Towards Data-Based Cache Optimization of B$^+$-Trees

Roland Kühn
Databases and Information Systems Group (DBIS)
TU Dortmund University
roland.kuehn@cs.tu-dortmund.de

Daniel Biebert
Design Automation for Embedded Systems Group (DAES)
TU Dortmund University
daniel.biebert@tu-dortmund.de

Christian Hakert
Design Automation for Embedded Systems Group (DAES)
TU Dortmund University
christian.hakert@tu-dortmund.de

Jian-Jia Chen
Design Automation for Embedded Systems Group (DAES)
TU Dortmund University
jian-jia.chen@cs.tu-dortmund.de

Jens Teubner
Databases and Information Systems Group (DBIS)
TU Dortmund University
jens.teubner@cs.tu-dortmund.de

ABSTRACT

The rise of in-memory databases and systems with considerably large memories and cache sizes requires the rethinking of the proper implementation of index structures like B$^+$-trees in such systems. While disk block-sized nodes and binary search were considered as good in the past, smaller node sizes and cache-friendly linear search within nodes can be noticeably more performant nowadays. Considering the probabilistic distribution of lookup values to the B$^+$-tree as part of a memory-friendly and cache-aware layout is a consequent next step, which is studied in this paper. Favoring frequently visited nodes and paths in the regard of cache hits can improve the overall performance of the tree and, thus, of the entire database system. We provide such an optimized B$^+$-tree layout, which takes the probabilistic distribution of the lookup values as a basis. Experimental evaluation shows that choosing rather small node sizes in combination with our optimization algorithm can improve the performance by up to 26% in comparison to a default baseline.

CCS CONCEPTS

• **Information systems → DBMS engine architectures**.

KEYWORDS

in-memory index, B+ tree, cache-aware layout

ACM Reference Format:
Roland Kühn, Daniel Biebert, Christian Hakert, Jian-Jia Chen, and Jens Teubner. 2023. Towards Data-Based Cache Optimization of B$^+$-Trees. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference acronym 'XX, June 19, 2023, Seattle, WA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

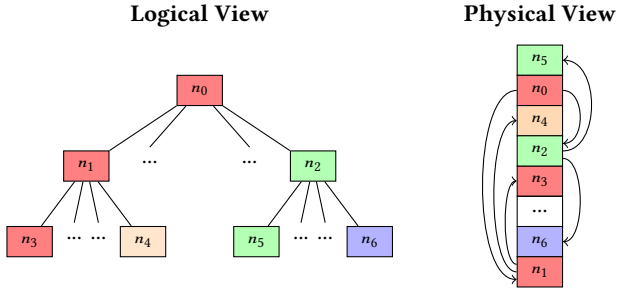


Figure 1: The logical view of a tree is shown on the left side. The colors reflect different paths in the tree. The right side depicts a possible physical layout of the tree in memory.

1 INTRODUCTION

Index data structures in modern database systems are a highly performance-engineered field since the availability of efficient index structures is a crucial aspect to the overall performance of the entire database management system (DBMS). Historically motivated by the blockwise management of hard disks or SSDs, B$^+$-tree structures with node sizes similar to the block size are a well-established and well-studied topic [4].

When it comes to in-memory databases or in-memory indices potentially on modern memory technologies, however, several observations can be made that no longer follow the historical design principle of B$^+$-trees. For example, the nodes of in-memory B$^+$-trees are no longer bound to a fixed block size of a secondary storage medium. The question, how to design memory and especially cache-friendly in-memory B$^+$-trees remains an active research topic and allows an extended design space of the index data structures.

One specific problem that arises with tree-based in-memory index structures is the structure and natural growth of the trees. For example, a tree-based index does grow logically in depth, as shown on the left in Figure 1. However, the arrangement of individual nodes in memory does not necessarily reflect this arrangement. For example, a split of a leaf node can cause further splits in the parent nodes and possibly even create a new root node, which might then be placed at a —from the application's perspective— random memory location. As illustrated in Figure 1 (on the right side), this results in arbitrary jumps through the address space when

navigating from the root to a leaf node. Modern hardware, however, is endowed with features, e.g., hardware prefetching or out-of-order execution, that exploit predictable memory access patterns. Hence, as many nodes as possible of a tree path should be placed at a contiguous memory region.

Despite general cache-aware optimization of the tree implementation itself, data-based cache optimization provides promising optimization potential, which, to the best of our knowledge, has not been explored for B^+-trees so far. In the field of machine learning, especially for decision trees within random forests, optimizing the tree structure of the binary decision tree based on an empirical probabilistic model of the training data towards the cache hierarchy is shown to provide massive performance improvement [1]. Although database systems do not provide a comparable static data distribution as the training data of a machine learning problem, repeating sequences of requests can be often found in database applications, which provides a similar potential for data-based cache optimization.

Consequently, in this paper, we study the adaption of a state-of-the-art in-memory B^+-tree to a proper memory layout, which favors frequently accessed nodes and paths in the tree in the regard of cache hits. The information of frequently used nodes and paths is gathered by maintaining an empirical probabilistic model of past requests to the B^+-tree, which reflects the probabilistic distribution of future requests. For the identified frequently used nodes and paths, prefetching and eviction of the cache is controlled by a corresponding memory layout to minimize the number of cache misses for these nodes and paths. We realize that by counting the number of requests following a certain key in the B^+-tree, forming a histogram for every node. This histogram gives relative probabilities, which can be used to extract highly probable paths, so-called hot paths, through the tree. We consequently place these hot paths in linear ascending memory locations, such that hardware prefetching of the cache loads the most probable next node already before it is needed. Assuming data age-based eviction, like Least recently used (LRU), hot paths are also protected from eviction since they are frequently accessed. The question of a proper runtime collection of the empirical probabilities in the tree nodes and the re-layouting of the tree upon changes in the probabilities during runtime is out of the scope of this paper.
In short, we provide the following contributions:

- A probabilistic data-based model for the frequency of usage of nodes and paths in B^+-trees.
- An algorithm that optimizes the layout of hot paths in B^+-trees for modern hardware.
- An experimental evaluation of the effectiveness of the optimization algorithm on a static setup.

2 BACKGROUND

B^+-tree variants and many other tree- and trie-based index structures have been subject to extensive research in the past decades (e.g., [2–5, 10, 12, 14, 16, 17, 19, 23]). Originally optimized for storage devices like hard disks, B^+-trees and variants are also widely used in in-memory database management systems. Over the years, several specialized tree-based structures were proposed that are especially designed for the use on modern hardware with huge main memories and deep cache hierarchies. For example, the optimization of B^+-tree layouts towards caches has been examined in [20], where child nodes of a node are stored contiguously in memory and only one pointer to that memory region is stored in the parent node. The individual child nodes can then be accessed by adding an offset to that pointer. In [22], cache optimization schemes for memory-based key-value stores are presented. The effect of different node sizes of a cache-sensitive B^+-tree (CSB^+-tree) to main memory with respect to cache and TLB misses has been modelled and analyzed in [6]. In contrast to previous assumptions, the authors show that node sizes larger than the size of cache lines can improve the performance. [8, 18] shows how memory accesses in index structures can be improved by exploiting software-based prefetching. Hardware-sensitive searching methods have been examined and analyzed in [21].

The usage of the empirical distribution in the processed data for cache-based performance optimization is studied extensively on decision trees within random forest machine learning methods [1]. The authors enable different tree implementations to feature a large degree of freedom for the memory placement of single nodes. Empirical probability distributions from the training dataset are used to identify often taken paths in the tree, which are layout in a consecutive, cache-friendly manner. This improves the cache locality and the memory latency of individual decision trees during the execution of the test dataset. Although the application of this paper is no machine learning application with training and test datasets, we focus similarly on the cache optimization of often taken paths in a B^+-tree model.

3 CACHE ARCHITECTURES

In this paper, we exploit non-uniform distributions of the frequency of accesses to different parts of an index in a DBMS in order to optimize system performance due to the exploitation of locality and prefetching in the system's cache architecture. Since cache strategies are not always well known for existing systems, we lay out the basic abstract properties of the management of the CPU cache and translation lookaside buffer (TLB) in this section.

We generally target our method to modern CPUs with advanced memory architectures, including a cache hierarchy with various levels, a TLB, eventually also across various levels and advanced eviction and speculative prefetching strategies in the caches and TLBs. Although the details about the exact realization of these strategies are usually not documented for available CPUs, a basic behavior can be assumed, which then at least is approximately realized by the CPU. Microbenchmarking, in general, could help to gain a deeper understanding of the internal management of caches and TLBs. However, it still only derives an approximated model.

Regarding the hierarchical organization of caches and TLBs, lower levels are only queried when a miss on a higher level happens. Hence, small working sets usually are served by higher level caches and TLBs, while lower levels are usually beneficial for larger working sets. Due to a usually similar management of eviction and prefetching across the levels, an optimization of the memory behavior of software with regard to this eviction and prefetching

strategies targets different levels of the hierarchy in a unified way, depending on the targeted working set size.

Focusing on the eviction strategy, caches are managed on the granularity of cache lines, which usually span multiple CPU words. An eviction happens when other content has to be loaded into the cache, but no space is available for this content. Although the strategy of choosing the victim for eviction is complex and usually unknown in modern CPUs, an age-based strategy, which protects frequently accessed data from eviction, is usually realized. Consequently, ensuring frequent accesses to certain cache lines from the software can protect them from eviction.

Focusing on prefetching, modern CPUs are able to speculatively load contents to the cache or to the TLB, which are expected to be accessed in the future in order to reduce the cache miss penalty for these accesses. The prefetched contents are decided based on patterns of past memory accesses. One easily exploited pattern is linear increasing access in the memory address space. Consequently, software can exploit eviction by placing contents in linear ascending memory ranges, such that these contents are hit by prefetching.

Despite caches, TLBs are similarly managed, such that often-used addresses are less likely to be evicted than seldom used addresses. From the perspective of optimization, exploiting locality in the TLB can be done similarly as for caches when the assumed line size is the size of a memory page. In consequence, TLBs can also be interpreted as a last layer of very coarse managed caches for the optimization.

4 PROBABILISTIC MODELING

The allover aim of this paper is to show how data-based probability distributions can be used to optimize the cache and TLB behavior of tree-based index structures. Hence, we detail the basic probabilistic model, the relation to the memory layout, and the possible collection of an approximation of the probability distribution of the B$^+$-tree in this section. Although the overall aim is to optimize the performance of a database system, the B$^+$-tree in insolation can be considered as a simple key-value store, which stores the associated values with a certain key. Hence, a data set which is used with a B$^+$-tree can be also expressed as a set of such key-value pairs $I = \{(k_1, v_1), (k_2, v_2), ..., (k_n, v_n)\}$ if it is used to populate the tree or just as a set of lookup keys $L = \{k_1, k_2, ..., k_m\}$ if it is used to perform lookups in the tree. The B$^+$-tree then forms a structure where incoming keys are compared to the keys that are currently stored in a node. This comparison determines which child node should be further used for the lookup. New keys are eventually inserted in leaf nodes, which may lead to split and rebalance operations of the entire tree.

4.1 Probabilistic Model Description

During lookup, every node n_x has a certain amount of comparison keys $C_{n_x} = \{ck_1, ck_2, ..., ck_n\}$, where the incoming lookup key lk is compared to. The corresponding child node n_y then is determined by the evaluation of the criterion $ck_{a-1} \leq lk < ck_a$, where n_y is the associated child node to the comparison key ck_a. Consequently, a given set of lookup keys LK forms a discrete probability distribution within every tree node n_x: $P_{n_x} = \{p_{ck_1}, p_{ck_2}, ..., p_{ck_n}\}$. This distribution is collected by simply counting how often the child of key ck_y is used for further lookup in the tree and normalizing all counted values in each node.

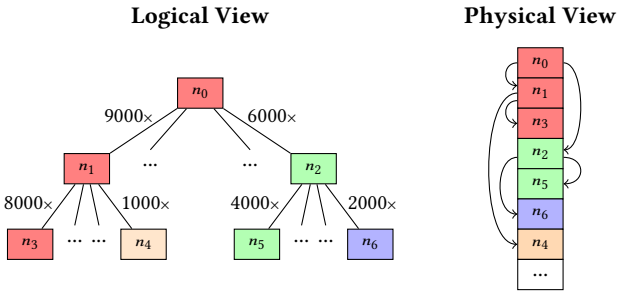


Figure 2: Schematic illustration of the node ordering according to our model. The numbers at the edges indicate how often this part of the path has been taken.

The point in time where a memory layout for a B$^+$-tree is decided, lies between the creation of the majority of the tree and the execution of many lookup operations to the tree. Consequently, the probabilistic distribution $P_{n_x}^{future}$ of the future lookup keys LK_{future} is crucial to the cache-aware layout. In the rest of this paper, we take the idealized assumption that the distribution of future lookups is exactly the same as the distribution of past lookups and hence the probability distribution in every node is precisely well-known. For database systems which are used in the backend of deterministic software, such idealized assumptions can even be considered.

4.2 Tree Layouting

Interpreting the probabilistic distribution in every node globally for the tree, the model allows forming a global probabilistic model for paths and sequences through the tree, which are likely taken by lookup keys. Generally, B$^+$-trees grow at a certain moment beyond the size of caches and probably also span more memory pages than the TLB can hold. Even in modern CPUs, the size of the TLB is limited to a few hundred or a little more than 1000 entries, which are not sufficient to hold all the pages that are needed to keep the addresses of all nodes. Once this situation is present, accesses to the B$^+$-tree, which include more nodes being present in the cache or more pages being present in the TLB are significantly faster than accesses which include fewer nodes in the cache or pages in the TLB. Consequently, the ultimate optimization objective is to layout the tree in such a way that frequently used paths, i.e., a subset of nodes, are more likely to be present in the cache than less frequently used nodes. Ideally, one would wish to greedily populate the cache with the most frequently used nodes. Since most modern systems, however, are not equipped with manual manageable caches, the memory layout of the tree has to be formed in a way that prefetching and eviction of the cache forms an approximation towards that ideal target.

In order to accommodate for the principle of prefetching, we first extract highly probable paths through the B$^+$-tree from the collected probabilistic model. These paths are greedily extracted from the tree nodes. In detail, the root node, the highest probable child, the highest probable child of this child, and so on form the

first highest probable path. Subsequently, the next highest probable child from the root node forms the beginning of the next highest probable path. It should be noted that these so-called hot paths are disjoint, and every node of the tree belongs to at most one hot path. After forming the hot paths on a logic level, we layout the B$^+$-tree in memory due to placing the nodes of the hot paths in linear ascending memory locations. An example of such a placement is illustrated in Figure 2. Across hot paths, also, the order of the probability is maintained. During inference of the tree, the prefetcher likely loads linear ascending memory addresses to the cache. This exactly covers the highest probable tree nodes to be accessed in future in our layout. Consequently, future accesses following the probabilistic distribution benefit from the layout since the number of cache hits is optimized.

The proposed mapping strategy for the B$^+$-tree further also accommodates for the principle of eviction. We form the hot paths while starting with early levels of the tree, i.e., preferably the root node or the direct children. In consequence, these nodes are accessed more often than deeper nodes of the tree. Consequently, the cache lines are more often touched and protected from eviction. As we place the hot path in linear ascending memory addresses, they are also partially covered by these frequently accessed cache lines and protected from eviction.

4.3 Probabilistic Model Approximation

Although it is not the focus of this paper to explore means to estimate $P_{n_x}^{future}$ precisely, simple mechanisms can be considered. Forming a proper set of lookup keys LK, which reflects a realistic and usable probability distribution, can be approached in multiple ways. Ideally, assuming that future requests to the tree would form a set of lookup keys LK_{future} and hence a probability distribution $P_{n_x}^{future}$ for all nodes, a set of lookup keys LK_{past} should be collected from past requests in such a way that the corresponding distribution $P_{n_x}^{past}$ is highly similar to $P_{n_x}^{future}$. One simple approach towards a collection of the probability distribution would be to track lookup keys during the entire lifetime of the B$^+$-tree and incrementally update the stored local probability distribution per node. Normalization of the stored counters could also be only performed when the distribution is actually queried, such that the overhead for this form of data collection can be minimized. In a productive system, however, the distribution of future lookup keys must not always be the same and can change over time. Considering the simple example of an online shop where products are looked up based on user requests. The distribution of looked-up products may even change multiple times over the day, depending on the group of users which are currently online. Consequently, it could be considered to update the probability distribution on the nodes in a certain sliding window manner, such that drifts in the distribution can be discovered with a relatively low latency and properly reflected.

5 EVALUATION

In this section, we present an experimental evaluation of our approach a state-of-the-art in-memory index. We use an implementation of a B$^+$-tree with optimistic lock coupling [11, 13] that was also used by Wang et al. [23].

5.1 Evaluation Setup

For our experiments, we use a B$^+$-tree with 100 Mio. random keys with a size of 8 bytes per key and per payload each. Since the keys within the nodes are typically ordered, a search algorithm like binary search is normally used to locate the position of the pointer to the next child node or the payload. For our benchmarks, we additionally implement a linear search as an alternative to the binary search, since it offers a sequential access pattern and may be better suitable for the hardware prefetcher. To reflect a realistic scenario, where just a smaller fraction of the keys in the B$^+$-tree is accessed and some of these keys are accessed very often, we use a Zipfian distribution with a skew of 0.99 on 10 Mio. random keys of our tree to generate our workload. The complete workload consists of 200 Mio. read operations.

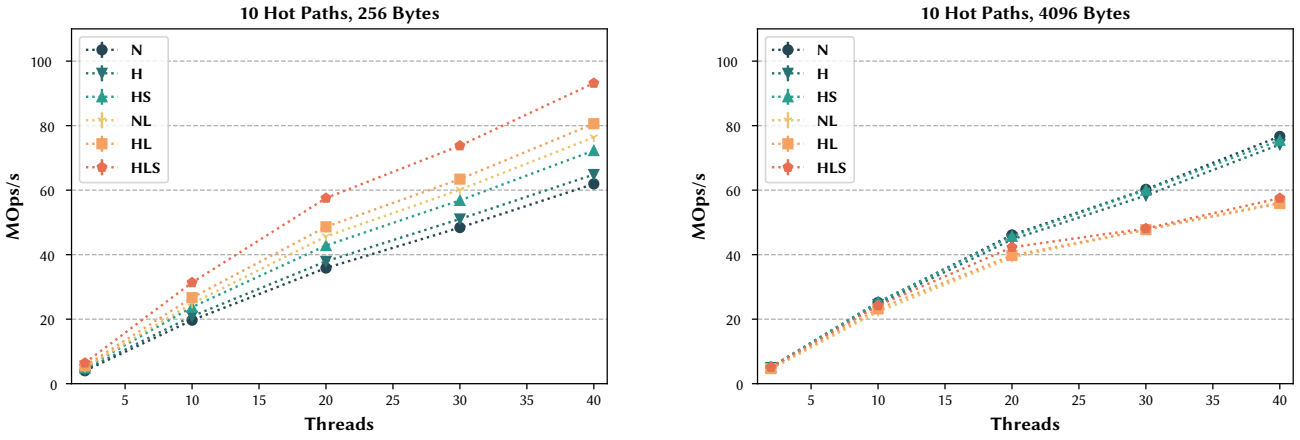
To investigate the effectiveness of our approach, we study the impact of considering hot paths by first extracting a few hot paths (10) and then gradually increasing the number of hot paths up to 10 000. Since we know which keys are accessed very often, we log the nodes of these keys during a tree traversal once before the remapping takes place. The hottest nodes are then placed in contiguous memory locations, starting with the path that is most likely to be taken.

Apart from the **Native (N)** implementation, where the B$^+$-tree is evaluated as it is, without any optimization, we consequently evaluate five different strategies.

1. **Hot path (H)**: In this setting, we only use the access information of the hot paths to generate a new mapping that tries to put as many nodes of a single hot path at contiguous memory locations. Nodes that are not part of a hot path will be stored at a random position.
2. **Native + Linear Search (NL)**: This strategy does not change the mapping of the nodes but uses a linear search instead of a binary search within the nodes to locate the correct position of a pointer/value.
3. **Hot path + Linear Search (HL)**: This approach combines the remapping of nodes from (1) with the linear search as used in (2).
4. **Hot path + Sort (HS)**: The nodes will be remapped like in (1), but this time we also sort the remaining nodes that are not part of any hot path in descending order by the number of total read accesses and remap them as well.
5. **Hot path + Linear Search + Sort (HLS)**: This configuration combines the approaches (3) and (4) by using linear search within the nodes, remapping the nodes of the hot paths and sorting the remaining nodes.

Besides different remapping schemes and amount of hot paths, we also evaluate the impact of different node sizes, since we expect that in combination with our strategies e.g. several nodes of a hot path can end up on a single page and therefore the TLB usage can be improved. For the experiments, we use five different node sizes (256 bytes, 512 bytes, 1024 bytes, 2048 bytes and 4096 bytes). Depending on the node size, the size of the tree also ranges from ≈ 2.1 GB (with a node size of 4096 bytes) and ≈ 2.7 GB (with a node size of 256 bytes).

The experiments are run on a machine with Ubuntu 20.04 and two Intel Xeon Gold 6230 clocked at 2.1 GHz with 20 physical cores



(a) Node size of 256 Bytes

(b) Node size of 4096 Bytes

Figure 3: Throughput of all evaluated strategies for a node size of 256 bytes and 4096 bytes with only 10 hot paths used.

and 40 threads each. Each processor has 20 × 32 KB of L1D Cache, 20 × 1 MB of L2 Cache, 27.5 MB of L3 Cache, a total memory of 192 GB DDR4 RAM. The first level TLB has a size of 64 entries and the second level TLB has a size of 1536 entries. We use perf [15] and Intel VTune [7] to collect information about hardware events.

5.2 Result Discussion

As described in Section 5.1 we examine our approaches with different numbers of hot paths (10, 100, 1000, 10000). Even with a low number of 10 hot paths, we can see promising results with our remapping strategies for our workload.

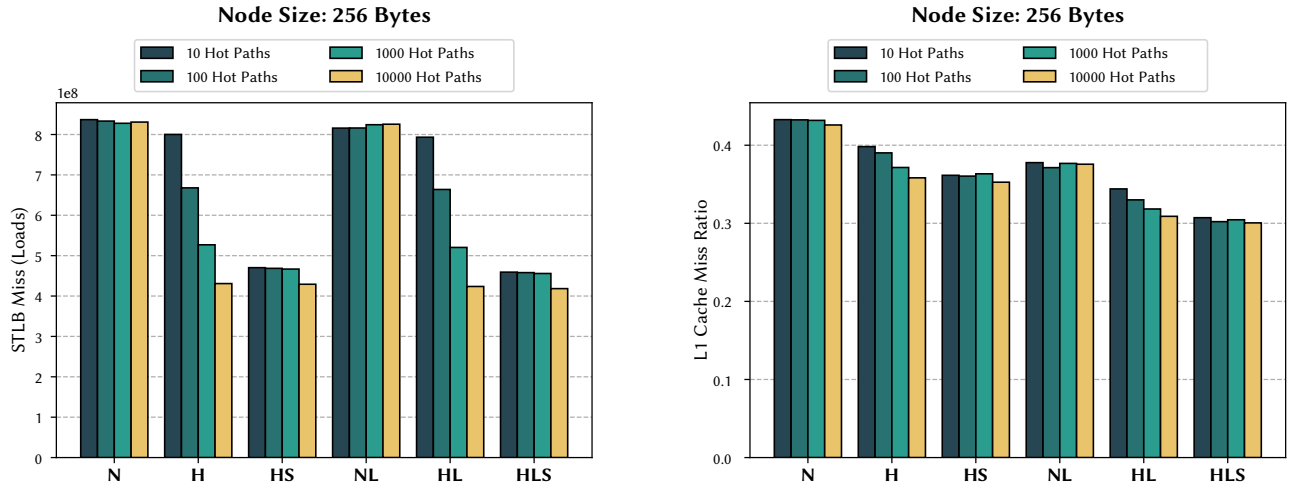
Figure 3 shows the resulting throughput of the five strategies and the native variant in Million operations per second (MOps/s) for up to 40 threads and a node size of 256 bytes and 4096 bytes. We observe that for small node sizes (256 bytes, Figure 3a), the HLS-method performs best. With 40 threads, *HLS* performs ≈ 15% better than the second-placed *HL*-method, which also exploits hot paths and linear search but does not sort the nodes, that do not belong to any hot path. Compared to the strategies without remapping, an improvement from *HLS* of ≈ 21% compared to *NL* and ≈ 50% compared to *N* can be observed.

In all of our scenarios, we also notice that the linear search works very well for node sizes smaller than 1024 bytes. This could be due to the fact that all keys in a node fit just in a few cache lines and only a few very simple comparisons must be executed with linear search. Since a node size of 256 bytes worked very well with our approaches, we also tested node sizes of 128 bytes, but the overall performance decreased noticeably. For node sizes larger than 1024 bytes the linear search becomes much more expensive, as can be seen in Figure 3b, since on average there are now much more comparisons that must be executed compared to a binary search. For node sizes larger than 1024 the native variant also performs best, however, the best *HLS* variant with a node size of 256 bytes has an improvement of ≈ 21% compared to the best performing native variant with a node size of 4096 bytes.

With small node sizes, however, the native (*N*) implementation does not perform as good as with larger node sizes, as illustrated in Figure 3a. The fact that the native method performs so poorly with small node sizes can be explained by the structure of the tree: A tree with small node sizes has a greater depth than a tree whose nodes have a high fanout. Because of the greater depth, more nodes must be accessed during a tree traversal and, unless rearranged, many of these nodes may be located on different pages. This becomes especially a problem when the amount of TLB entries is no longer sufficient and the page table must be accessed. In our experiments, we found that native methods with smaller node-sizes result in more second-level TLB misses, compared to our strategies that use remapping. Figure 4a depicts the the amount of second level TLB misses for *HS* and *HLS*, where hot paths and sorted nodes are used is much lower compared to the strategies where no sorting is applied. This indicates that our remapping strategies indeed improve the TLB usage.

Another interesting aspect is that with a larger number of hot paths (e.g. 10000 hot paths) just minor improvements in throughput (≈ 2%) for the best method *HLS* compared to 10 hot paths are visible. The *HL* strategy, in contrast, shows significant performance improvements and is now nearly as fast as the *HLS* strategy. We also observe that the performance of *H* improves and converges with throughput of *HS*. We attribute this to the large amount of frequently touched nodes that are already remapped because of the hot paths, and therefore the amount of remaining nodes that must be sorted is small. This is also supported by the amount of STLB misses for *H* and *HL* which is drastically reduced for 10000 hot paths in contrast to 10 hot paths. Sorting the remaining nodes in *HLS* and *HS* therefore just leads to a small improvement of about 2% over *HL* and *H* respectively.

For a node size of 256 bytes we can also see an improved miss rate of the L1D cache (fig. 4b) especially for the strategies that uses remapping. It can also be observed that the miss rate of *H* and *HL* converges to *HS* and *HLS* respectively.



(a) Second Level TLB Misses

(b) L1D Cache Miss Ratio

Figure 4: Second Level TLB Misses and L1 Miss ratio of the different strategies with a node size of 256 bytes on 40 threads. If more hot paths are considered, like the throughput, the amount of STLB misses and the L1 miss ratio for *H* and *HL* converges to *HS* and *HLS*

With more than 40 threads, a further improvement in performance can be observed. With 80 threads the *HLS* method can also achieve the highest throughput here with a node size of 256 bytes, the lead over the best *native* variant (with a node size of 4096 bytes) is 26%. We also notice that some of the strategies that make use of remapping have a relatively high variance. We relate this, at least partially, to the allocation of memory for our remapping algorithm. Our remapping algorithm is executed single-threaded and due to the first-touch policy of Linux on NUMA-Systems [9] the memory will be very likely allocated on the NUMA node, where the thread is executed. Therefore, with more than 40 threads, the memory access can become costly. Since building the tree is done in parallel, for more than 40 threads the nodes are allocated on both NUMA regions and therefore a drop in performance is not really noticeable for the non-remapping strategies.

The experimental evaluation of our proposed model shows that data-based cache optimization can contribute to a significant performance gain for read operations in B^+-trees. Especially for small node sizes all remapping strategies show an improvement in performance regarding L1 misses and STLB misses compared to the native strategy. Our best performing approach *HLS* with a node size of 256 bytes shows an improvement of $\approx$ 21% with 40 threads and $\approx$ 26% with 80 threads compared to best performing native strategy with a node size of 4096 bytes.

6 SUMMARY

In this paper, we study the problem of data-based cache-aware B^+-tree layouting. Given a B^+-tree, exceeding the size of the available cache and consequently provoking cache misses during execution, the performance can be optimized by layouting the tree in memory in such a way, that highly probable nodes and paths are favored in the cache and face fewer cache misses. Towards this, we propose a method to collect an adequate probabilistic distribution of the frequency of the usage of nodes and provide an algorithm which identifies so-called hot paths. These paths are consequently placed in linear ascending memory locations to make use of the benefits of sequential data access, like cache prefetching and protection from cache eviction.

We implement a static evaluation setup of our proposed layouting scheme and compare the performance of different optimized implementations on a dedicated test system. Comparing to the native implementation each, the optimizations can achieve an improvement of up to $\approx$ 50% in performance. This improvement is gained when the node size of the tree is chosen rather small, i.e., 256 bytes, and the tree is formed deeper with longer paths. For the native layout, decreasing the node size degrades the performance, hence the best decision for the native layout turns out to rely on large nodes. Comparing the best data-based layout on small nodes with the best native layout on big nodes, we can improve the performance by $\approx$ 26%.

7 FUTURE WORK

Since this paper focuses on investigating if data-based cache-aware layouting of a B^+-tree can improve its performance, the question of a proper collection of access frequencies and the application at runtime still needs to be answered. Hence, we plan to explore different methods for approximate and precise collection of the empirical probability distributions in future work, which provides a trade-off between overhead and quality of the data collection. Consequently, we will explore different approaches to detect a change in this distribution at runtime and trigger a re-layouting of the tree. Lastly, we plan to investigate a wide variety of target applications and workloads to identify under which scenario the data-based layouting can be more beneficial.

REFERENCES

[1] Kuan-Hsun Chen, Chiahui Su, Christian Hakert, Sebastian Buschjäger, Chao-Lin Lee, Jenq-Kuen Lee, Katharina Morik, and Jian-Jia Chen. Efficient realization of decision trees for real-time inference. *ACM transactions on embedded computing systems*, 21(6):1–26, 2022.

[2] Shimin Chen, Phillip B Gibbons, Todd C Mowry, and Gary Valentin. Fractal prefetching b+-trees: Optimizing both cache and disk performance. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 157–168, 2002.

[3] Goetz Graefe. Sorting and indexing with partitioned b-trees. In *CIDR*, volume 3, pages 5–8. Citeseer, 2003.

[4] Goetz Graefe et al. Modern b-tree techniques. *Foundations and Trends® in Databases*, 3(4):203–402, 2011.

[5] Goetz Graefe and P-A Larson. B-tree indexes and cpu caches. In *Proceedings 17th International Conference on Data Engineering*, pages 349–358. IEEE, 2001.

[6] Richard A Hankins and Jignesh M Patel. Effect of node size on the performance of cache-conscious b+-trees. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 283–294, 2003.

[7] Intel. Vtune profiler. https://software.intel.com/vtune/, 2023. [Online; accessed 23-March-2023].

[8] Onur Kocberber, Babak Falsafi, and Boris Grot. Asynchronous memory access chaining. *Proceedings of the VLDB Endowment*, 9(4):252–263, 2015.

[9] Christoph Lameter. Numa (non-uniform memory access): An overview: Numa becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 11(7):40–51, 2013.

[10] Tobin J Lehman and Michael J Carey. A study of index structures for main memory database management systems. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1985.

[11] Viktor Leis, Michael Haubenschild, and Thomas Neumann. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Eng. Bull.*, 42(1):73–84, 2019.

[12] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49. IEEE, 2013.

[13] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The art of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, pages 1–8, 2016.

[14] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.

[15] Linux. perf. https://perf.wiki.kernel.org/, 2023. [Online; accessed 23-March-2023].

[16] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[17] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33:351–385, 1996.

[18] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with coroutines: a practical approach for robust index joins. *Proceedings of the VLDB Endowment*, 11(CONF):230–242, 2017.

[19] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 78–89. Morgan Kaufmann, 1999.

[20] Jun Rao and Kenneth A Ross. Making b+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 475–486, 2000.

[21] Lars-Christian Schulz, David Broneske, and Gunter Saake. An eight-dimensional systematic evaluation of optimized search algorithms on modern processors. *Proceedings of the VLDB Endowment*, 11(11):1550–1562, 2018.

[22] Kefei Wang, Jian Liu, and Feng Chen. Put an elephant into a fridge: optimizing cache efficiency for in-memory key-value stores. *Proceedings of the VLDB Endowment*, 13(9), 2020.

[23] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, pages 473–488, 2018.