Sebastian Haas, Tomas Karnagel, Oliver Arnold, Erik Laux, Benjamin Schlegel, Gerhard Fettweis, Wolfgang Lehner

HW/SW-database-codesign for compressed bitmap index processing

Erstveröffentlichung in / First published in:

International Conference on Application Specific Systems (ASAP), Architectures and Processors. London, 06.-08.07.2016. IEEE, S. 50-57. ISBN 978-1-5090-1503-0.

DOI: http://dx.doi.org/10.1109/ASAP.2016.7760772

Final edited form was published in "International Conference on Application Specific Systems (ASAP), Architectures and Processors. London 2016", S. 50-57,
ISBN 978-1-5090-1503-0
http://dx.doi.org/10.1109/ASAP.2016.7760772

HW/SW-Database-CoDesign for Compressed Bitmap Index Processing

Sebastian Haas*, Tomas Karnagel†, Oliver Arnold*,
Erik Laux*[1], Benjamin Schlegel†[2], Gerhard Fettweis*, Wolfgang Lehner†
*Vodafone Chair Mobile Communications Systems, †Database Technology Group
Center for Advancing Electronics Dresden (cfaed)
Technische Universität Dresden, Germany
Email: {first.last}@tu-dresden.de

***Abstract*—Compressed bitmap indices are heavily used in scientific and commercial database systems because they largely improve query performance for various workloads. Early research focused on finding tailor-made index compression schemes that are amenable for modern processors. Improving performance further typically comes at the expense of a lower compression rate, which is in many applications not acceptable because of memory limitations. Alternatively, tailor-made hardware allows to achieve a performance that can only hardly be reached with software running on general-purpose CPUs. In this paper, we will show how to create a custom instruction set framework for compressed bitmap processing that is generic enough to implement most of the major compressed bitmap indices. For evaluation, we implemented WAH, PLWAH, and COMPAX operations using our framework and compared the resulting implementation to multiple state-of-the-art processors. We show that the custom-made bitmap processor achieves speedups of up to one order of magnitude by also using two orders of magnitude less energy compared to a modern energy-efficient Intel processor. Finally, we discuss how to embed our processor with database-specific instruction sets into database system environments.**

I. Introduction

Compressed bitmap indices are heavily used in scientific and commercial database systems because they largely improve query performance for various workloads. Application areas include traditional data warehouse applications as well as multimedia and scientific processing. Most of the applications operate on huge amounts of data, which makes it imperative to store the indices as compactly as possible but at the same time without affecting the index performance too much.

Early research focused on developing index compression schemes that are optimized for general-purpose processors. For example, several authors proposed to use byte-aligned [1] and word-aligned [2], [3] compression schemes instead of generic bitwise compression algorithms. After the compression, index operations like intersection can be done directly on the compressed bitmaps to save memory and computation time. However, even with elaborate optimizations, e.g., the use of SIMD instructions, all existing algorithms are highly CPU bound. Improving performance further thus typically comes at the expense of a lower compression rate, which is not acceptable for many applications because of memory limitations especially in main memory settings. More recent works therefore try to port existing algorithms to GPUs [4] to benefit from their high computation capabilities. This seems to be a good fit since the algorithms are mostly stateless and easy to parallelize. The next logical step is therefore to build hardware that is solely used to speed up processing on compressed bitmap indices.

Generally, 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 [5]. Besides a better performance, such hardware is much more energy-efficient, which is a feature that becomes more and more important for future processors. The main disadvantages of custom hardware are the high development costs that come with designing and verifying a new CPU, as well as building respective drivers and software stack. However, there is actually no need to build a full processor from scratch. Recent work in the field of databases [6] and video coding [7] has shown that these costs are manageable when using customizable CPUs. Two areas, besides others, would mostly benefit from these customized CPUs: high end solutions, where customers are willing to pay for the enhanced performance, and cloud settings, where the hardware can be exclusively used for database acceleration while being highly energy-efficient.

In this paper, we will show how to create a custom instruction set framework for compressed bitmap processing that is generic enough to implement most of the major compressed bitmap indices. We exemplarily show the integration of the developed instruction set within a customizable processor. Nevertheless, the instruction set is generic enough to be integrated into future instruction sets of various hardware vendors as well as to be part of open-hardware platforms like OpenSparc.

Our contributions are as follows:

- We review two types of bitmap index compressions and explain both with a number of examples.
- We describe the development of instruction set extensions together with a processor architecture that supports these adjustments.
- We propose a hardware framework for the easy implementation of instruction set extensions for stateless compressed bitmap processing.
- We evaluate our framework using three different compressions each with three different processing operations.

[1]Author is now at BearingPoint GmbH, Berlin, Germany.
[2]Author is now at Oracle Labs, Belmont, CA, USA.

Final edited form was published in "International Conference on Application Specific Systems (ASAP), Architectures and Processors. London 2016", S. 50-57,
ISBN 978-1-5090-1503-0
http://dx.doi.org/10.1109/ASAP.2016.7760772

- Finally, we give an outlook on future work, especially on settings in which our ideas could be integrated in larger hardware and database systems.

II. Processing Compressed Bitmap Indexes

Bitmap indexes are well known and widely used in database systems, mainly to accelerate OLAP queries on large relational data. There, for example, bitmaps can be used for the fast evaluation and intersection of tuples satisfying query arguments. Usually columns with a limited number of distinct values are encoded in a number of bitmaps, where each bitmap is dedicated to one value or a range of values within a column. A set bit then signals a match between the corresponding row and the attribute.

Bitmaps themselves compress the tuples immensely by encoding only one bit per tuple. However, the number of bitmaps to represent a column depends on the number of distinct values, i.e., having many distinct values leads to storing many bitmaps, which as a consequence could lead to a significant memory overhead. Fortunately, when having many bitmaps for a single column, the individual bitmaps are only sparsely filled leading to a high compression potential.

In database systems, sparsely filled bitmaps are traditionally compressed using run-length-encoding (RLE) [8]. However, over the last 20 years, many algorithms were proposed to, on the one hand, compress bitmaps as good as possible while, on the other hand, supporting basic operations directly on the compressed representation. Ideally, bitmaps can be intersected without being decompressed beforehand.

A. Compression Approaches

When looking at bitmap compression algorithms, we identified two different approaches: (1) Algorithms using signal words to describe the encoding of the following words, and (2) Algorithms using stateless compression words, with encoding information and the actual data encoded in the same word. The first approach uses special signal words, usually to state the number of following uncompressed words (literals), RLE compressed words (fills), or bit positions for single set bits (sparse). Well known examples for this approach are Byte-Aligned Bitmap Code (BBC) [1] and Enhanced Word-Aligned Hybrid (EWAH) [9].

The second approach omits these signal words by specifying different words (e.g., fills or literals) within the word itself. Each word encodes an identifier, leaving the different words independent of each other. Therefore the algorithms do not have to know a state when extracting encoded words. Examples are Word-Aligned Hybrid (WAH) [10], Position List Word Aligned Hybrid (PLWAH) [2], and COMPAX [3].

In this work, we accelerate logical bitmap operations through the usage of application-specific processor instructions. In order to provide a general overview, we investigated the characteristics of both approaches. Stateless compression on the one hand always produces an overhead through encoding word types and other information within the code word, while approaches with signal words can minimize this overhead down to only one signal word, for certain kinds of data. However, compression, decompression, and logical operations are usually more complex with signal words, forcing the algorithm to access signal words multiple times, which may lead to more random memory lookups for read operations, as well as to a larger number of random memory writes for write operations. Stateless approaches on the other hand, only access the current word and always read or write sequentially. Especially, the latter point makes it ideal for acceleration in hardware, where a streaming-like approach is preferred. Therefore, we limit the scope of this work to stateless compression approaches.

B. Stateless Bitmap Compression

In the following subsection, we describe three stateless bitmap compression algorithms that we chose as an example to be implemented within our hardware instruction framework.

1) Word-Aligned Hybrid (WAH): The Word-Aligned Hybrid code (WAH) [10] was developed to reduce computational complexity compared to compression algorithms using signal words. The WAH compressed bitmaps can contain RLE compressed fills and uncompressed literals but instead of using signal words, a 1-bit flag is used to differentiate the two types of words. In this context, a word usually has the size of 32 bits or 64 bits. For simplicity, we limit ourselves to 32-bit words.

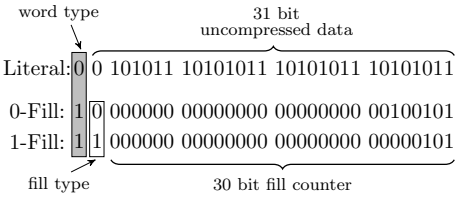


Fig. 1: WAH Code

Three example words are illustrated in Figure 1. A literal is encoded with the most significant bit (MSB) set to zero (or unset bit) and 31 bits of uncompressed data. A fill is encoded with the MSB set to one, a second bit indicates if the fill is representing set or unset bits, and 30 bits of payload indicate the number of represented chunks. For WAH, chunks have the size of 31 bits. With a fill counter of 30 bits, a single fill can represent up to 2^{30} *chunks* $*$ 31 *bits per chunk* $=$ 3.875 GB, if the bitmap contains solely set or unset bits. In the opposite case, if the uncompressed chunks always contain a mixture of set and unset bits, the whole bitmap is encoded as literals leading to an overhead in size of $1/32 = 3.125\%$. The actual compression ratio (size of compressed bitmap related to size of uncompressed bitmap) depends on the bit density and distribution of the uncompressed data. However, the worst case is only slightly larger than the original data, while the best case encodes about 3.9 GB of data in just 4 bytes.

Logical operations on compressed bitmaps include AND, OR, XOR, and many more. These operations are simple for uncompressed bitmaps, however, the execution is highly memory bound. With compressed bitmaps, the operations are shifting from being memory bound to being computation bound, depending on the complexity of the bitmap encoding.

For the intersection of two WAH compressed bitmaps, the following steps are performed in a loop, as long as there are words left in both bitmaps: 1) loading next word(s) if needed, 2) calculating the output, and 3) combining the output

Final edited form was published in "International Conference on Application Specific Systems (ASAP), Architectures and Processors. London 2016", S. 50-57,
ISBN 978-1-5090-1503-0
http://dx.doi.org/10.1109/ASAP.2016.7760772

if needed. A new word is loaded from a bitmap, if the last word was completely consumed by the previous calculation. The two current words, one of each bitmap, are used in the subsequent calculation step. There are three types of calculations: fill-fill, literal-fill, and literal-literal. Literals are always consumed and a new word has to be loaded after the calculations. For fills, the fill counter is updated and if zero, a new word needs to be loaded. Each calculation step produces one WAH encoded output word. The words of multiple calculation steps might be combined, if they are fills of the same kind.

2) Position List Word Aligned Hybrid (PLWAH): WAH was extended by several approaches. One of these approaches is PLWAH [2]. Here, the main assumption is that a fill is mostly followed by a literal and that this literal is most likely sparse. Especially, for a bitmap index on a column with many distinct values, the single bitmap is only sparsely filled. The idea of PLWAH (which is also used in the similar compression CONCISE [11]) is to encode the following literals within the fill. For WAH, 30 bits were used for the fill counter. Small counter values exhibit several unused bits. This allows recycling these bits to encode the following literal. In the case of PLWAH, five bits are used to encode the position of one divergent bit in the following literal. The divergent bit for a 1-fill is an unset bit and vice versa.

3) COMPressed Adaptive indeX (COMPAX): COMPAX [3] extends the PLWAH approach by encoding multiple fills or literals into one 32-bit code word. Likewise in WAH, a literal is encoded with a 1-bit type flag and 31 bits of uncompressed data. Additionally, there are three other code words, each having the type encoded in the most significant three bits. 0-fills contain three unset bits as type description and 29 bits for the counter. A 1-fill type is not supported. The reason leads back to the main use case of COMPAX - encoding of network traffic. The authors analyzed typical networking data and conclude that 1-fills nearly never occur. For the application in databases, the conclusion could be similar when working on a large number of distinct values for a column and therefore many sparsely filled bitmaps. Besides literals and 0-fills, the authors propose two hybrid words: Fill-Literal-Fill and Literal-Fill-Literal. We refer to the literature [3] for more information.

III. Hardware Foundation

After outlining the core principles of stateless bitmap compression schemes, we give an overview of our processor environment and the involved tool flow in order to provide a solid background for the instruction set framework.

A. Extending the Core Processor

We rely on a Tensilica LX5 RISC processor as a starting point for our proposed framework. The LX5 core basically consists of a RISC instruction set as well as register files. A 64 KB local instruction memory and two 32 KB local data memories are connected, respectively; both memory interfaces exhibit a width of 32 bit.

In order to develop an instruction set extension, a complex process has to be implemented incorporating different tools from Tensilica or Synopsys depending on the state of the development. On the one hand, the processor may be extended

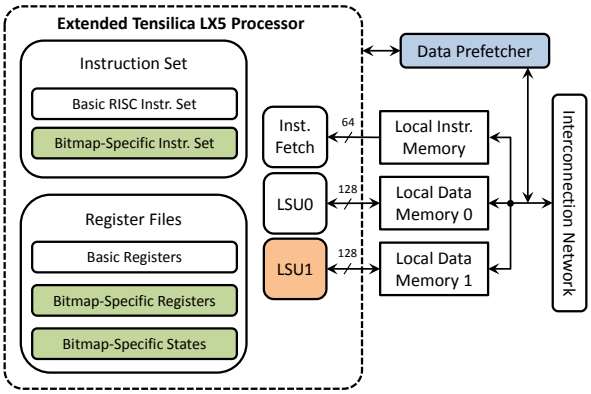


Fig. 2: Extended Processor Model

by changing the local memory size, memory interface widths, adding additional load-store units, and other configurations like additional floating point units and extended memory management units. On the other hand, Tensilica Configurable Processors allow the development of new instructions exploiting the different processor units.

For identifying the potential of additional instructions, an algorithm is being profiled using a cycle accurate simulator in order to detect hotspots in the original C/C++ code. The simulator enables simple debugging and maintenance in a software environment. Within a subsequent step, synthesis of the generated processor on register-transfer level is performed to obtain results of timing, area, and power. While developing the framework for different bitmap algorithms, we synthesized all processor variants with 65 nm low-power TSMC libraries under typical conditions (25 °C, 1.2 V).

B. The Extended Processor

Following the tool flow, we extended the Tensilica LX5 RISC processor as shown in Figure 2 allowing us to integrate the extended instructions by providing additional registers and states (states are bound to our special bitmap instructions and allow read and write accesses in the same clock cycle). The extended processor exhibits a second load-store unit (LSU) enabling simultaneous accesses to two local data memories. Moreover, these two memory interfaces are enlarged from 32 bit to 128 bit for an even faster memory access. Additionally, the local instruction bus is extended from 32 bit to 64 bit, enabling us to introduce a new 64-bit *Very Large Instruction Word* (VLIW) format. This instruction format ensures the control of our two load-store units by executing multiple instructions within a single clock cycle.

Figure 2 also outlines a *Data Prefetcher* operating next to the processor allowing us to preload data from an external off-chip memory into the local memories over a Network-on-Chip. With respect to the given experiments in Section V, the prefetcher unit was not yet available, but will be part of our future work.

In addition to unit extensions, the final processor also incorporates a bitmap-specific instruction set. It represents the extension to the basic instruction set (transfer, arithmetic, logical, and jump instructions), which is included in every processor to perform normal program execution. In contrast

Final edited form was published in "International Conference on Application Specific Systems (ASAP), Architectures and Processors. London 2016", S. 50-57,
ISBN 978-1-5090-1503-0
http://dx.doi.org/10.1109/ASAP.2016.7760772

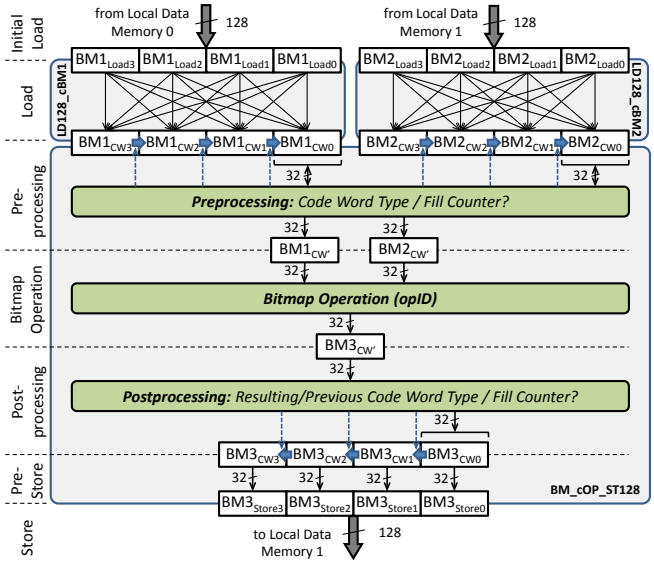


Fig. 3: Framework for Bitmap Processing Algorithms

to other processor families that use fixed instruction set extensions, Tensilica additionally provides an environment to develop application-specific instructions using the Tensilica-specific Hardware Description Language (HDL), which can then be used like any other conventional assembler macro. Due to the real hardware representation, the instructions can be executed within a single clock cycle, or–if required–scheduled over multiple cycles to improve the processor's clock frequency.

IV. Instruction Framework

As mentioned, the novel idea of our approach is not only to deploy HW/SW-CoDesign for database primitives as instruction set extensions, but to provide a framework allowing the implementation of a variety of state-of-the-art stateless bitmap algorithms. In the following, we outline the building blocks of an instruction set extension framework in order to perform logical operations on compressed bitmaps. We demonstrate the feasibility of the framework by using the three bitmap encodings revisited in Section II.

A. Detailed View of the Framework

Figure 3 provides a detailed view of the overall framework for logical operations on compressed bitmaps. It also provides the complete set of instructions and their relationship. The instruction set extension consists of two load instructions *LD128_cBM1* and *LD128_cBM2* for both input bitmaps and a combined process/store operation named *BM_cOP_ST128*, with *OP* denoting the free parameter of the framework having an impact on the three processing phases of Preprocessing, Bitmap Operation, and Postprocessing. Merging the process and store operations into a single instruction and adding an internal pipeline structure using 4 cycles for the actual processing turned out to be the optimum between execution time and latency. Diving into more detail, the following list describes the different phases of the framework and the associated pipeline steps necessary to reduce the longest combinatorial path in the processor.

Initial Load: Within the initial load pipeline stage, four 32 bit words of the compressed input bitmap stream 1 and bitmap stream 2 are loaded from the local data memory 0 and from local data memory 1, respectively.

Load: The memory access is aligned to 128-bit lines. Hence, after loading the four 32 bit words from memory, the data has to be reordered to ensure a continuous availability.

Preprocessing: In this cycle the first 32 bit word of each bitmap stream is considered and a comparator checks whether the words are literals, fills, or other compression specific code words. Hence, it is decided if the input words are overwritten with a changed fill, or the 128 bit input stream is shifted to preserve the code word. The arrows within the input registers of this pipeline stage in Figure 3 illustrate this dependency. Obviously, this step is already depending on the specific compression scheme.

Bitmap Operation: This stage performs the actual bitwise operation (defined by *opID*) of the two prepared code words.

Postprocessing: The resulting code word from the previous cycle is written to the output bitmap stream. Again, depending on the code word type, it has to be distinguished among overwriting the previous word, or appending the result to the output bitmap stream.

Prepare Store: Since we have a 128 bit memory interface, we have to buffer multiple 32 bit words to reduce accesses and the longest combinatorial path (critical path) to the local memory. A 128 bit register holds the fully processed code words.

Store: The final pipeline stage uses the wide memory interface to store four 32 bit compressed code words simultaneously. The structure is designed in order to fully exploit the capacity of the processor's memory interfaces by loading and storing 128 bit of data in every clock cycle.

In order to use the framework, the instruction-specific registers and states have to be initialized and set to the addresses of the two input bitmaps as well as to the address of the output bitmap. Additionally, the specific compression scheme as well as the logical operation *opID* have to be specified. The initialization goes hand in hand with a preliminary execution of both load instructions and it also signals the start of the pipelined code implementing the main body of the operation. Since this main loop reflects the most time consuming part and takes more than 95% of the total execution time, the number of cycles per loop iteration should be reduced as much as possible. For this reason, the process and store operations are merged into one instruction. The resulting latency increase by one clock cycle is negligible in terms of the overall performance. The last phase contains further calls of the *BM_cOP_ST128* instruction at the end of the algorithm to finish the pipeline. The following code snippet illustrates the use of our framework within a C environment:

```
do {
    LD128_cBM1(); LD128_cBM2();
    BM_cOP_ST128();
    BM_cOP_ST128();
    BM_cOP_ST128();
} while(BM_cOP_ST128());
```

Final edited form was published in "International Conference on Application Specific Systems (ASAP), Architectures and Processors. London 2016", S. 50-57,
ISBN 978-1-5090-1503-0
http://dx.doi.org/10.1109/ASAP.2016.7760772

While the load instructions can be used for all implementations of our framework, the *BM_cOP_ST128* instruction needs to be adapted to the specific bitmap encoding and processing operation.

Within the first line, we combine the two load instructions to activate the two load-store units at the same time and trigger a concurrent load of eight 32 bit code words within one single clock cycle. Thereafter, the main instruction *BM_cOP_ST128* is called four times representing the four corresponding pipeline stages. If the end of a bitmap is reached, a zero is returned and the 4th call of *BM_cOP_ST128* aborts the loop.

The complete `do-while` loop takes 6 clock cycles to finish one iteration; 5 cycles are consumed by the bitmap-specific instructions (1x load plus 4x process/store) and 1 cycle by checking the condition of the while loop. However, this extra cycle can be avoided by partially unrolling the `do-while` loop. For this purpose, a `for`-loop is placed in front of the `do-while` loop and compiling the final code using at least the GCC optimization level $-O2$. The `for`-loop would cover the minimal number of operations, e.g., when every word is entirely consumed in each step, and the following `do-while` loop would catch cases, where the minimal number of operations is not sufficient.

B. Exploiting the Framework

In order to demonstrate the feasibility of the introduced compressed bitmap processing framework, we outline the implementation of the logical AND, OR, and XOR operators based on different compression schemes. To show the versatility of our approach, we refer to the bitmap compression schemes revisited in Section II, i.e., WAH, PLWAH, and COMPAX by adapting the Preprocessing, Bitmap Operation, and Postprocessing step of the proposed framework.

Word-Aligned Hybrid (WAH): For the WAH compression scheme, the Preprocessing step includes identifying the code word types and modifying the fill counters. Here, we only need to consider fills and literal. The first code word of each input bitmap register is passed to the Bitmap Operation stage. If multiple chunks of only zeros or ones are compressed in a fill, the individual chunks are passed to the next stage successively; the fill counter determines if the remaining code words located in the input registers are shifted to the front. Within the Bitmap Operation stage, the specific operator (identified by the parameter *opID*) is applied, i.e., a real bitwise operation is only necessary for two literals. The Postprocessing step finally determines the resulting fill counter and, if possible, merges consecutive code words. In contrast to the pure software solution, our memory accesses are parallelized to store four code words simultaneously.

Position List Word-Aligned Hybrid (PLWAH): Compared to WAH, PLWAH code words exhibit sparse bits indicating an additional literal after the specified number of fills. This requires that shifting the code words in the input bitmap has to be delayed by one clock cycle, since an extra cycle is used to evaluate the sparse bit separately. The Bitmap Operation step of the PLWAH implementation is identical to the WAH approach. The Postprocessing again has to deal with sparse bits when concatenating the resulting code words, so additional checks are needed to identify sparse bits and add them to a possible fill word.

COMPressed Adaptive indeX (COMPAX): Besides 0-fills and literals used in WAH, COMPAX compressed bitmaps may contain combinations of them, requiring a modified shifting of the code words in the input registers because only one section of a code word is passed to the Bitmap Operation stage, if a hybrid code word is encountered. Therefore, the Preprocessing step extracts the fills and literals from the Fill-Literal-Fill and Literal-Fill-Literal code words, respectively, and passes the sections individually to the next step. Again, the Bitmap Operation step is identical to the WAH approach. Postprocessing has to consider possibilities to concatenate the code words to hybrid words.

V. Evaluation

After presenting our bitmap instruction set extension framework as well as multiple specific implementations, we evaluate our approach by outlining the final processor including the additional instructions. We evaluate the final processor with respect to performance and power consumption and compare the results to the base processor as well as to two state-of-the-art x86 processors.

A. Evaluation Setup

In the following, we present the test data as well as our new processor called `BitiX` (BITmap Instruction eXtension), together with the three different processors we want to compare it with.

Test Data: As test data, we use bitmaps containing $N = 50,000$ bits. The number of bits is limited by the size of the local data memories of our Tensilica processors. In the future, we want to remove this limitation with an intelligent data prefetcher. All implemented algorithms show a linear complexity $\mathcal{O}(N)$ and scale with the size of the bitmaps. We therefore fix the size for the remainder of this paper to $N = 50,000$ bits. However, more significant for our experimental analysis is the distribution of set bits within the input data, described by the bit density D_{Bit}. A bit density of $D_{Bit} = 1$ corresponds to a bitmap containing only set bits, $D_{Bit} = 1/2$ means every second bit is set, for $D_{Bit} = 1/4$ every forth bit, etc. We further assume a uniform distribution of the bits. The final compression ratio depends on the bit density as well as the efficiency of the chosen algorithm.

The BitiX Processor: Our final processor design, `BitiX`, combines all three framework implementations for the processing of different compression algorithms, each with the operation AND, OR, and XOR. On the one hand, the entire logic area of the `BitiX` processor comprises 0.443 mm^2 and increases by factor of 2.5x compared to the base processor. Almost 20% of the additional area is occupied by additional 128 bit registers. This fact is justified because the new registers allow SIMD processing and are necessary to implement the pipelining within the instructions. On the other hand, the total chip area increases only slightly because the most significant area of the chip, the memory area, remains constant at 0.874 mm^2. Our final processor performs with a maximum frequency of 384 MHz.

Final edited form was published in "International Conference on Application Specific Systems (ASAP), Architectures and Processors. London 2016", S. 50-57,
ISBN 978-1-5090-1503-0
http://dx.doi.org/10.1109/ASAP.2016.7760772

Reference Processors: In order to evaluate a software version of the bitmap encodings and operations, we compare it with three other processors. First, we use the basic LX5 Xtensa processor (`RISC`) with no instruction set extensions, one load-store unit, and 32-bit memory interfaces. The comparison with this processor shows the direct improvements achieved by new instructions of `BitiX`. This processor is also synthesized in a 65 nm process and reaches 555 MHz. The difference in frequency between `RISC` and `BitiX` results in longer critical paths of `BitiX`, forcing the overall frequency to go down. Both, `RISC` as well as `BitiX` are single core processors.

As a second processor, we choose a powerful Intel i7 extreme-edition (`Intel i7-3960X`) based on the Sandy-Bridge architecture with 6 cores, 15 MB L3 cache, and a maximum turbo frequency of 3.9 GHz. The processor is manufactured using a 32 nm process.

Third, we use a highly energy-efficient mobile processor (`Intel M-5Y10`) based on the Broadwell architecture with 2 cores, a 4 MB cache, a maximum frequency of 2 GHz, and a manufacturing process of 14 nm. The `Intel M-5Y10` exhibits a significantly lower performance than the `Intel i7-3960X` but consumes about 29x less power (according to *Thermal Design Power* (TDP)). For fair comparisons, we located all data within the L3 cache and perform all measurements using only a single core. The Intel processors execute the same code as the `RISC` core.

As a summary, we compare all four processors in Table I. The total area A_{total} indicates the sum of the entire logic area and the memory or cache area for all processors. As can be seen, `RISC` and `BitiX` are much smaller than the Intel multi-core processors, even with a larger manufacturing process. Considering only the area of Intel's single cores, the two Xtensa processors are still significantly smaller.

Processor	Tech. $[nm]$	A_{total} $[mm^2]$	f_{max} $[GHz]$	$P_{max}[W]$ $@f_{max}$
`BitiX`	65	1.32	0.384	0.089
`RISC`	65	1.05	0.555	0.055
`Intel i7-3960X`	32	435	3.9	32.2
`Intel M-5Y10`	14	82	2.0	3.0

TABLE I: Configuration of our `BitiX` processor and three processors for comparison

B. Performance

For the Intel processors, we measure the wall-clock execution time (`gettimeofday()`). For the `RISC` and the `BitiX` processors, we obtain the cycle count of cycle accurate simulations and divide that by the achieved frequency to get the actual runtime.

In Figure 4, we show the AND operation for WAH, PLWAH, and COMPAX on `RISC` and `BitiX` with varying bit densities. We evaluated different bit density combinations from 0.000015 to 0.5 for two input bitmaps and found symmetric behavior. Therefore, for simplicity, we modify the bit density of both bitmaps similarly. A mirrored behavior appears at bit densities greater than $D_{Bit} = 0.5$, since 1-fills with high bit densities show similar behavior as 0-fills with low bit densities. The COMPAX compression scheme is an exception, because

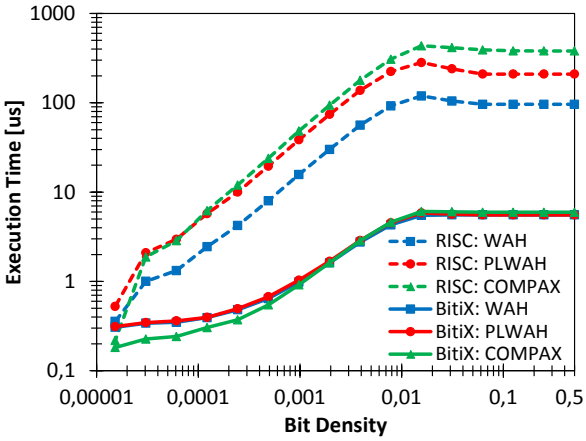


Fig. 4: Execution Time vs. Bit Density: AND operation of WAH/PLWAH/COMPAX compressed bitmaps, comparison of processors `BitiX` and `RISC`

it does not support 1-fills. The logical operation of COMPAX compressed bitmaps stays constant for bit densities greater than $D_{Bit} = 0.5$. As shown in Figure 4, the execution time decreases for all algorithms for lower bit densities, i.e., for enhanced compression ratios. The more zeros occur in bitmaps during compression, the more 0-fills are merged together, resulting in less code words, less iterations during processing, and in a shorter execution time. When considering bit densities in the range of $1/100 \leq D_{Bit} \leq 1/10$, the execution times remain relatively constant for the two processors. In such cases, we achieve our maximum speedup of `BitiX` compared to `RISC` for WAH, PLWAH, and COMPAX of 39.2x, 75.9x, and 97.7x, respectively. Please note, we only want to compare the implementations and not the algorithms themselves, as system administrators have to decide on the best algorithm depending on their requirements for space and computation.

The WAH and PLWAH processing show very similar execution times. Nevertheless, PLWAH performs a little worse than WAH because additional operations are required to merge them with 0- and 1-fills due to the available sparse code words of PLWAH. However, this effect only impacts the execution of the `RISC` processor. Since the `BitiX` processor fuses the AND operation into a single instruction, only the number of code words affects the execution time. COMPAX exhibits the best performance at very low bit densities. This can be traced back to a better compression ratio due to more advanced code words and, therefore, a lower execution time.

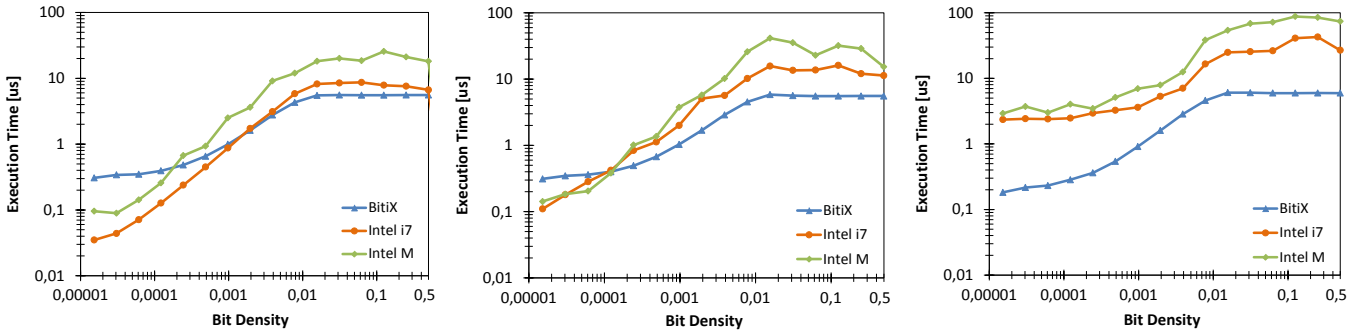
Table II shows the average performance of the four processors for bitmap processing. One can see that the `BitiX` processor always exhibits the best execution times without variations at different logical operators since the new instructions are independent from the input values. In contrast, the `RISC` processor as well as the Intel processors rely on a pure software solution with slight differences in handling the results. For instance, the OR operation of a literal with a 1-fill leads directly to a 1-fill. However, performing the XOR, a more time consuming logical operation is required to switch all bits.

When looking at the averaged results of the AND operation of WAH compressed bitmaps in Table II, the `BitiX` processor outperforms the `Intel i7-3960X` and the `Intel M-5Y10` of almost a factor of 1.3x and 3.3x, respectively.

Final edited form was published in "International Conference on Application Specific Systems (ASAP), Architectures and Processors. London 2016", S. 50-57,
ISBN 978-1-5090-1503-0
http://dx.doi.org/10.1109/ASAP.2016.7760772

Processor	WAH			PLWAH			COMPAX		
	AND	OR	XOR	AND	OR	XOR	AND	OR	XOR
RISC	59.68	48.27	49.20	115.04	110.45	108.45	184.23	202.38	192.14
BitiX	2.69	2.69	2.69	2.75	2.75	2.75	3.16	3.16	3.16
Intel i7-3960X	3.53	3.07	2.77	6.39	6.38	6.51	15.32	14.62	14.38
Intel M-5Y10	8.87	6.94	7.50	13.26	12.98	20.89	34.04	37.07	39.02

TABLE II: Execution Time [us] averaged over bit densities from 0.000015 to 0.5



(a) WAH (b) PLWAH (c) COMPAX

Fig. 5: Execution Time vs. Bit Density: comparison of BitiX processor and Intel cores, AND operation

However, when considering very low bit densities in Figure 5a, the Intel processors outperform BitiX and benefit from an up to 10x higher clock frequency as well as from a straightforward algorithm with an effective utilization of branch predictions. Especially the Intel M-5Y10 includes the advanced branch prediction of the Intel Broadwell architecture, leading to the surprising performance results in Figure 5b.

We observe a typical trade-off of our chosen stateless bitmap processing algorithms regarding efficiency and performance. While the compression ratio increases when using code words like PLWAH or COMPAX, the processors require a longer execution time. Figure 5 shows that relationship and, in contrast to this, that the BitiX processor behaves differently. The specialized instructions allow a similar processing on different coding schemes as long as no additional memory access is needed. Hence, the performance of BitiX is largely independent of the algorithms.

C. Energy and Power Consumption

We obtain the power consumption of our Xtensa processors for the given algorithms by analyzing the switching activities of the processor. Thereby, the resulting power per cycle is summed up and averaged over the total execution time of the chosen algorithm. We measure the power of the Intel processors with the integrated *Running Average Power Limit* (RAPL) counter, only measuring the core power (PP0). As expected, our additional instructions in the BitiX processor lead to a higher power consumption than the RISC processor (Table III). However, BitiX consumes 29.2x to 308.8x less power than the Intel processors.

The power consumption shows the average power usage, which is important for the system design. However, the energy usage of an operation should be favored when comparing the execution because it also depends on the runtime, e.g., even with a high power consumption, an operation could be highly energy-efficient by being faster than others. In our test scenario, the BitiX processor consumes 0.16 uJ for a WAH-AND operation. For the same operation, the energy-efficient Intel M-5Y10 uses 20.22 uJ, an almost 130x higher energy consumption while the Intel i7-3960X processor's energy is at 85.04 uJ more than 530x higher.

Processor	WAH	PLWAH	COMPAX
RISC	0.058	0.057	0.056
BitiX	0.096	0.078	0.075
Intel i7-3960X	24.09	15.15	19.90
Intel M-5Y10	2.28	2.58	2.57

TABLE III: Power Consumption [W] averaged over bit densities from 1/2 to 1/65536, AND Operation

VI. Related Work

Only few works concentrate on accelerating database algorithms by introducing additional and highly specialized hardware structures and instructions. The well-known instruction set extensions of general-purpose Intel processors like MMX, SSE, and FMA [12] were developed to deal with wider registers and data parallelism. Slingerland et al. [13] provide a list of such application-specific instruction sets. Furthermore, Schlegel et al. [14] used the STTNI extension of Intel SSE4.2 to improve the performance of sorted-set intersection algorithms. Besides that, in our previous work we provided a completely new instruction set for enhancing sorted-set and sorting applications [6] as well as primitive hashing operations [15]. We achieved a single-core speedup of up to 180x compared to modern x86 processors. The improvements result from the massive single-instruction, multiple-data (SIMD) processing. Research about query processing on low-energy Multiprocessor System-on-Chips has been presented in [16]. The presented platform could serve as a basis to integrate application-specific processors that provide a very low power consumption while reaching high performances.

In the field of bitmap indices, processing on bitmaps with new instructions was only mentioned but–to the best of our knowledge–not implemented. Firstly, Wu et al. [17] identified

Final edited form was published in "International Conference on Application Specific Systems (ASAP), Architectures and Processors. London 2016", S. 50-57,
ISBN 978-1-5090-1503-0
http://dx.doi.org/10.1109/ASAP.2016.7760772

simple compression algorithms without data dependencies for acceleration. Later, the developers of PLWAH [2] showed the benefit of the functions *popcount* and *bitscan* provided by x86 processors. The instructions count the set bits and determine the position of the first set least significant bits (LSB) within one clock cycle. Thus, PLWAH code words with the enclosed sparse bits benefit from the instructions.

From the background of observing increasing network traffic in real time, Fusco et al. [18] investigated WAH and PLWAH bitmap indexing algorithms on GPUs. Their used NVIDIA GPU achieves a speed of around 40 GBit/s for WAH and, thereby, outperforms the Intel i7 CPU by a factor of 5.

Another framework for bitmap indices with variable aligned lengths was presented by Guzun et al. [19]. Primarily, their performance improvement results from a previous profiling of the input bitmap that allows choosing an appropriate compression encoding scheme. Further, the work includes an input tuning parameter to select either a high compression ratio or improved execution time. In summary, their proposed framework executes queries that apply a logical AND on the bitmaps and are 30% faster.

VII. Conclusion and Future Work

In this paper, we evaluated the potential of an application-specific instruction set extension for compressed bitmap index processing. We discussed, which types of bitmap compression are suited for hardware acceleration and that the best compression algorithms have much in common. We could derive a common framework, which makes it relatively easy to implement these bitmap operations as hardware instructions. Finally, we chose three different bitmap encodings for which we implemented three bitmap operations each. We achieved performance results that keep up with modern x86 processors and show enormous power savings. Based on our feasibility analysis, there are two ways to integrate our bitmap instruction approach into larger systems:

Accelerator Approach: Our presented approach can be used to build a small specialized bitmap processing core (as shown in this paper). We also could add previously proposed sorted-set [6] and hashing instructions [15] to the processor by adding specialized cores. One or more of these cores fit next to bigger cores on the same chip to establish a heterogeneous *Multiprocessor System-on-Chip* (MPSoC). Operations can be offloaded to the small more energy-efficient cores, while at the same time, leaving the bigger cores to work on other tasks.

Extension Approach: The presented idea of our framework can also be used to implement instruction set extensions for bitmap processing directly in larger general purpose cores. This approach was applied before, for some instructions like hashing (e.g., PEXT) or encryption (e.g., AES) [12]. Hardware vendors could speed up bitmap processing and therefore query processing for large databases in general by extending their processors with our proposed instructions. Here, the integration would only require changes in the compilers to support the specialized instructions.

Both approaches would help to prevent dark silicon [5] by using parts of the chip only for specialized operations. In any case, we have demonstrated how to build a generic instruction set extension for a wide variety of state-of-the-art stateless compression algorithms and shown the general applicability and extreme relevance of HW/SW-CoDesign for efficient query processing in database systems.

VIII. Acknowledgments

This work has been supported in part by the German Research Foundation (DFG) within the Cluster of Excellence 1056 “Center for Advancing Electronics Dresden”, the Collaborative Research Center 912 “Highly Adaptive Energy-Efficient Computing”, is partially supported by “Ultra-Low Power Technologies and Memory Architectures for IoT” (692519 PRIME), and 610456 Euroserver. Further, we would like to thank Synopsys and Cadence for software and IP.

References

[1] G. Antoshenkov, “Byte-aligned Bitmap Compression,” in *Proceedings of DCC*, Washington, DC, USA, 1995.

[2] F. Deliège and T. B. Pedersen, “Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps,” in *Proceedings of EDBT*, 2010.

[3] F. Fusco, M. P. Stoecklin, and M. Vlachos, “NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic,” *Proceedings of VLDB Endowment*, 2010.

[4] W. Andrzejewski and R. Wrembel, “GPU-WAH: Applying GPUs to Compressing Bitmap Indexes with Word Aligned Hybrid,” in *Proceedings of DEXA*, 2010.

[5] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, “Dark Silicon and the End of Multicore Scaling,” in *ISCA*, 2011.

[6] O. Arnold, S. Haas, G. Fettweis, B. Schlegel, T. Kissinger, and W. Lehner, “An Application-Specific Instructions Set for Accelerating Set-Oriented Database Primitives,” in *Proceedings of SIGMOD*, 2014.

[7] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, “Understanding Sources of Inefficiency in Gernal-Purpose Chips,” in *Proceedings of ISCA*, 2010.

[8] M. Bassiouni, “Data Compression in Scientific and Statistical Databases,” *IEEE Transactions on Software Engineering*, 1985.

[9] O. Kaser, D. Lemire, and K. Aouiche, “Histogram-aware Sorting for Enhanced Word-aligned Compression in Bitmap Indexes,” in *Proceedings of DOLAP*, 2008.

[10] K. Wu, E. J. Otoo, and A. Shoshani, “An Efficient Compression Scheme for Bitmap Indices,” ACM Transactions on Database Systems, Tech. Rep., 2004.

[11] A. Colantonio and R. D. Pietro, “CONCISE: Compressed ’n’ Composable Integer Set,” *CoRR*, 2010.

[12] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corp., March 2014.

[13] N. T. Slingerland and A. J. Smith, “Multimedia extensions for general purpose microprocessors: a survey,” *Microprocessors and Microsystems*, vol. 29, 2005.

[14] B. Schlegel, T. Willhalm, and W. Lehner, “Fast Sorted-Set Intersection using SIMD Instructions,” in *Proceedings of ADMS*, 2011.

[15] O. Arnold, S. Haas, G. Fettweis, B. Schlegel, T. Kissinger, T. Karnagel, and W. Lehner, “HASHI: An Application-Specific Instruction Set Extension for Hashing.” in *ADMS*, 2014.

[16] A. Ungethüm, D. Habich, T. Karnagel, W. Lehner, N. Asmussen, M. Völp, B. Nöthen, and G. Fettweis, “Query Processing on Low-Energy Many-Core Processors,” in *Proceedings of HardBD*, 2015.

[17] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg, “Notes on design and implementation of compressed bit vectors,” in *Technical Report LBNL/PUB-3161*, 2001.

[18] F. Fusco, M. Vlachos, X. Dimitropoulos, and L. Deri, “Indexing million of packets per second using gpus,” in *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC)*, 2013.

[19] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin, “A Tunable Compression Framework for Bitmap Indices,” in *Proceedings of ICDE*, 2014.