (LSUs) to the `Load` states. Each of them is equipped with a 128-bit interface to the local memory. As soon as the data is present in the processor and is partially reordered according to the available elements of the `Word` states, always four elements per set are ready for comparison. An all-to-all comparison performs the actual work, e.g., obtaining common values within the loaded values from both input sets. The results are stored in the `Result` states and are prepared for storage with a shuffle network (see Figure 9). It assures, that always four elements are written back from the `Store` states to the local memory. The store instruction is delayed in the case of three or less available elements. As soon as four elements are present, the store operation is performed. The `TmpStore` states are used as intermediate buffers.

The pipeline of the processor is depicted in Figure 10. A snippet during the execution is shown in which in each cycle either two times 128-bits are loaded from the memory or 128-bits of results are written back to the memory. Therefore, the memory interface is the limiting factor at this point of time. The latency is six cycles. The maximum theoretical throughput is the number of elements, which can be loaded with two LSUs multiplied by the clock frequency of the processor. Since every two cycles eight elements are loaded from the local memory, a maximum theoretical throughput of 2,000 million elements per second at a clock frequency of 500 MHz would be achieved.

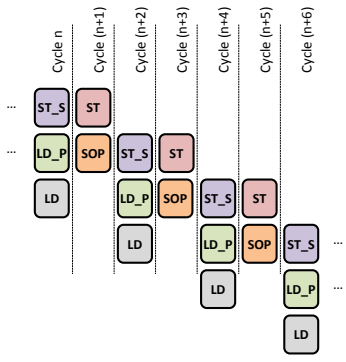

Figure 10: Pipeline snippet

```
INIT_STATES(); // state initializations and initial load
LD_LDP_SHUFFLE();

while( STORE_SOP() ) { // core loop
  LD_LDP_SHUFFLE();
}
```

Figure 11: Sorted-set intersection core loop with TIE instructions

Figure 11 illustrates the initialization and the core loop of a sorted-set implementation that exploits our new instructions. The initialization and the first load takes one cycle each. One iteration of the core loop requires only three cycles. One cycle is required for the actual operation and the store. A second cycle is required to load the data before the next evaluation. An additional cycle is required to evaluate the loop condition.

`LD_0`, `LD_1`, `LD_P_0`, `LD_P_1`, and `ST_S` are concurrently executed by integrating all parts in a fused `LD_LDP_SHUFFLE` instruction. In the final implementation the code is further accelerated by applying loop unrolling. Consequently, the average number of cycles per loop is reduced. For example, if 32 loops are unrolled the average number of cycles per loop is reduced to 2.03.

```
INIT_STATES(); // state initializations and initial load
LD();

while( LD() ) { // core loop
  STORE_MERGE();
}
```

Figure 12: Merge-sort core loop with TIE instructions

Figure 12 illustrates the core loop of the merge-sort implementation that exploits our new instructions and merges sorted sequences. All elements are loaded as well as stored and no elimination of duplicates is necessary. Consequently, the shuffle instruction (ST_S) and the LD_P instruction are not applied in this case. The LD instruction loads always from LSU0 with instruction LD0 and thereafter the merge instruction compares eight values. The additional presorting, i.e., building sorted sequences of four sorted values, is not shown in this example. For this purpose, special load and store instructions exists, which concurrently perform a sort operation to built such sequences. Furthermore, as soon as one list is empty the remainder elements of the other lists are copied using 128-bit copy instructions.

5. EXPERIMENTAL RESULTS

This section contains the results of our experimental evaluation. We provide details to the used experimental setup, compare the performance for different processor models, and evaluate the area and power consumption of our implementations. Finally, we compare two algorithms employing our instructions with two existing highly-optimized algorithms that run on modern x86 processors.

5.1 Setup

We use the configurable Tensilica Xtensa LX4 Core as foundation of our processor. In the following, it is called DBA_1LSU. The DBA_1LSU processor has similar features like the 108Mini[2] processor, but includes a 64KB local data

[2] `http://www.tensilica.com/uploads/pdf/108Mini.pdf` contains more information about this processor.

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 767–778, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2593677

store, which is accessable via a single load–store unit (LSU). Furthermore, the instruction and data bus width was increased from 32 to 64 and from 32 to 128 bit, respectively. In contrast to the 108Mini no hardware support for integer division is available. The configurations DBA_1LSU_EIS and DBA_2LSU_EIS further contain our instruction set extension from Section 4. The difference between both configurations is that the latter contains two load–store units. Each of them is equipped with its own local data memory. The data memory is equally distributed between both memories. Hence, 32KB are accessable by each LSU. All DBA processors have 32KB local instruction memory.

For comparison a Tensilica Diamond standard processor 108Mini was chosen. It is a mid-sized controller with additional instructions for digital signal processing. Furthermore, the DBA_2LSU processor is synthesized to obtain area, maximum frequency and power consumption. These can be compared to the DBA_1LSU processor to analyze the impact of the second LSU. Nevertheless, the compiler is not able to make use of it. Consequently, performance is the same and will not be further evaluated.

All versions of the processors have been synthesized with Synopsys Design Compiler for a 65 nm low-power TSMC process using typical case conditions (25°C, 1.25 V). The processor as well as the memories are regarded. For the latter low power TSMC libraries are used. Additionally, the same tool flow was used with a 28 nm super low-power (SLP) Global Foundries process, including super low voltage (SLVT) parameters. Typical case conditions are applied (25°C, 0.8 V).

Power Consumption was simulated as follows: the core and the memories have been synthesized as previously described. Results of this step are a net list on gate level as well as a Standard Delay Format (SDF) file with timing information. Simulation of representative test cases were performed in Mentor Questa. Hereby, a dump file is created containing all switching activity of the entity. In a last step, Synopsys PrimeTime with the previously created files as input is used to obtain the corresponding power consumption.

5.2 Performance

In the first set of experiments, we compare the performance of our processor variants (or *processor configurations*) for the three set operation algorithms and the merge-sort algorithm. The first processor variant is 108Mini, which is one of Tensilica's default processor configurations with a minimal set of features. It does not contain features (e.g., an FPU) that are not required to run the four algorithms. For both, DBA_1LSU_EIS and DBA_2LSU_EIS, there exists a variant with and without partial loading. Table 2 illustrates the core frequency and the throughput achieved by the six different configurations.

As can be seen, the core frequency of the different processor configurations is nearly the same. 108Mini achieves the highest core frequency with 442MHz. With more features integrated into the processors, the core frequency slightly decreases because the longest critical path increases. With all our features enabled, i.e, the two load–store units and our instruction set extensions (DBA_2LSU_EIS), the core frequency decreases to 410MHz. For partial loading however, we observe no decrease in the core frequency whether it is enabled or not. In general, we can conclude that our instruction set extension is well-designed because it has only a small impact on the core frequency of the processor.

Processor	Partial Load–Store	f[MHz]	Intersection	Union	Difference	Merge–Sort
108Mini	-	442	31.3	26.4	35.7	1.7
DBA_1LSU	-	435	50.7	47.7	50.4	3.2
DBA_1LSU_EIS	no	424	513.4	665.0	658.8	29.3
DBA_2LSU_EIS	no	410	693.0	643.0	637.0	28.3
DBA_1LSU_EIS	yes	424	859.0	574.2	859.0	29.3
DBA_2LSU_EIS	yes	410	1203.0	780.4	1192.6	28.3

Table 2: Maximum throughput [million elements per second]

The maximum throughput for the four algorithms under test is depicted on the right side of Table 2. We define the throughput T_{set} of any of the three set-operation algorithms as $T_{set} = \frac{l_a+l_b}{t}$ where l_a and l_b denote the number of values in the input sets a and b and t denotes the time required to perform the operation. The throughput T_{sort} of sorting is defined as $T_{sort} = \frac{n}{t}$ where n denotes the number of values being sorted and t denotes the time required for sorting. We further define the *selectivity* of a set operation as the number of results which can be minimally (0%) and maximally (100%) obtained. This is achieved by varying the input data, e.g, the intersection has 100% selectivity if both input sets contain the same elements. If not mentioned otherwise, we set the selectivity in all following experiments to 50%. The input set size for the set-operation algorithms and the sorting algorithm are 5000 and 6500 32-bit elements respectively. These values represent the maximum number of elements, which fit in the local data memories. If more values should be used, the data prefetcher is required for reloading elements. System level simulation validates a constant throughput of the processor for larger data sets due to the concurrently performed data prefetch. In this work, only processor variations and the impact of the application-specific instruction set are regarded. Consequently, the data prefetcher is not included in the evaluation.

The order of the values being sorted has no impact on the throughput of our chosen merge-sort implementation, i.e., we do not employ any shortcuts when the data is already partially sorted. The 108Mini and the DBA_1LSU run the scalar sorted-set algorithms.

The throughput of 108Mini is around 30 million elements per second for the three set operations. The union operation's throughput is typically lower than the throughput of the other operations because it produces more output tuples, which have to be written into the result set. With the attached local store (DBA_1LSU), the throughput of all three operations almost doubles because access to memory is less expensive. As soon as our instruction set extension is employed, however, the throughput increases by an order of magnitude compared to the processor configurations that provide only the standard instruction set. The use of a second load–store unit (DBA_2LSU_EIS) increases throughput even further by 35% since values of both input sets can now be read in one cycle. Finally, partial loading increases throughput up about 1,203 million elements per second for the intersection and difference operation. This

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 767–778, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2593677

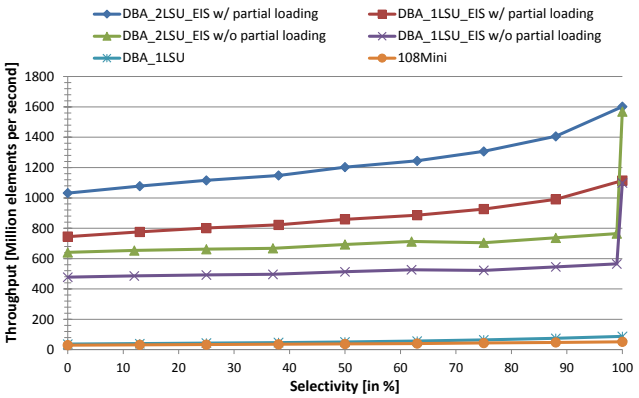

Figure 13: Selectivity of the intersection

amounts to a speedup of up to 38.4x compared to the initial processor configuration 108Mini.

We observed that the throughput of the set operation algorithms varies depending on the selectivity of both sets. If the selectivity increases, the throughput usually increases as well because the number of comparisons decreases with more common values in both sets (cf. Section 2.3). Figure 13 illustrates the throughput of the six different processor configurations for the intersection operation. As before, 108Mini and DBA_1LSU always show the lowest throughput, which slightly increases as the selectivity increases to 100%. The throughput of the other four configurations however, increases at a faster pace so that the processors employing our instruction set extension reach even higher speedups when the selectivity approaches 100%. Again, the processors with partial loading achieve a higher throughput than the processors without. Only if the selectivity reaches 100%, i.e., the values in both sets are equal, partial loading has no advantage anymore because the algorithms then proceed by 4 values in each input set in each iteration. We obtain similar results also for the other two set operation algorithms.

Finally, the throughput for sorting is depicted in the rightmost column in Table 2. Since sorting requires multiple runs over the data, the absolute throughput numbers are clearly smaller compared to the ones of the set operations. We furthermore compare only three processor configurations because partial loading as well as two load–store units are not beneficial for sorting. The 108Mini processor achieves the lowest throughput of all three configurations with only 1.3 million elements per second. Using a local store as in DBA_1LSU almost doubles the throughput since memory is accessed using a single cycle. Again, throughput increases by an order of magnitude when using our instruction set extension. DBA_1LSU_EIS is 16x and 8.5x faster than 108Mini and DBA_1LSU, respectively.

5.3 Area, Timing and Power Consumption

Within the next set of experiments, we compare the area and power consumption of the different processor configurations. The synthesis results reflect the accurate area and timings that would be used in the final processor after the tape-out. To obtain the power consumption of the processor configurations, we use the Synopsys PrimeTime estimation tool.

Technology	Processor	$A_{LOGIC}[mm^2]$	$A_{MEM}[mm^2]$	f_{MAX}[MHz]	P[mW] @f_{MAX}
65 nm	108Mini	0.220[1]	-	442[1]	27.4[1]
	DBA_1LSU	0.177	0.874	435	56.6
	DBA_2LSU	0.177	0.870	429	57.1
	DBA_1LSU_EIS	0.523	0.874	424	123.5
	DBA_2LSU_EIS	0.645	0.870	410	135.1
28 nm	DBA_2LSU_EIS	0.169	0.232	500	47.0

[1] http://www.tensilica.com/uploads/pdf/108Mini.pdf

Table 3: Synthesis results

Table 3 provides the synthesis results for the 65 nm and all five processor configurations as well as for the 28 nm technology and the DBA_2LSU_EIS processor. 108Mini occupies only 0.22mm^2, which is completely covered with logic since it has no caches or a local store on the chip. It is therefore 500x smaller than a Intel Xeon 3040 processor[3], which is a dual-core processor, similarly manufactured using 65 nm technology, and has a die size of 111mm^2. DBA_1LSU occupies less area for logic than 108Mini but is in total 5x larger because of its on-chip local-store. Adding our instruction set extension and the second load–store unit increases only the logic area, which remains smaller than the area used for the local store. The DBA_2LSU_EIS processor including all features is roughly 7x larger than 108Mini and thus still 73x smaller than the Intel Xeon 3040 processor. When switching to the 28 nm technology, the area occupied by DBA_2LSU_EIS shrinks by 3.8x and the core frequency slightly increases. The super-low power process, the super-low voltage libraries and the 0.8 V supply voltage restricts the maximum frequency but achieves significant lower power consumption.

Part	Area[%]
Basic Core	20.5
Decoding/Muxing	14.4
States	14.7
Op: All	11.3
Op: Intersection	6.8
Op: Difference	9.0
Op: Union	17.6
Op: Merge-Sort	5.7
SUM	100

Table 4: Relative area consumption per newly introduced instruction (DBA_2LSU_EIS processor)

Table 4 lists the relative area broken down to the components of the DBA_2LSU_EIS processor. The basic core and standard instruction set already amount to one third of the overall area. The decoding and multiplexing part, which is shared by all of our new instructions, occupies the second third of the chip. The circuits for the novel instruction occupy the remaining area, whereby the union operation is most expensive. It requires more wires than the other instructions for writing result values back, i.e., the instruction may write values from both input sets in one operation back whereas the other instruction write at most values from one input set back. The circuits used for the sorting instructions

[3] http://ark.intel.com/products/27203

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 767–778, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2593677

require less area than the other instruction because they do not include partial loading and use only one load–store unit.

The power consumption of the processor variants are depicted on the right-most column of Table 3. The basic core and standard instruction set consumes 57 mW. This is 2.1x more than the 108Mini processor. Especially due to the integration of the local memories and the increased processor bus widths. By applying the newly developed instruction set and a second LSU the power consumption of the DBA_1LSU is increased by 2.4x to 135 mW. When switching to the 28 nm technology, the power consumed by DBA_2LSU_EIS shrinks by 2.9x to 47 mW.

5.4 Comparison with Other Architectures

In the last set of experiments, we compare the performance of algorithms using our instruction set extension with existing highly-optimized algorithms. More specifically, we compare our merge-sort implementation (HWSORT) with the merge-sort implementation (SWSORT) of Chhugani et al. [6] as well as our sorted-set intersection implementation (HWSET) with the sorted-set intersection implementation (SWSET) of earlier work [33]. We use absolute performance numbers provided in the respective papers and compare them with results obtained in our experiments. Our implementations run always on our DBA_2LSU_EIS processor whereas the existing implementations run on modern x86 processors, which provide multiple cores. Since our processor has only a single core, we only compare the implementations' single-threaded performance. Besides the performance, we compare the features of the employed processors.

For the sorting-performance comparison, we calculate—using the formulas in Section 5.2—the throughput based on the time for sorting 6500 values in HWSORT and the time provided by Chhugani et al. [6] for sorting 512,000 values with SWSORT. As before, we cannot sort more values with HWSORT because of the small local store. We expect, however, a similar performance for sorting a larger number of values. We obtain the intersection throughput based on the time for intersecting two sets with 2500 values each in HWSET and the time for intersecting two sets with 10 million values each in SWSET. In both cases, the selectivity is set to 50%.

	Intel Q9550	DBA_2LSU_EIS
Throughput (elements/s)	60 mio	28.3 mio
Clock frequency	3.22 GHz	0.41 GHz
Max. TDP	95 W	0.135 W
Cores/Threads	4/4	1/1
Feature size	45 nm	65 nm
Area (logic & memory)	214 mm^2	1.5 mm^2

Table 5: Merge-sort comparison

Table 5 illustrates the results for the comparison of the merge-sort implementations. Chhugani et al. [6] run SWSORT in their experiments on an Intel Q9550 processor[4], which has a 8x higher core frequency but consumes up to 700x more power than our DBA_2LSU_EIS processor. Clearly, this power consumption comparison is not entirely fair, since the Intel Q9550 supports four hardware threads while our processor supports only a single hardware thread. The Intel Q9550 itself, however, has a slight advantage through its lower feature size, which typically improves energy consumption (cf. Table 3). When comparing the throughputs, SWSORT can sort roughly twice as much values in the same time compared to HWSORT. However, already small improvements on DBA_2LSU_EIS's core frequency, i.e., by reducing the feature size or integrating pipelining, would easily allow matching up throughput of SWSORT and HWSORT. Furthermore, the number of cores of DBA_2LSU_EIS could be largely increased until it occupies the same area as the Intel Q9550 processor. Even under pessimistic assumptions, DBA_2LSU_EIS could provide an order of magnitude more cores than the Intel Q9550 processor. Summing up, although HWSORT is slower than the highly-optimized SWSORT, it is much more energy-efficient and may easily outperform SWSORT when our processor is further optimized.

	Intel i7-920	DBA_2LSU_EIS
Throughput (elements/s)	1,100 mio	1,203 mio
Clock frequency	2.67 GHz	0.41 GHz
Max. TDP	130 W	0.135 W
Cores/Threads	4/8	1/1
Feature size	45 nm	65 nm
Area (logic & memory)	263 mm^2	1.5 mm^2

Table 6: Sorted-set intersection comparison

The results for comparing HWSET and SWSET are depicted in Table 6. In our earlier work [33], we used an Intel i7-920 processor[5] within our experimental evaluation; it has a 6.5x higher core frequency and consumes up to 960x more power than DBA_2LSU_EIS but can run 8 threads in parallel with Hyperthreading enabled. As before, Intel i7-920's per thread power consumption is lower but it is manufactured using 45 nm feature size, which forms an advantage over DBA_2LSU_EIS. The throughput of HWSET is 9.4% higher than the SWSET's throughput despite of the much lower core frequency and power consumption. Again, we expect even a higher throughput when our processor is further optimized.

6. FURTHER RELATED WORK

In this section, we provide additional related work that is not already covered within the paper; it can be divided into three parts covering (1) existing instruction set extensions and database algorithms that exploit them, (2) a brief overview about early attempts of database processors, and (3) novel tailor-made FPGA circuits and ASICs for processing database tasks.

Hardware vendors like Intel, AMD, or IBM started early to enhance their processors with additional instruction sets to speed up algorithms of various applications. Starting with floating-point instructions running in co-processors, instruction set extensions for multimedia processing, encryption, video encoding, and string processing were integrated into processors. For modern general-purpose processors from Intel, for example, there exist MMX, various revisions of SSE and AVX, SSSE3, Quick Sync [15], AES, VT-x, and FMA3.[6] Although none of the instruction sets available in any general-purpose processor is primarily developed to support database operations, many of them can still be utilized to speed up such operations. Typically, the SIMD instruction sets have been proved to be helpful for this purpose. Examples are vectorized versions of compression algorithms

[4] `http://ark.intel.com/products/33924`

[5] `http://ark.intel.com/products/37147`

[6] Details to these instruction sets can be found in Intel's optimization reference manual [16].

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 767–778, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2593677

[26, 36], database operators [38, 4], tree-based and hash-based search algorithms [18, 32], and sorting algorithms [6, 12]. However—as already known and again shown in this paper—even highly-optimized algorithms that use general-purpose instruction sets cannot match the energy efficiency and therefore performance of algorithms that use application-specific instruction sets. Since instruction sets specifically designed for database operations will most likely not appear in general-purpose processors, our paper provides the necessary background to develop and build such instruction sets efficiently. The aforementioned vectorized algorithms might be a good starting point for this development.

In the 1980s, database machines were developed to speed up query processing by using application-specific hardware. Early approaches put processing logic directly into the write–read heads of magnetic disks. Simple tasks like tuple parsing, searching and selecting tuples could therefore be performed while reading from the disks' tracks. Boral et al. [5] provide an excellent overview of such *disk-head processing* architectures. Later database machines, e.g., GRACE [21], the Super Database Computer [20], or the GAMMA dataflow machine [8], are mostly shared-nothing architectures where multiple nodes, which consist of a processor, a disk cache, and disks, are connected via a network. Query processing is therefore performed in a highly parallel fashion. The nodes itself are often enhanced using specialized circuits, e.g., hardware sorter [22]. The CAFS system [3], the LEECH processor [28], the pipelined relational query processor [19], and the relational database machine [2] are even processors that are solely designed for database query processing. They support complex operations, e.g., join, aggregation, and sorting, and provide a much higher query processing performance than general-purpose processors having a similar core frequency. Although all these processors and systems aim at speeding up query processing using specialized hardware, none of the existing works has an answer on how to build processors efficiently. Employing customizeable processors—as proposed in our paper—eases the processor development and decreases its costs and thus first enables building database processors.

Based on the disk-head processing architectures, recently tailor-made circuits intended to be run on FPGAs were developed to speed up mostly stateless data processing tasks. For example, Mueller et al. [29, 30] proposed circuits for various data streaming operators, e.g., selection, projection, and windowed aggregation and median; Koch et al. [24] discussed FPGA sorting approaches; and Teubner et al. [35] proposed an FPGA stream join. Besides academic research, IBM Netezza [11] uses FPGAs in the data path from disks to host processors for filtering and transforming tuples loaded from disk. More or less all these approaches rely on the reconfigurability of the FPGAs, e.g., the circuit running on the FPGA is changed for each streaming query [29, 30, 11]. The reconfigurability, however, comes at the price of a lower density, higher power consumption, and lower core frequency compared to application-specific integrated circuits (ASICs) where the circuits cannot be changed. Kuon [25] reported that in average an FPGA is 40x larger, consumes 12x more dynamic power, and is 3.2x slower than an ASIC. Since most database processing tasks (e.g., sorting, aggregating, or joins) are relatively fixed and well-defined, the reconfigurability of an FPGA is not required. Hence, specialized instruction sets as proposed in our paper should be superior to FPGA circuits in terms of energy consumption and performance when processing the same tasks.

Lately, several authors proposed specialized instruction sets for database systems similar to our instruction set extension. Wu et al. [37] proposed the HARP accelerator for speeding up partitioning of numerical values, which is heavily used in database systems. The key idea of the approach is to build specialized *streaming cores* that solely partition numeric sequences besides the regular cores used for processing. Stream buffer instructions are used to initialize the streaming cores and feed them with values that should be partitioned. Similarly, Kim et al. [27] built specialized cores for processing hash joins. These *walker units* are solely used to traverse the collision lists of hash tables to find whether a hashed key occurs in them or not. Hayes et al. [14] proposed a vector instruction set with 28 instructions to speed up hash joins in columnar database systems. Besides the different application areas, all three authors have no answer how to eventually build a processor supporting their instructions. They use different simulators to evaluate them and discuss their integration into x86 processors, however, only our work considers all steps from developing an instruction set extension to its final integration into a processor.

Lastly, Iyer [17] proposed two instructions to speed up tournament-tree sorting used within DB2. Basically, certain often reused instruction sequences in this sort algorithm are merged to obtain the two instructions. Although the instructions are integrated into IBM's z-series processors, Iyer neglects the question how to ease the processor development. Only our paper provides a viable solution for this issue.

7. CONCLUSION

Database systems should provide a high query throughput and low query latency. Consequently, one of the key tasks in the development of database systems is a high query performance. To achieve this, they are optimized manifoldly: efficient data structures or in-memory data layouts, cache-optimized and parallel algorithms, and even data processing circuits for FPGAs are developed with the goal of processing queries faster. Our goal in this paper is to show how to improve the query performance using database-specific instruction set extensions. We discuss the necessary background to obtain such instructions, proposed the model of a database-centric processor, and provide exemplarily instructions for set-oriented database primitives. In our experiments, we show that algorithms that are run on our processor and exploit the presented instruction set achieve a similar performance than highly-optimized algorithms on modern x86 processors but require much less energy at the same time. Since general-purpose processors will not scale because of thermal and energy constraints, the development of application-specific processors is inevitable. Clearly, many more instruction sets need to be developed to run entire database systems efficiently on such processors. Our paper provides the foundation for building such instruction sets.

8. ACKNOWLEDGEMENTS

This work has been supported by the state of Saxony under grant of ESF 100098198 (IMData) and 100111037 (SREX) and the German Research Foundation within cfaed and the grant LE 1416/22-1 (HAEC). Furthermore, we would like to thank Synopsys and Tensilica for software and IP.

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 767–778, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2593677

9. REFERENCES

[1] ARC Inc. *750D Core Architecture Manual*, 2005.

[2] H. Auer, W. Hell, H.-O. Leilich, J. Lie, H. Schweppe, S. Seehusen, G. Stiege, W. Teich, and H. C. Zeidler. Rdbm-a relational data base machine. *Information Systems*, 6(2), 1981.

[3] E. Babb. Implementing a relational database by means of specialzed hardware. *ACM Transactions on Database Systems*, 4(1), 1979.

[4] 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.

[5] H. Boral and D. J. DeWitt. *Database machines: An idea whose time has passed? a critique of the future of database machines*. Springer, 1983.

[6] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proc. VLDB Endow.*, 2008.

[7] CoWare Inc. *LISA Language Reference Manual*, 2011.

[8] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *Knowledge and Data Engineering*, 1990.

[9] B. Ding and A. C. König. Fast set intersection in memory. *Proc. VLDB Endow.*, 4(4), Jan. 2011.

[10] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[11] P. Francisco. *The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics*. IBM Corp., 2011.

[12] B. Gedik, R. R. Bordawekar, and P. S. Yu. Cellsort: high performance sorting on the cell processor. In *VLDB*, 2007.

[13] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *Micro*, 2011.

[14] T. Hayes, O. Palomar, O. Unsal, A. Cristal, and M. Valero. Vector extensions for decision support dbms acceleration. In *MICRO*, 2012.

[15] Intel corp. *Technology Insight: Intel Next Generation Microarchitecture Codename Ivy Bridge*, 2011.

[16] Intel Corp. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, July 2013.

[17] B. R. Iyer. Hardware assisted sorting in IBM's DB2 DBMS. In *COMAD*, 2005.

[18] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, 2010.

[19] W. Kim, D. Gajski, and D. J. Kuck. A parallel pipelined relational query processor. *ACM Transactions on Database Systems*, 9(2), June 1984.

[20] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *VLDB*, 1990.

[21] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1), 1983.

[22] M. Kitsuregawa, W. Yang, T. Suzuki, and M. Takagi. Design and implementation of high speed pipeline merge sorter with run length tuning mechanism. In *Database Machines and Knowledge Base Machines*. Springer, 1988.

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

[24] D. Koch and J. Torresen. Fpgasort: a high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *ACM/SIGDA*, FPGA '11, 2011.

[25] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. *Computer-Aided Design of Integrated Circuits and Systems*, 26(2), 2007.

[26] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 2013.

[27] K. Lim, B. Falsafi, J. Picorel, B. Grot, P. Ranganathan, O. Kocberber, et al. Meet the walkers: Accelerating index traversals for in-memory databases. In *MICRO*, 2013.

[28] D. R. McGregor, R. G. Thomson, and W. N. Dawson. High performance hardware for database systems. In *VLDB*, 1976.

[29] R. Mueller, J. Teubner, and G. Alonso. Data processing on fpgas. *Proc. VLDB Endow.*, 2(1), Aug. 2009.

[30] R. Mueller, J. Teubner, and G. Alonso. Streams on wires: a query compiler for fpgas. *PVLDB*, Aug. 2009.

[31] V. Raman, L. Qiao, W. Han, I. Narang, Y.-L. Chen, K.-H. Yang, and F.-L. Ling. Lazy, adaptive rid-list intersection, and its application to index anding. In *SIGMOD*, 2007.

[32] K. A. Ross. Efficient hash probes on modern processors. In *ICDE*, 2007.

[33] B. Schlegel, T. Willhalm, and W. Lehner. Fast sorted-set intersection using simd instructions. In *ADMS*, 2011.

[34] Tensilica Inc. *Tensilica Instruction Extension (TIE) Language Reference Manual*, 2013.

[35] J. Teubner and R. Mueller. How soccer players would do stream joins. In *SIGMOD*, 2011.

[36] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2009.

[37] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *ISCA*, 2013.

[38] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *SIGMOD*, 2002.