SIMD Acceleration for Index Structures

Marten Wallewein-Eising
Otto-von-Guericke University
Magdeburg, Germany
marten.wallewein-eising@st.ovgu.de

Abstract—

- **summary:**
    Give short an overview of SIMD and modern index structures
    Explain what are the problems of the "old" index structures made for disk-based database systems
    Explain which approaches were made to adapt index structures to modern systems and what they have in common and what are differences
- **Why is this work important:**
    Give a state of current development of the index structures
    Collect common approaches to adapt other index structures TODO: ReThink
- **K-ary search trees, FAST, VAST and ART compared**
- **Contribution: What are important approaches used by different implementations to adapt index structures to modern systems**

*Index Terms—***SIMD, index**

I. INTRODUCTION

After decades of creating and improving index structures for disk-based database systems, nowadays even large databases fit into the main memory. Since index structures like the B^+-tree or the radix tree have an important part in database systems to realise scan or range-based search operations, these index structures experienced many adaptions to fulfill the needs of modern database-systems. Instead of overcoming the bottleneck of IO-oprations from disk to RAM, the target of modern index-structures is to improve the usage of CPU cache and processor architectures.

Several index structures have already shown that the bottleneck from RAM to CPU can be overcome using Single Instruction Multiple Data (SIMD) [1] operations. These index structures include: the K-ary Search Tree (Seg-tree) [3], Adapted Radix Tree (ART) [4], Fast Architecture Sensitive Tree (FAST) [6], and Vector-Advanced and Compressed Structure Tree (VAST) [5]. As the authors of VAST-Tree show, important causes for increased runtime are cache misses and branch mispredictions. To overcome branch mispredictions and to decrease CPU cycles, SIMD is used in modern index structures for tree traversal [2]. The authors of the k-ary search show how to use SIMD to compare mutliple keys in one CPU cycle. To decrease cache misses, the authors of FAST and ART show how to adapt index structures to the cache line size.

TODO: Fix! All approaches use SIMD only for key comparison within tree traversal and try to decrease the key size to fit more keys into one SIMD register. Therefore FAST and
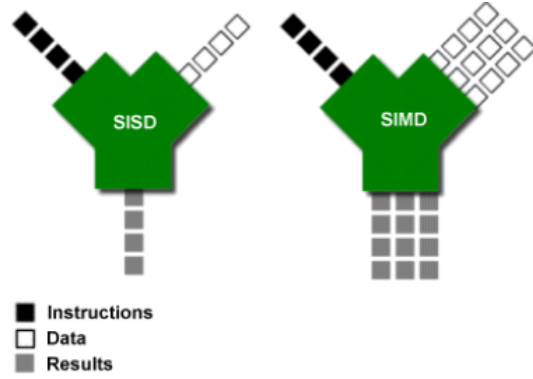

Fig. 1. Coherence between data, instructions, and the results of SISD and SIMD.

Seg-tree only provide implementations for search algorithms. We consider the design approaches VAST and ART make to implement operations like update and insert und name ideas to use SIMD for them. Consequently, with this work we make the following contributions:

- We compare different adaptions of index structures to fulfill requirements of modern database systems
- We highlight the usage of SIMD and the cache line adaptions in all approaches
- We show opportunities for adaption for other approaches to use SIMD

We organized the rest of the paper as follows. In Section 2 we give the preliminaries for SIMD in general and for the use in index structures. In Section 3 we analyse the different approaches of adapted index structures and evaluate the comparision in Section 4. In section 5 we name related work. In Section 6 we present our conclusion und describe future work in Section 7.

II. PRELIMINARIES

A common approach of decreasing CPU cycles for algorithms is to adapt the algorithm to pipelining. While one instruction is executed, the next instruction is already fetched. This approach executes one instruction on one data item, called Single Instruction Single Data (SISD). In contrast to execute one operation on one data item after another, the idea of SIMD is to execute a single instruction on multiple data. In Figure 1 we show the coherence between data, instructions, and the results of SIMD and SISD. Modern CPUs have

SIMD instruction	Explanation
__m128i _mm_load_si128 (__m128i *p)	Loads a 128-bit value. Returns the value loaded into a variable representing a register.
__m128i _mm_cmpgt_epi32 (__m128i a, __m128i b)	Compares 4 signed 32-bit integers in a and 4 signed 32-bit integers in b for greater-than.

additional SIMD registers along with an additional instruction set adapted to process multiple data with a single instruction. In Table 1, we show some SIMD instructions from Streaming SIMD Extensions 2 (SSE2). We consider _mm_cmpgt_epi32 as example to show how SIMD works. After loading 4 different signed 32-bit integers in a and 4 equal signed 32-bit integers in b, we compare them using _mm_cmpgt_epi32. The comparision returns a bitmask showing which of the search keys in a is greater than the one in b. This example i... es the performance of comparision by times of four.

Consequently, the main advantage of SIMD is to process multiple data parallel in contrast to pipelining and SISD. The main restriction of SIMD instructions is that a sequential load of data is required. To load data into a SIMD registert, the data has to be stored consecutively in main memory. Additionally, the size of SIMD registers is limited. Therefore processing data types of the common size of 64-bit and more lead to a small performance increase since only few data items are processed with a single instruction.

Polychroniou et al. [7] show two general approaches to use SIMD in in-memory databases, horizontal and vertical vector processing. They name the comparision of one search key to multiple other keys horizontal vectorization, whereas processing a different input key per vector lane is named vertical vectorization.

Since FAST, Seg-Tree, ART and VAST only use horizontal vectorization, we focus on this approach. For example, Zeuch et al. [3] use 128-bit SIMD registers and adjusted SIMD operations to load data into a register and to compare the data of one SIMD register with another. A 128-bit SIMD register processes sixteen 8-bit or eight 16-bit data items with one instruction. In Table 1 TODO: Insert Table! we show a comparision of key size and the number of keys that can be processed parallel with one SIMD instruction.

III. ADAPTED TREE STRUCTURES

A. Seg-Tree

Zeuch et al. adapted the B^+-Tree by having a k-ary search tree as each inner node, called segment, and perform a k-ary search on each segment. In Figure 2, we show the adaption of nodes made by Zeuch et al. for Seg-Tree. The k-ary search bases on the binary search but divides the search space into k partitions with k-1 seperators. Compared to binary search the k-ary search reduces the complexity from $O(\log 2n)$ to $O(\log kn)$. They consider m as the most bits to represent a datatype and $|SIMD|$ as the size of SIMD-register, called SIMD bandwith. Furthermore, $k = \frac{|SIMD|}{m}$ defines the count of partitions for the k-ary search.
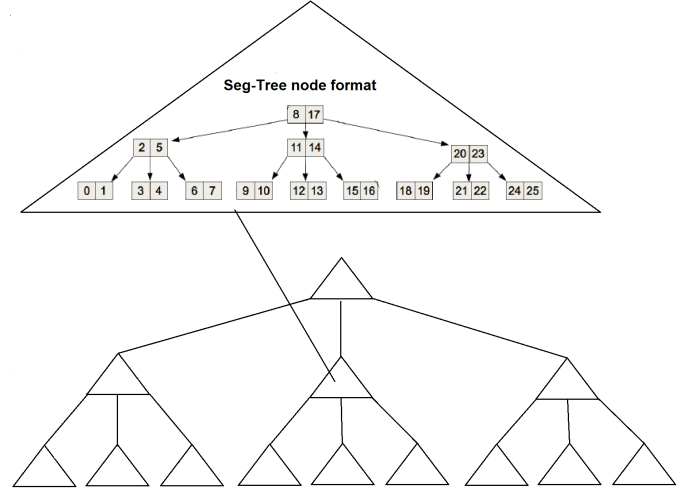

Fig. 2. Inner node format of Seg-Tree

As mentioned before, each segment of the Seg-Tree is a k-ary search tree. To perform a k-ary search on a segment, Zeuch et al. make a linearization of the segment. They show two algorithms for linearization, breadth-first search and depth-first search. Because of the condition $k = \frac{|SIMD|}{m}$, each partition of the k-ary search fits into a SIMD register and is compared to the search key. A perfect k-ary search tree contains $S_{max} = k^h - 1$ keys for an integer $h > 0$. The considered search algorithm only works for sequences with a multiple of $k - 1$ keys. In case of sequences with less than a multiple of $k - 1$ keys, they replenish the sequence with Elements having the value $k_{max} + 1$ for the maximal key value k_{max} in the sequence. Consequently, the adapted search algorithm works for sequences with less than a multiple of $k - 1$ keys.

The performance of Seg-Tree depends on k-ary search. The smaller a key the more keys are compared parallel. According to the relevance of 32 and 64-bit data types in modern systems, the k-ary search performance increases only by the factor of four for 32-bit types and two for 64-bit types.

Zeuch et al. also show the k-ary search on an adapted prefix tree *(trie for short)* called Seg-Trie. A trie is a search tree where each node stores a part of the key. Each node is again designed as a k-ary search tree. Complete keys are stored in leave nodes or are build by concatenating partial keys from the root node to a leave node. This approach benefits of the separation of the keys in different levels of the tree. Consequently, the compare keys are smaller and more keys can be compared parallel. The Seg-Trie$_L$ is defined as a balanced

Fig. 3. (a) Node indices (=memory locations) of the binary tree (b) Rearranged nodes with SIMD blocking (c) Index tree blocked in three-level hierarchy first-level page blocking, second-level cache line blocking, third-level SIMD blocking.

trie where each node on each level contains one part of the key with L Bits. The tree has $r = \frac{m}{L}$ levels $(E_0, E_1, .., E_r)$, where m is the number of most bits to represent the data type.

To perfom a tree traversal on the seg-trie, the search key is split into r segments and each segment r_i is compared to the level E_i. If a matching partial key is found in one node of E_i, the search continues at the referrenced node for the partial key. If no match of the partial key is found, the Seg-Trie does not contain the search key and the search is finished. Consequently, the advantage of Seg-Trie against tree structures is the reduced comparision effort from non-exisiting key segments.

B. FAST

Changkyu et al. adapted a binary tree to optimize for architecture features like page size, cache line size, and SIMD bandwith called Fast Architecture Sensitive Tree. In contrast to Seg-tree, FAST is also adapted to disk-based database systems. Changkyu et al. show the performance increase because of decreasing cache misses and better cache line usage. In order to optimize for architectural features, tree nodes are rearranged for hierarchical blocking. In Figure 3, we show an index tree blocked in three-level hiearchy introduced by Changkyu et al.

According to Figure 3, we define $d_K = 2$ as the depth of a SIMD block, $d_L = 5$ as the depth of a page block, $N_K = 2^{d_K} - 1$ as the number of keys that fit into a SIMD register, and $N_L = 2^{d_L} - 1$ as the number of keys that fit into a cache line. To rearrange the nodes, Changkyu et al. sorted the nodes, as shown in Figure 3 (a). Afterwards, they combine the first N_K keys and laid them out in breadh-first fashion (the first key as root, the next $N_K - 1$ keys as children on depth 2). These first N_K represent a SIMD block. They continue to lay out the next keys in same fashion, building the SIMD blocks with root keys 3,6,9, and 12, as shown in Figure 3 (b). This process continues for all sub-trees that are completely within the first d_L levels from the root. If a sub-tree does not completely fit in the first d_L levels, the keys are laid out according to the appropriate number of levels $(d_L \% d_K)$ as shown in depth 4 in Figure 3 (b). The first N_L keys represent a cache line block. We define $d_P = 31$ as the depth of a page block, $N_P = 2^{d_P} - 1$ as the numberof keys that fit into a page. The procedure of

hierarchical blocking continues analogue until the N_P keys are laid out, representing a page block. After building one page block, the next page block is build up the same way. In Figure 3 (c) we show the different blocks in a hierarchical blocked index tree.

Changkyu et al. show implementations for building and traversing the tree adapted for CPU and also for GPU. Bulding up the tree, SIMD is used to computing the index for N_K keys within the SIMD level block parallel, achieveing around 2X SIMD scaling as compared to the scalar code. With a Core i7 processor, the runtime of building a FAST tree with $64M$ tuples is less than 0.1 seconds. Traversing the tree, they compare one search key to multiple keys of the index structure, analogue to the k-ary search. To use the complete bandwith of cache and main memory within the search, blocks are loaded complete into associated memories from large blocks to small blocks. For a page block, at first the page is loaded into main memory. Then cache line blocks are loaded one after another in the cache and for each cache line block, the included SIMD blocks are loaded into the SIMD register. All keys of this SIMD block are compared with one SIMD instruction. After looking up the resulting bitmask, the corresponding SIMD block is loaded (including the load of the surrounding larger blocks) until the key is found or the last level of the index structure is reached.

Changkyu et al. consider the search performance as queries per second. The CPU search performance on the Core i7 with 64M 32-bit (key, rowID) pairs is 5X faster than the best reported number [8 TODO ref], resulting in a throughput of 50M search queries per second. The GPU search on the GTX 280 is around 1.7X faster than the best reported number, resulting in 85M search queries per second. Considering larger index structures, the GPU performance increase exceeds the CPU performance increase, because TLB and cache misses grow up and the CPU search becomes memory bandwidth bound.

C. ART

D. VAST

IV. Evaluation

In common:

- SIMD instructions used to compare the search key with multiple keys of the index
- Segmenting tree to blocks for a better usage of cache lines, save the data of the nodes in an adapted way
- The keys should be as short as possible to compare more keys in one step and to decrease the passed data to the cache line
- Each approach improves the tree traversal

Differences:

- Node compression in VAST, Path compression in ART and K-ary seg trie
- FAST and K-ary trees readonly to improve traversal, ART and FAST adapt insert too
- FAST uses and K-ary trees will use GPU calculation instead of CPU

Why performance can not be compared in a useful way... performance issues:

- Seg-Tree:

V. Related Work

TODO:

- ART and VAST compared to FAST??
- Ideas and implementations of the adapted trees already in III...
- KD-Tree with SIMD

VI. Conclusion

VII. Future work

Open questions, use SIMD for tree creation/updates instead of only for traversal

References

[1] Mohammad Suaib, Abel Palaty and Kumar Sambhav Pandey, "Architecture of SIMD Type Vector Processor" in International Journal of Computer Applications (0975 - 8887) Volume 20 No.4, April 2011.

[2] Jingren Zhou and Kenneth A. Ross "Implementing Database Operations Using SIMD Instructions" in ACM S1GMOD '2002 June 4-6, Madison, Wisconsin, USA

[3] Steffen Zeuch, Frank Huber and Johann-Christoph Freytag "Adapting Tree Structures for Processing with SIMD Instructions" in Proc. 17th International Conference on Extending Database Technology (EDBT), March 24-28, 2014, Athens, Greece

[4] Viktor Leis, Alfons Kemper and Thomas Neumann "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases"

[5] Takeshi Yamamuro, Makoto Onizuka,Toshio Hitaka, and Masashi Yamamuro "VAST-Tree: A Vector-Advanced and Compressed Structure for Massive Data Tree Traversal" in EDBT 2012, March 26-30, 2012, Berlin, Germany.

[6] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt and Pradeep Dubey "FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs" in SIGMOD10, June 6-11, 2010, Indianapolis, Indiana, USA.

[7] Orestis Polychroniou, Arun Raghavan and Kenneth A. Ross, "Rethinking SIMD Vectorization for In-Memory Databases" in SIGMOD15, May 31-June 4, 2015, Melbourne, Victoria, Australia.