Modularizing B+-Trees: Three-Level B+-Trees Work Fine

Shigero Sasaki
s-sasaki@di.jp.nec.com
Cloud System Research Laboratories, NEC
1753 Shimonumabe, Kawasaki, Kanagawa, Japan

Takuya Araki
t-araki@dc.jp.nec.com
Cloud System Research Laboratories, NEC
1753 Shimonumabe, Kawasaki, Kanagawa, Japan

ABSTRACT

The objective of this research is to improve the single-thread performance of a B+-tree in memory. Existing works have been utilized changes in hardware for the performance improvement. While utilizing these works, we modularize B+-trees in memory especially for write-intensive workloads. The modularization is mainly aimed at utilizing the difference in read/write ratio between levels, which is large for write-intensive workloads. In this paper, we show how to modularize B+-trees and effective selections of algorithms and the node size at each level. The cost to switch algorithms is minimized because algorithms at a level are statically defined at compile time. The best selection in this paper formed three-level B+-trees and achieved two- or threefold performance improvement over a typical implementation of a B+-tree. In the three-level B+-trees, we perform linear search on small unsorted leaf nodes, and interpolation search on the large sorted root node, and linear search on small sorted internal nodes.

1. INTRODUCTION

The objective of this research is to improve the single-thread performance of a B+-tree in memory. Single-thread performance still has a huge impact on the overall performance in the recent multi-threaded environments because multi-thread performance will be improved if single-thread performance is improved. We focus on the performance of a B+-tree because it is an index tree that is widely used in data operations. The whole tree may be stored in memory because the amount of memory available has been increasing and may be more than that of disk storage in 2001, for instance, 20GB [4]. Therefore, most of the execution time of a data operation to insert, find, or delete a key tends to be spent in traversing and maintaining B+-trees in memory.

Existing works have been utilized changes in hardware for the performance improvement. A B-tree [5] and its variants such as a B+-tree [12] provide data operations with a minimal number of storage I/Os. However, in-memory index trees such as a T-tree [26] were proposed when most index nodes could be stored in memory. B+-trees optimized for in-memory environments such as a Cache-Sensitive B+-tree [32] were also proposed. Enriched CPU features gave various directions for the optimization of index trees. For example, software prefetch instructions are issued in a prefetching B+-tree [8] before searching within a node. SIMD instructions are utilized by Zhou and Ross [35] for faster traversal of index trees. Large pages can be utilized in a Fast Architecture Sensitive Tree [24] because this tree has three-level blocks including page-level blocks.

We modularize B+-trees in memory especially for write-intensive workloads while utilizing these existing works. Recently, write-intensive data such as sensor data, call log, and billing data, tend to be stored and processed in memory. These write-intensive workloads make the difference in read/write ratio between levels in a B+-tree large due to its balancing mechanism. The modularization is mainly aimed at utilizing the difference, and enable us to select algorithms and the size of nodes at each level of the tree. For example, the modularization enables us to perform linear search on small unsorted leaf nodes and on small sorted internal nodes except the root one and interpolation search on the large sorted root node in a single B+-tree. Leaf nodes different from internal ones were utilized in existing works, for example, by Zhou and Ross [35] and by Chen *et al.* [10], and techniques that optimize B-trees for writes on disk-based databases were surveyed by Graefe [15]. However, the modularization provides a framework to make most of the difference in read/write ratio in memory and a large optimization space on the basis of these works.

In this paper, we show how to modularize B+-trees and effective selections of algorithms and the node size at each level. The cost to switch algorithms may be incurred in the modularized B+-tree. The cost must be minimized because operations on a node or at a level takes very short time if the node is in cache. Therefore, we statically define algorithms at each level at compile time using C++ templates. The modularization preserves the balancing mechanism of a B+-tree and parent-child relationships in a tree. This is because we could not statically select appropriate algorithms to utilize the difference in read/write ratio if the relations dynamically change. A part of the code of the modularized B+-tree is generated by some scripts because the modularization prohibits from using recursion that is often seen in the implementation of an index tree.

The best selection in this paper formed three-level B+-trees and achieved two- or threefold performance improve-

ment over a typical implementation of a B+-tree. We chose Google's C++ B-tree [14] as a typical B+-tree implementation. In the three-level trees, leaf nodes were unsorted so that a new entry can be appended. These nodes were relatively small because linear search, which is an $O(n)$ search algorithm, must be performed on an unsorted node. Interpolation search is an $O(\log \log n)$ search algorithm if the distribution of keys are given [30]. This algorithm was performed on the sorted root node. The node was large to achieve significant performance improvement over binary search which is an $O(\log n)$ algorithm or linear search using SIMD instructions. Even though updating a large sorted node incurs much data copy, the root node is the least frequently updated. Internal nodes between the root node and leaf nodes were small and sorted ones to glue write-optimized leaf nodes and the read-optimized root node together.

The rest of this paper is organized as follows. Section 2 summarizes related work. In Section 3, we describe how to modularize B+-trees and algorithms that are performed in the trees. In Section 4, we explore effective selections of algorithms and the size of nodes, and examine the effect of using the recent hardware such as SIMD instructions. We conclude this paper in Section 5.

2. RELATED WORK

A B-tree [5] and its variants such as a B+-tree [12] can balance naturally even when random keys are inserted or deleted. An index tree intrinsically enables a key in logarithmic time to be found, inserted, and deleted as long as it balances. Therefore, the balancing mechanism of B-trees has often been adopted in subsequent trees. An important parameter of B-trees is the size of nodes. In the five-minute rule [19] [20] [17], the optimal node size with respect to storage I/Os is calculated from parameters such as latencies, transfer rates, and prices of the devices. The typical size of nodes in B-trees is a multiple of that of disk blocks in order to reduce disk I/Os, which were usually the bottleneck around 1970, when the trees were proposed. Small memory suffices to store all internal nodes because the large nodes lead to few internal nodes and many leaf ones. Finding a key usually causes one disk I/O, which is an access to a leaf node. While the parameters of B-trees are often optimized for read performance, Graefe [15] surveyed techniques that optimize B-trees for writes such as optimizing I/Os, buffering insertions, and weakening transaction guarantees primarily for disk-based databases.

As memory became cheaper and larger, we can store most of nodes in memory. The primary goal of a T-tree [26], one of the earliest main memory indexes proposed in the mid-1980's, is to reduce overall computation time while using as little memory as possible. The pointer to a key instead of the actual value of a key is stored in a T-tree to reduce memory space used. Many pointers are stored in a node of a T-tree, which causes fewer rotations to rebalance. Meanwhile, the ratio of the access time in memory to that in CPU cache has been increased. For example, the ratio was 4 on VAX 11/780 in 1980 and 64 on Sun UltraSparc-2 in 1997 [11]. L2 data stalls were a major component of the query execution time in 1999 [1].

The cache line size became an appropriate node size for main memory indexes because the size reduces cache misses. Scattered accesses to lines, which accompany a binary search in a large node, are likely to incur cache misses. Slow memory prompts us to reduce cache misses. The node size of a Cache-Sensitive Search Tree (CSS-tree) [31] and a Cache-Sensitive B+-tree (CSB+-tree) [32], which were proposed around 2000, is basically the same as that of cache lines. Pointer elimination in a CSS-tree and a CSB+-tree contributes to packing more entries in a node and reducing lines to be accessed. Hankins and Patel [21] developed the analytical model of the performance of CSB+-trees and confirmed that the optimal node size for equality search was much larger than the line size, for example, 512B to 32B cache lines. One reason is that small nodes increase TLB misses.

Enriched CPU features gave another direction for the optimization of index trees. A prefetching B+-tree (pB+-tree) [8] is a B+-tree the node size of which is a multiple of the line size. All lines that comprise a node in a pB+-tree are prefetched before searching within the node. This prefetching technique alleviates the performance gap between a cache and memory, and confirms one of the findings by Lee *et al.* [25] that prefetching improves performance if prefetching short array streams which are accessed randomly reduces L1 cache miss. A fractal prefetching B+-tree (fpB+-tree) [9] improved pB+-tree in that it could reduce disk I/Os and cache misses at the same time with its tree-structured nodes. The node size of an fpB+-tree is a multiple of the disk page size, while the size of inner nodes is a multiple of a cache line size. Zhou and Ross [35] utilized SIMD instructions in linear search, binary search, and a hybrid of them for faster traversal of index trees. They also mentioned that unsorted leaf nodes could reduce the amount of data copy. The B+-tree used in the evaluation consisted of nodes of a single type. The size of the nodes was 4KB and hybrid search was performed on them. A pair that was inserted in the tree consisted of a 32-bit floating-point value and a 32-bit pointer.

Key compression is the technique to trade-off the CPU time against the bandwidth and the space. Graefe and Larson [18] surveyed techniques for exploiting CPU caches in the implementation of B-trees and provide a poor man's normalized key, which is the first k bytes of a key that is added to an indirection vector. The vector typically consists of the byte offset and record length of a key. After common prefixes are truncated as in prefix B-trees [6], the normalized key usually works well. Partial keys were used in a pkT-tree and a pkB-tree [7] to minimize cache misses and the cost of comparisons during a tree traversal. The k bits from the dth bit of the whole key are compared in the trees. In a Fast Architecture Sensitive Tree (FAST) [24], the prefix compression scheme of prefix B-trees is extended to obtain order-preserving partial keys while utilizing SIMD instructions. FAST extends other optimization techniques described above; for example, FAST can utilize SIMD blocking besides the two-level blocking in an fpB+-tree.

Some indexing technique for new storage media uses multiple types of nodes. Chen *et al.* [10] researched B+-trees for phase change memory (PCM), which is byte-addressable and non-volatile memory. They made leaf nodes unsorted to minimize expensive write operations in PCM, and showed that performance of delete operations was improved by replacing the entry counter in a node with a bitmap. An FD-tree [27], an index tree for flash memory, consists of a small B+-tree called the head tree and a few levels of sorted arrays. Entries in the head tree are moved to the top arrays

when the head tree overflows.

Interpolation search is on average an $O(\log\log n)$ [30] search algorithm while binary search, a typical search algorithm within a node of a B-tree, is an $O(\log n)$ algorithm. A naive interpolation search, however, implicitly assume the uniform distribution of keys and skew in the distribution deteriorates its performance. Graefe [16] surveyed techniques for interpolation search in B-tree indexes to deal with the skew. These techniques include mixing binary search, which is resilient to skew, into a procedure of interpolation search, detection of skew before the search, and mapping skewed keys to uniformly distributed keys. One of the recent applications of interpolation search is the read-only storage engine for LinkedIn's Voldemort [33]. In this engine, keys were a fairly representative uniform distribution because they are based on MD5.

B+-trees are often used in key-value stores, which are indispensable for dealing with large data. Atikoglu *et al.* [3] analyzed five workloads from Facebook's Memcached deployment. One of their findings showed that different applications of Memcached could have extreme variations in terms of read/write mix, request sizes and rates, and usage patterns. As also mentioned in this paper [3], one workload used in the evaluation of SILT [28] had the ratio of read and write of 9:1 for 20B keys and 100B values. Another workload has the ratio of 1:1 for 64B pairs. The entry size in the evaluation of FAWN [2] was 256B or 1024B. Even though workload characteristics differ from application to application, existing works give good guidelines to evaluate the performance of a B+-tree.

3. MODULARIZING B+-TREES

In this section, we describe how to modularize B+-trees. Section 3.1 provides an overview of the modularization. Section 3.2 describes the implementation of the modularized B+-tree, which enables us to switch algorithms at each level at low cost. Section 3.3 explains the algorithms which can be selected and performed on a node in this paper.

3.1 Overview

The read/write ratio on a node is lower than that on its parent in a B+-tree because of its balancing mechanism. Insertion and deletion in a B+-tree leave it balanced without any balancing behavior such as rotations. When a given entry, which is a pair of a key and an identifier, is inserted into a B+-tree, an appropriate leaf node is found. The leaf node is split into two before the insertion of the entry if it overflows, and the entry that corresponds to the newly created leaf node is inserted into its parent. The node splitting will occur along the search path while maintaining the balance of the tree. Therefore, an entry is inserted into an internal node in a B+-tree only when its child node overflows, and the read/write ratio on nodes at a level is higher than that at a lower level.

The difference in read/write ratio can be utilized by write-optimized nodes only at a low level. Sorted nodes are an example of read-optimized ones while unsorted nodes are write-optimized ones as implied by Zhou and Ross [35]. The algorithm to insert an entry to an unsorted node simply appends the entry as the last entry on the node. However, that to a sorted node must find the place where the new entry is inserted and move the entries larger than the new entry before the new key is written. Suppose that we make leaf nodes unsorted and internal ones sorted. We can insert entries efficiently to leaf nodes, which are written more frequently and read less frequently than internal nodes are, while finding entries efficiently on sorted internal nodes.

The difference can also be utilized by read-optimized nodes only at a high level. For example, interpolation search, of which complexity is $O(\log\log n)$ [30], can be performed at a high level instead of binary search of which complexity is $O(\log n)$. Interpolation search, as long as the keys are uniformly distributed, can find an entry more efficiently than binary search on a large node, even though insertion to a large node incurs much data copy. Consequently, utilizing multiple types of nodes in a single tree can improve the performance of put operations while minimizing the deterioration of the performance of get operations, and vice versa.

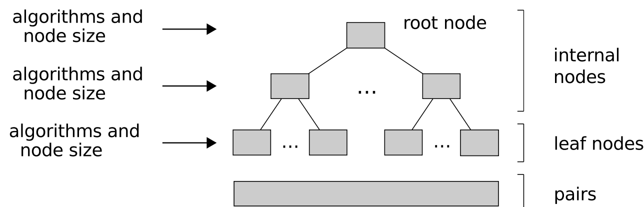


Figure 1: Modularizing a B+-tree to select algorithms and the node size at each level

The modularization of B+-trees is aimed at enabling us to select algorithms and the size of nodes at each level, as shown in Figure 1. We regard nodes at a level as a module, and algorithms and the node size are defined to them. The modularization enables us to utilize both write-optimized and read-optimized nodes in a single tree. Pairs of keys and values are also regarded as a module even though they are not changed in this paper.

The node size as well as the algorithms is an important factor that determines the trade-off between performance of get and put operations. We note that the performance of delete operations was between that of get and put ones as shown in Section 4. Insertion of an entry to a small sorted node incurs the small amount of data copy, and the insertion is performed rather efficiently. We cannot, however, find an entry efficiently when the node size is too small as shown by Hankins and Patel [21]. On an unsorted node, we can improve performance of put operations by enlarging the node because it reduces the overhead of node splitting, even though linear search on a large node is inefficient. When there are many entries on a large sorted node, interpolation search outperforms binary search because of the complexity. LinkedIn's Voldemort gives the functionality to perform interpolation search for read-only stores [33]. However, insertion to a large sorted node incurs much data copy to keep its keys sorted. The node size has been considered from the point of view of the number of data blocks, such as disk blocks and cache lines, accessed. However, we put more emphasis on the trade-off than ever.

3.2 Implementation for the Modularization

The modularization enables us to switch algorithms at each level and therefore the switching cost must be minimized. As described in Section 3.1, the trade-off between performance of get and put operations heavily depends on algorithms and the node size. The difference in node size

incurs no overhead. Switching algorithms, however, may cause an additional processing such as virtual function table lookups. Although the switching cost is typically very small, this cost matters when operations within a node or at a level takes very little time.

We statically define algorithms at each level at compile time using C++ templates to minimize the switching cost. Specifically, a C++ idiom called the *curiously recurring template pattern* is used in the implementation. The static polymorphism requires to preserve parent-child relationships in a tree, as the balancing mechanism of a B+-tree does. If the relationship dynamically changes as rotations to balance a tree do, the read/write ratio on a node and algorithms that are appropriate to the ratio also changes dynamically. The change leads to a dynamic selection of algorithms and incurs the run-time overhead. The static polymorphism imposes another restriction on using recursive function calls. As a result, a part of the code of the modularized B+-tree is generated by some scripts in our implementation.

The five methods below are defined on nodes at each level as in a typical implementation of a B+-tree. An implementation of a method on a leaf node and on an internal one are not exactly the same. For example, a split key still exists on a leaf node even though the key is removed on an internal one. Mechanisms to deal with the differences are implemented outside a node, and therefore we can focus on the core part of the methods.

find method finds an entry on the node, the key of which is the largest key that is less than or equal to a given search key, and returns the corresponding identifier.

insert method inserts a given entry to the appropriate position on the node or a new split node and returns whether the node is split.

delete method deletes an entry that has the same key as a given key.

split method chooses and returns the split key, which is usually the median, and move the keys that is larger than or equal to the split key.

merge method merges an empty node with an adjacent node.

In order to statically define the algorithms at each level, we preserve the balancing mechanism of a B+-tree while algorithms can be different level by level. Appropriate algorithms must be statically given to utilize the difference in read/write ratio at the minimum switching cost. It follows that parent-child relationships in a tree must be statically defined among nodes, and balancing mechanisms such as rotations are unsuitable for this modularization. As a consequence, we define write-optimized algorithms at the level 0 and more read-optimized algorithm at the level $(i + 1)$ than at the level i. Note that nodes at the level 0 are called leaf nodes and the level $(i + 1)$ is higher than the level i in this paper.

The pseudo code for tree traversals when the maximum height is 3 is described in Algorithm 1. The modularization prohibits from using recursion or loop that is often seen in the implementation of an index tree because the methods of different types of nodes are called at each level in a static way. As a consequence, we must calculate the maximum

Input: $height, root$
Output: id
$id = root$;
switch $height$ **do**
  case *3*
    $id = level_3.find(key, id)$;
  case *2*
    $id = level_2.find(key, id)$;
  case *1*
    $id = level_1.find(key, id)$;
  case *0*
    $id = level_0.find(key, id)$;
end

Algorithm 1: Tree traversal when the max height is 3

height and write out all the traversal process before compile time as in this algorithm. The maximum height is calculated from the number of entries in a node. The number depends on the size of a key and an identifier, and the size and the data structure of a node. This structure, which implies whether a node is sorted or unsorted within the scope of this paper, is determined by the insert method. The actual code that corresponds to this traversal algorithm is generated by some ruby scripts. In the algorithm, $level_i$ denotes nodes at the level i and $find(key, id)$ represents the find method that finds the largest key that is less than or equals to key on the node id at level i.

Input: $key, id, path[], entry[]$
if $height < 1$ **then**
  $height + +$;
  $path[1] = root = level_1.new_root()$;
  $entry[1] = 0$;
end
$ret = level_1.insert(path[1], entry[1], key, id)$;
if $ret < kSplit$ **then**
  $return$;
end
if $height < 2$ **then**
  $height + +$;
  $path[2] = root = level_2.new_root()$;
  $entry[2] = 0$;
end
$ret = level_2.insert(path[2], entry[2], key, id)$;

Algorithm 2: Insertion of an entry to the parent node

Algorithm 2 is the pseudo code for inserting the pair of the split key key and the identifier of a new leaf node id into its parent node when the maximum height is 2. A search path was preserved in $path[]$ and $entry[]$ when we find the leaf node to insert a new entry. $insert(path[i], entry[i], key, id)$ inserts the pair of key and id into the node $path[i]$ as the $(entry[i])$th entry at the level i. When a node on a search path is split, key and id is overwritten by the split key and the identifier of a new node, respectively. If no split occurs, the algorithm is ended. We omit the algorithm for deleting an entry for simplicity, however, it also ascend a tree.

3.3 Algorithms on a Node

This subsection explains the five interface methods in Section 3.2. Two types of the insert, delete, and split method

was implemented for sorted and unsorted nodes. A single type of the merge method was implemented because the method was called only when there is only one identifier on an internal node was deleted. As the find method, we implemented various search algorithms, which are linear search, binary search, and interpolation search with and without using Intel Streaming SIMD Extensions (SSE) or Advanced Vector Extensions (AVX).

The insert method for sorted nodes inserts a new entry into an appropriate place. The place is found by performing the find method and then data copy is incurred to make room for the new key and to keep keys in the node sorted. The split method for sorted nodes chooses the $((n+1)/2)$th key as the split key and moves the $((n+1)/2)$th to $(n-1)$th entry on an overflowing node to a new node where n denotes the maximum number of entries on the node. We can append a new entry to an unsorted node without copying data even though linear search must be performed to find a key on the node. The split method for unsorted nodes move the keys which are greater than or equal to the split key, which is the median of the last three keys. All the keys on an overflowing node are compared with the split key in the method since the keys are unsorted. Approximately, the utilization of unsorted nodes that we measured was 60% while the utilization of sorted nodes was 70% as previously studied in existing works [34] [23].

The delete method for sorted nodes incurs data copy while the method for unsorted nodes does not. The larger keys than the deleted key must be moved to keep the order on a sorted node. On an unsorted node, it is sufficient to move the last key to the place where a key is deleted. The merge method is called when there is only one identifier on an internal node. The pair of the last identifier and the corresponding key is appended to the left adjacent node that has the same parent node. If the node is the leftmost one, the last identifier is inserted into the right adjacent node. When one of the last two child nodes is merged, the parent node is also merged to a sibling node.

We implemented some types of linear search to perform in the find method. The simplest type examines keys from the first to the last on a sorted node until the key examined exceeds the search key. Linear search using SIMD instructions can find a key faster than the simple one. Eight float keys which are aligned to 32-byte boundary can be examined at once using AVX in the linear search while four keys aligned to 16-byte boundary can be examined at once using SSE. We can count the number of keys that are less than or equal to the search key without any conditional branch and find the largest key that is less than or equal to the search key as described by Zhou and Ross [35]. However, the counting approach was slower than the examining approach in our environment because of the size of nodes. For equality search on an unsorted node, we implemented a type of linear search which examines keys until the examined key equals to the search key. This type of linear search, as described by Zhou and Ross [35], efficiently finds a unique key on a leaf node. Consequently, we implemented six types of linear search that examine keys, three on a sorted node and three on an unsorted node.

Three types of binary search are used in this paper. The first type is the normal binary search, the second one is the hybrid binary search using SSE, which was proposed by Zhou and Ross [35], and the third one is hybrid binary search using AVX. In a typical B+-tree, the size of a node is large, for example, 4KB which is equal to the size of a disk block. A large node reduced the number of disk I/Os if most of the nodes are on a disk storage as around 1970 when B-trees were proposed. However, we assume that binary search is performed on a small node, the size of which is multiple of the cache line size, because binary search on a large node accesses many cache lines. In addition, small sorted nodes reduce data copy when a new entry is inserted.

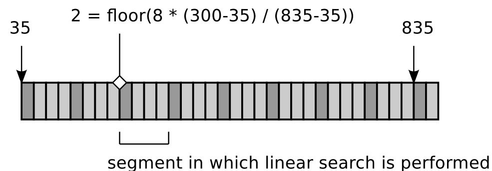


Figure 2: Hybrid interpolation search segments keys

We embedded linear search in interpolation search as Zhou and Ross [35] did regarding binary search. The hybrid interpolation search segments keys in a sorted node and perform interpolation search for the first key in each segment. After searching a segment, linear search is performed in the segment. Figure 2 shows a behavior of hybrid interpolation search. A rectangle in the figure denotes a key on a node, and a dark gray rectangle represents the first key in a segment. A segment consists of four keys, and there are 8 segments and 2 keys on a node in the figure. The smallest key is 35, and the second largest key, the smallest key that is larger than any key in segments, is 835. Suppose that the search key is 300. We first perform interpolation search and find that linear search must be performed in the second segment since the floor of the product of the number of segments and the ratio of $(300-35)$ and $(835-35)$. If the search key is in the second segments, the key is found by linear search. If all keys in the segment are larger than the search key, we perform hybrid interpolation search again, excluding the zeroth, first, second segment. In both hybrid binary search and hybrid interpolation search, the highest search performance was achieved when the size of segment equals to eight keys or 32 bytes in our environment as described in the following section.

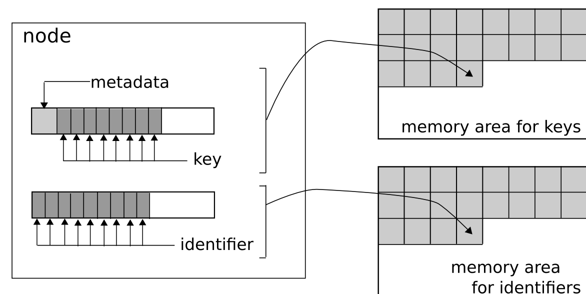


Figure 3: node is stored in two distinct areas

Figure 3 shows the memory allocation for nodes. A node consists of metadata, keys, and identifiers. Metadata and keys are stored in a memory area for keys and identifiers are stored that for them. The size of metadata is equal to that of a SIMD register to align keys, which is 16 bytes for SSE

and 32 bytes for AVX. Metadata contains the counter of the number of entries on the node.

A node is stored in two distinct memory areas because the keys and identifiers have different access patterns. These two memory areas are reserved for leaf nodes and internal nodes. This is because they are really different in read/write ratio and total size. In the identifier area for leaf nodes, 64-bit identifiers are stored and 32-bit identifiers are stored in the identifier area for internal nodes. The pairs are stored in their memory area, the size of which may need 64-bit identifiers. The size of a node is defined as the sum of the size of metadata and the product of the size of a key and the maximum number of keys on the node.

4. SELECTIONS OF ALGORITHMS AND THE NODE SIZE

This section investigates selections of algorithms and the node size at each level through performance evaluation. Section 4.1 describes the evaluation environment. Section 4.2 measures the baseline performance of our prototype of B+-trees that are modularized and a typical implementation of a B+-tree, Google's C++ B-tree [14]. Section 4.3 shows throughput in the prototype that consisted of nodes of a single type. Section 4.4 and 4.5 evaluate throughput in the prototype that consists of nodes of two and three types, respectively. Section 4.6 investigates increase in throughput when TLB misses were reduced by allocating large pages and when the penalty of cache misses was alleviated by software prefetch.

4.1 Evaluation environment

The evaluation was carried out on a computer that had a 2.50GHz quad-core Intel Core i5-2400S. The CPU supports SSE4.1/4.2 and AVX, and has a 32KB instruction and 32KB data first level cache for each core, a 256KB instruction/data second level cache for each core, and a 6MB instruction/data third level cache, shared among all core. The CPU is connected to 8GB dual channel DDR3-1333 SDRAM, the bandwidth of which is 21 GB/second. The computer runs Ubuntu 12.04, and programs used in the evaluations were compiled by g++ 4.6.3.

In the evaluation, we measured throughput of put, get, and delete operations. We put 12 million pairs and then measured the time taken to put 4 million pairs. The number of pairs was determined from the maximum amount of memory cache that Berkeley DB (11.2.5.3.21) can use, 2GB. We executed get operations for keys in the last 4 million of the 16 million pairs and delete operations to keys in the first 4 million of the 16 million pairs. We implicitly assumed that the recent keys were accessed frequently and that old pairs were more likely to be deleted than a new key.

Unless otherwise noted, the size of a pair is 32 bytes, the type of keys is float, and the size of value is 28 bytes. A small key size can be a practical assumption because key compression techniques such as partial keys [7] make a large or variable-length key a small one in an index tree. Float keys are less common and require less simple operations than integer keys. However, we used float keys as in [35] to take into SIMD width into consideration because AVX does not support integer operations. The size of values is based on that of vector in the C++ Standard Template Library, 24 bytes. As Atikoglu *et al.* [3] analyzed, workloads varies from system to system. However, we regard the 32-byte pairs suitable as a starting point of performance evaluation.

4.2 Baseline performance

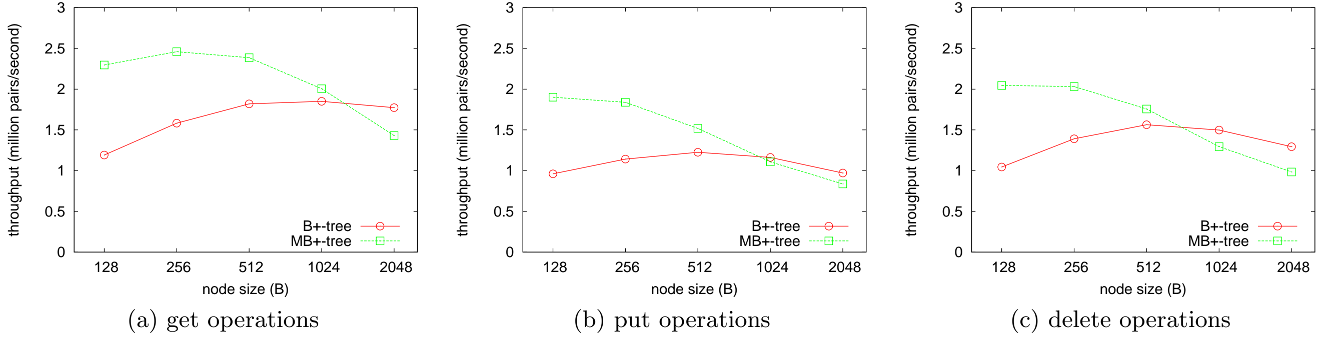
We measured throughput in our prototype of a B+-tree that was modularized and a typical implementation of a B+-tree, Google's C++ B-tree [14], in this subsection. We hereafter call the prototype a Modular B+-tree (MB+-tree). The behavior as a B+-tree in an MB+-tree was prototyped in accordance with the textbook by Garcia *et al.* [13] and Intel's optimization manual [22]. Google's B-tree performs linear search on a sorted node if the type of a key is an integral or a floating-point type. The default size of all nodes is 256 bytes, and can be changed by specifying a template parameter.

Even though Berkeley DB uses B+-trees as an access method, we prefer to compare MB+-trees with Google's B-tree. Berkeley DB provides rich functionality and designed for disk-base systems unlike MB+-trees and Google's B-tree. And throughput of put, get, and delete operations of Berkeley DB were 0.188, 0.466, and 0.129 million pairs/second, which are much smaller than those in Google's B-tree. These were measured after we configured Berkeley DB to keep data entirely in memory as stated in [29].

Figure 4(a), 4(b), and 4(c) show throughput of get, put, and delete operations in the typical B+-tree and an MB+-tree with varying node sizes from 128 bytes to 2048 bytes, respectively. The MB+-tree was composed of a single type of nodes on which linear search was performed, and the size of which is the same. Therefore, the difference between the MB+-tree and the B+-tree is the place where values are stored. The MB+-tree stores the pair of a key and a value in the memory area for pairs while the typical B+-tree stores a pair in a leaf node. It follows that the MB+-tree can store more entries in a node and incur less copy than the B+-tree. The reason that the smallest node size was 128 bytes is that the size of a pair was basically 32 bytes.

Throughput of get operations in the trees depends on the node size because the execution time of linear search is basically in proportion to the number of keys. The highest throughput of get operations was 1.851 on 1024-byte nodes in the B+-tree and 2.460 on 256-byte nodes in the MB+-tree. This indicates that the difference in place where the pairs were stored affected throughput of get operations. The optimal node size is larger than the cache line size even though nodes, the size of which equals that of a cache line, minimize the number of cache lines. This is because TLB misses as well as cache misses decrease throughput of get operations as investigated by Hankins and Patel [21]. The height of an index is high if the size of the nodes is small, and accessing a different node may incur a TLB miss.

Throughput of put operations was the highest on smaller nodes than those on which the highest throughput of get operations was achieved. Data copy incurred by insertion on a sorted node decreases as the node size becomes smaller. Therefore, the optimal node size for put operations was smaller than that for get operations. The highest throughput of put operations was 1.225 million pairs/second on 512-byte nodes in the B+-tree and 1.901 million pairs/second on 128-byte nodes in the MB+-tree. Throughput of delete operations in Figure 4(c) was between that of get and put operations. This is because a delete operation incurs data copy, which is not incurred in a get operation, but does not



(a) get operations (b) put operations (c) delete operations

Figure 4: Throughput in the typical B+-tree and an MB+-tree

split a node as in a put operation. The highest throughput of delete operations was 1.564 million pairs/second on 512-byte nodes in the B+-tree and 2.046 million pairs/second on 128-byte nodes in the MB+-tree.

The total throughput of get, put, and delete operations depends on the ratio of these three operations. We can see in Figure 4(a)–(c), however, that the MB+-tree that consisted of 256-byte nodes and the B+-tree that consisted of 512-byte nodes achieved a good performance in total. Throughput of get, put, and delete operations in the MB+-tree was 35.2%, 50.1%, and 30.0% higher than that in the B+-tree.

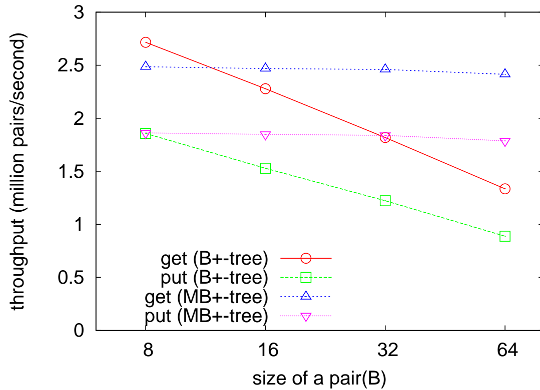


Figure 5: The size of pairs affected very little on throughput of the MB+-tree

Figure 5 shows that throughput in the MB+-tree does not heavily depend on the size of a pair while that of the typical B+-tree does. Throughput when the pair size was 8, 16, 32, and 64 was measured in the figure. A pair consists a 4-byte float key and a 4-, 12-, 28-, or 60-byte value, and therefore a larger value increased only the amount of sequential copy in the memory area for pairs in the MB+-tree. The increase in size of values changes nothing of operations in a MB+-tree. However, the number of entries in a node decreases and the amount of data copy on a node increases in the B+-tree as the size of value becomes larger. Throughput of put and get operations is shown in the figure and that of delete operations, which was between that of get and put operations, is omitted to simplify the graph.

Throughput in an MB+-tree depends on the size of a key more than the type of a key as shown in Figure 6. Throughput was measured when a key was integer, float, long long,

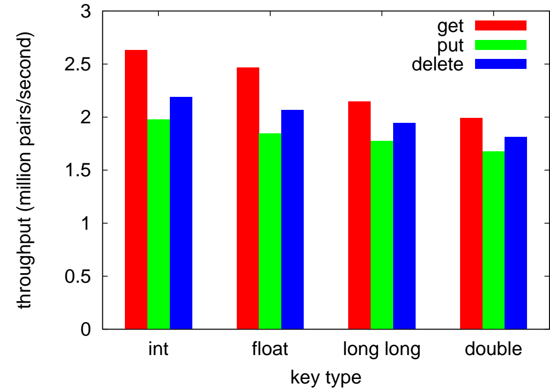


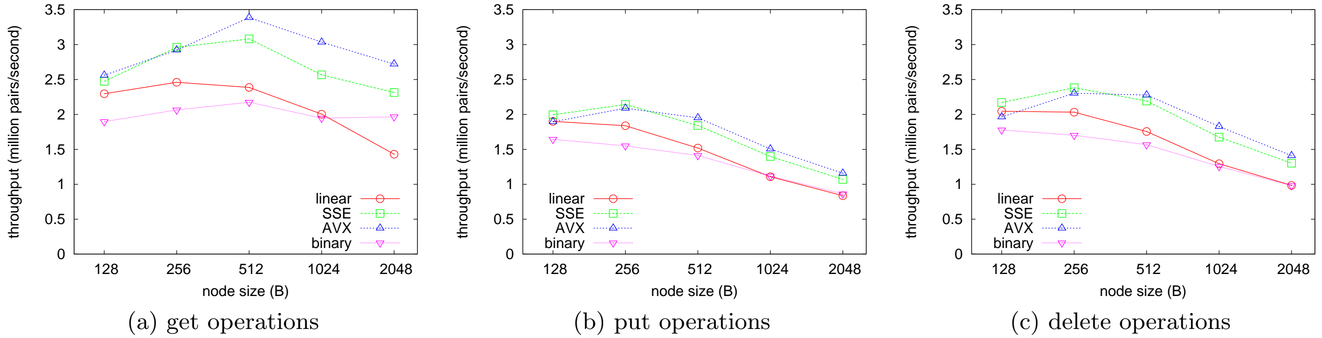
Figure 6: Influence of key types on throughput was limited

double. The type of key, an integer and a float key are different types even though the size of them is the same, did not significantly affect throughput on the recent CPU. The size of key, however, had an impact on throughput because the size changed the number of keys on a node and in a cache line. Performance evaluation in the following section was conducted using float keys because we can assume small keys as described above, and because the difference in throughput when we used integer and float keys was small.

4.3 Throughput on nodes of a single type

We evaluated throughput of MB+-trees that consisted of nodes of a single type in this subsection. Throughput when the node size is between 128 and 2048 bytes was measured. An MB+-tree was composed of nodes that performed linear search, linear search using SSE, linear search using AVX, or binary search. The evaluation results indicated that performing linear search using AVX on 512-byte sorted nodes was one of the best selection within the scope of this subsection. In the previous subsection, we show that the MB+-tree that consisted of nodes that performed linear search achieved the highest throughput when the node size was 256 bytes. The MB+-tree using AVX achieved 37.7% and 6.2% higher throughput of get and put operations than those in the MB+-tree that did not use AVX, respectively.

Figure 7(a) shows that throughput of get operations was the highest when linear search using AVX was performed on a 512-byte sorted node. A get operation calls only the find



(a) get operations (b) put operations (c) delete operations

Figure 7: Throughput when linear or binary search was performed on all nodes

method on a node while put and delete operations call other methods. Therefore, throughput of get operations indicates the performance of search algorithm used in the find method. The measurement revealed that linear search using AVX was the best algorithm of four for read-only workloads. Linear search using SSE outperformed that using AVX on 128- and 256-byte nodes because of the size of metadata, which was equal to that of a SIMD register. The size of metadata when AVX was used was 32 bytes to align keys to 32-byte boundary, and that when SSE was used was 16 bytes. Large metadata decreased throughput on a small node even though the decrease was negligible on a large node.

Throughput when binary search was performed in an MB+-tree was lower than that when linear search using SIMD instructions was performed if the node size ranged from 128 to 2048 bytes. Linear search performs faster than binary search on a small node as shown by Zhou and Ross [35]. Binary search on a 1024- and 2048-byte node was slower than that on a 512-byte node because more cache lines were accessed on a larger node.

Figure 7(b) shows that throughput of put operations and the highest throughput was achieved when linear search using SSE was performed on 256-byte nodes. As shown in the previous figure, linear search on a 512-byte node using AVX was faster than that using SSE. However, an increase in size of nodes decreases throughput of put operations. This is because the insert method incurs much data copy especially on leaf nodes. As a result, the highest throughput of put operations was achieved on 256-byte nodes and decreased on larger nodes. Throughput of delete operations was still between that of get and put operations as shown in Figure 7(c).

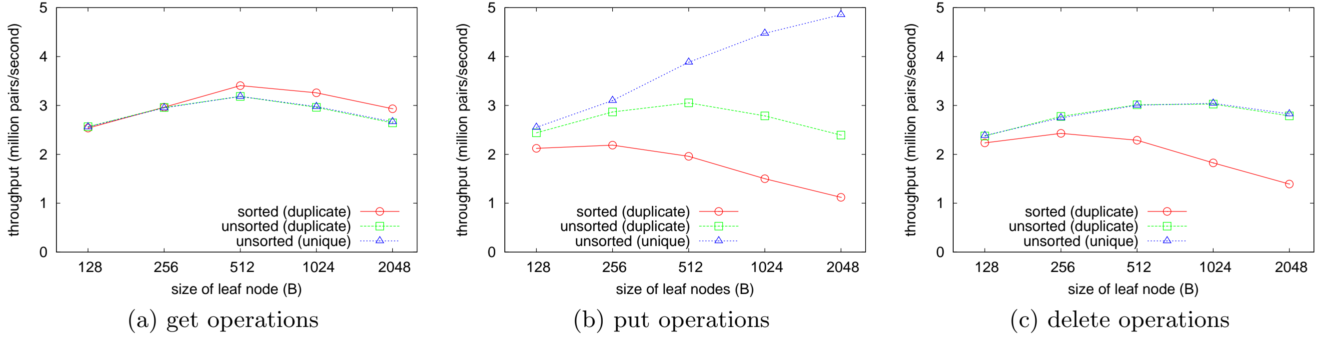
4.4 Throughput on nodes of two types

Here, we evaluated throughput of MB+-trees that were composed of nodes of two types. The read/write ratio on a leaf node is much larger than that on an internal node. For example, a 512-byte node, which can contain 60–120 float keys, is split for every 60 inserts. This means that the ratio on an internal node at the level 1 is 1/60 of that on a leaf node. Therefore, we make leaf nodes different from internal ones to utilize the difference in ratio. We measured throughput with varying sizes of leaf nodes even though the size of internal nodes was fixed to 512 bytes, on which the highest throughput of get operations was achieved as shown in Figure 7(a).

We measured throughput when leaf nodes were sorted and unsorted. In the case that duplicate keys were allowed to be inserted, we need to examine whether there exists the same key. This examination incurs little overhead on a sorted node because the existing key that can be the same as the new key is determined when the place to insert is found. However, linear search needs to be performed on an unsorted node before insertion. If only unique keys are allowed to be inserted, put operations on unsorted nodes can be performed efficiently because the new key can be appended and no search activity is required [35]. We therefore measured throughput on unsorted leaf nodes when duplicate and unique keys are inserted.

Figure 8(a) shows that throughput of get operations on sorted leaf nodes was higher than that on unsorted leaf ones, and that throughput on unsorted leaf nodes for duplicate keys and for unique keys were almost the same. Linear search on a sorted leaf node and on an unsorted leaf one accesses a half of the keys on a node on average. On a sorted node, the first key that is larger than the search key is found. On an unsorted node, the first key that is the same as the search key is found. Therefore, a half of keys is examined on average on both of nodes. However, utilization of unsorted nodes is smaller than that of sorted nodes because the median of the last three keys, not all keys, is chosen as the split key on an unsorted node. The split key will unevenly split the unsorted node, and the nodes are poorly utilized. The difference in utilization makes the difference in throughput. The behavior of linear search on an unsorted node did not depend on whether duplicate or unique keys were searched. Hence, throughput of get operations for duplicate and unique keys were almost the same.

In Figure 8(b), unsorted nodes improved throughput of put operations, especially for unique keys. When leaf nodes were sorted, we could improve throughput by making leaf nodes smaller than internal nodes. Smaller leaf nodes decrease the amount of data copy when a new key is inserted even though the find method may work inefficiently as shown in Figure 8(a). The highest throughput of put operations was achieved when the size of leaf nodes was 256 bytes and was 2.188 million pairs/second, which was 11.4% higher than throughput when all nodes were 512 bytes. Unsorted leaf nodes significantly improved throughput of put operations. Throughput of put operations for duplicate keys when we used 512-byte unsorted leaf nodes was 3.053 million pairs/second, which was 39.5% higher than the highest throughput when leaf nodes were sorted. The improvement was brought by eliminating the overhead of data copy on



(a) get operations (b) put operations (c) delete operations

Figure 8: Throughput when leaf nodes were sorted and unsorted

leaf nodes. When unique keys were inserted, throughput of put operations increased as the size of leaf nodes became larger. This is because no search activity was required and because larger nodes were split less frequently.

Throughput of delete operations for duplicate keys and for unique keys when leaf nodes were unsorted were almost the same as shown in Figure 8(c). As in get operations, the behavior of linear search on an unsorted node did not depend on whether duplicate or unique keys were searched. The last key and identifier were moved to the place where the deleted key and identifier were when leaf nodes were unsorted. Throughput of delete operations when leaf nodes were sorted was lower than that when leaf ones were unsorted because the keys larger than the deleted key and the corresponding identifiers must be moved in a delete operation on sorted nodes. Consequently, we do not show throughput of delete operations as well as get ones for duplicate keys and for unique keys individually.

For unique keys, we could achieve higher throughput of put operations as keys became larger. The rate of increase in throughput with node size was higher than the rate of decrease in throughput of get and delete operations. It follows that we should select the node size on the basis of the ratio of operations. For example, throughput of get and put operations were 3.185 and 3.884 million pairs/second on 512-byte leaf nodes, and 2.666 and 4.857 million pairs/second on 2048-byte nodes, respectively. Therefore, we should select 2048-byte leaf nodes when put operations account for more than 65.3% of all operations, for example, when the ratio of get and put operations is one to two.

4.5 Throughput on nodes of three types

We here evaluate throughput of MB+-trees that were composed of nodes of three types. In this subsection, we show throughput when interpolation search on sorted nodes at high levels, namely on the root node, was performed. Interpolation search is an $O(\log \log n)$ algorithm and provides a faster find method than an $O(\log n)$ algorithm does as n or the node size becomes larger. However, a larger sorted node increases the amount of data copy when a new key is inserted. We therefore performed interpolation search only at high levels, on which the find method is mainly called.

Figure 9 shows throughput when hybrid interpolation search or hybrid binary search was performed on the root node at the level 2. Linear search in both of the hybrid algorithms used AVX and did not use SSE since that using AVX performed faster than that using SSE. The number of keys in a

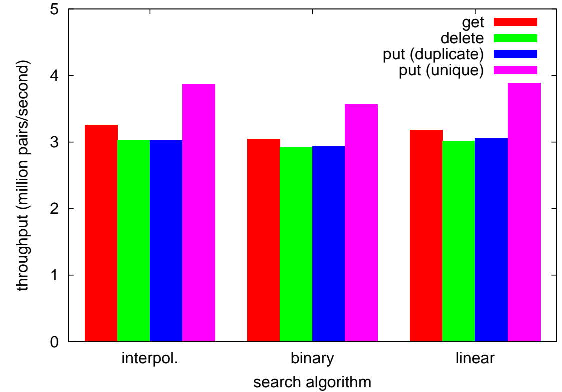


Figure 9: Interpolation search on the root node improved throughput of get operations

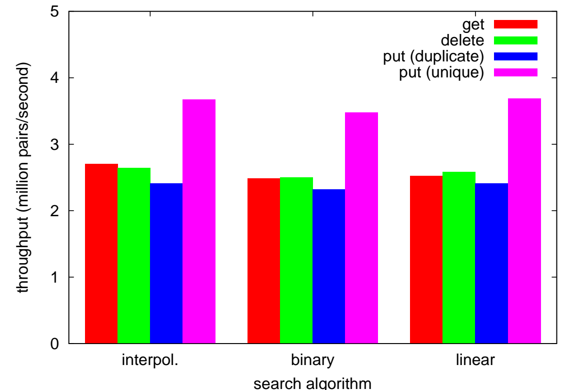


Figure 10: Throughput when the number of entries was doubled

segment in the hybrid algorithms was eight. This is because eight keys can be compared at once using AVX and because segments that contain more than eight keys decreased throughput. However, we omit the details of the evaluation of SIMD instructions or the segment size for simplicity. Inserting 16 million pairs to the MB+-tree generated 2017 entries at the level 2. We made the root node at the level 2 large enough to store all of them. The large root node was aimed at performance improvement by performing the

$O(\log \log n)$ search algorithm. For comparison purposes, we also show throughput when linear search using AVX was performed on all internal nodes, the size of which is 512 bytes, in the figure. Through these measurements, nodes at the level 1 were 512-byte sorted ones and leaf nodes were 512-byte unsorted ones, on both of which linear search using AVX was performed. The nodes achieved a good performance in total as shown in Figure 8(a) and Figure 8(b).

Figure 9 indicates that the MB+-tree that performed hybrid interpolation search on the root node achieved higher throughput than that performed hybrid binary search on the root node and that performed linear search on all nodes. The difference between throughput in the MB+-tree that performed interpolation search and that performed linear search was small, however, the difference become larger as the number of entries increase. Figure 10 shows throughput when the number of entries was doubled. Even though the doubled number of entries decreased throughput, the difference in throughput when interpolation search was performed and that when linear search was performed became larger.

A two-level B+-tree, which had the very large root node on which hybrid interpolation search was performed, brought poor throughput of put operations, which was 0.422 million pairs/second since the root node at the level 1 contained 169836 entries. However, throughput of get operations was 3.526 million pairs/second, which was higher than that in any three-level B+-tree evaluated so far. Consequently, we can improve throughput of get performance more in the three-level MB+-tree as the number of keys becomes larger with minimizing the deterioration of throughput of put operations as long as the keys are uniformly distributed. When keys were distributed logarithmically, throughput of get and put operations were sharply dropped to 0.393 and 0.447 million pairs/second, respectively, Because the bit patterns of random integers were distributed logarithmically as float keys, we measured the throughput by comparing random integers as float keys.

4.6 Using large pages and software prefetch

TLB misses and cache misses affect the time taken to traverse an index tree as modeled by Hankins and Patel [21]. Many recent CPUs have TLBs for large pages, and the functionality to allocate large pages such as Linux hugepages is provided. The size of a large page in this environment was 2MB while that of a normal page was 4KB. We therefore allocated large pages to reduce TLB misses in the three-level MB+-tree that utilized interpolation search in Section 4.5.

Figure 11 shows that we could increase throughput by allocating large pages. We could choose the type of pages for internal and leaf nodes since the nodes had its own memory area. Allocation of large pages to internal nodes increased throughput of get operations by 2.8%, and that to both internal and leaf nodes increased throughput by 23.2%. It was reasonable that accesses to leaf nodes were more likely to cause TLB misses than accesses to internal nodes because the amount of memory consumed by leaf nodes was much more than that by internal nodes. We note that keys on internal nodes, identifiers on internal nodes, keys on leaf nodes, and identifiers on leaf nodes were allocated 2, 2, 69, and 137 large pages, respectively.

Large pages increased throughput of put operations as well as that of get operations. Allocation of large pages to internal nodes increased throughput by 3.2% for duplicate

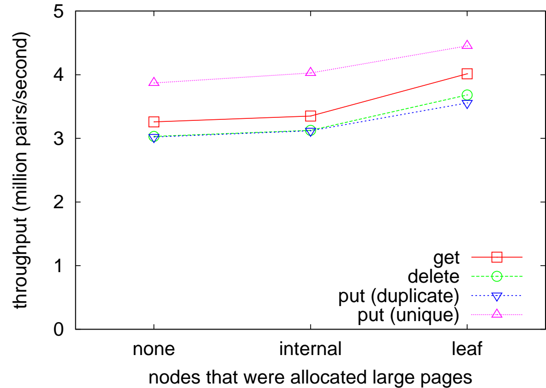


Figure 11: Allocation of large pages to nodes increased throughput

keys and by 4.1% for unique keys. Allocation of large pages to all nodes increased throughput by 17.7% for duplicate keys and by 15.1% for unique keys. Throughput of delete operations was slightly higher than that of put operations for duplicate keys. These evaluation results imply that allocation of large pages only to internal nodes was efficient. We can see in Figure 11 that 4 large pages for internal nodes increased throughput about 3% even though 206 large pages for leaf nodes increased throughput about 20%.

We counted two hardware events for DTLB using Intel VTune Amplifier XE 2011 in the same settings as in Figure 11. One event is *DTLB_LOAD_MISSES.STLB_HIT*, which counts the number of DTLB first level load misses that hit in the second level DTLB (STLB). Another is *DTLB_LOAD_MISSES. MISS_CAUSES_A_WALK*, which counts the number of STLB misses. The number of DTLB misses (STLB hits) and STLB misses in millions that were incurred by the put, get, and delete operations were 120 and 30, respectively, when no large page was allocated. Allocation of large pages to internal nodes eliminated all STLB hits without changing the number of STLB misses, and therefore STLB misses were not incurred on internal nodes.

Software prefetch is one of the ways to reduce the overhead of cache misses. As investigated by Lee *et al.* [25], performance is expected to be improved when we prefetch small data that are randomly accessed. In an MB+-tree, identifiers are such data. For example, we issued prefetch instructions that prefetched all identifiers on a node before linear search started. When hybrid search was performed, we prefetched identifiers that corresponds to the keys in the segment, on which linear search is performed, before the search starts.

In Figure 12, we show throughput of get operations when prefetch instructions were issued The horizontal axis denotes the prefetch hint; Ti specifies that L$j(i < j)$ cache should be filled with the prefetched data and NTA specifies that only L1 cache should be filled with the data. Circles in the figure represent throughput of get operations when identifiers on the root node were prefetched. The circles show that throughput was the highest when the hint is T0, even though prefetching the identifiers did not affect throughput significantly. Because the number of entries on the root node is 2017, the identifiers could be stored in 8KB area. Therefore, the identifiers, which were frequently read, were likely

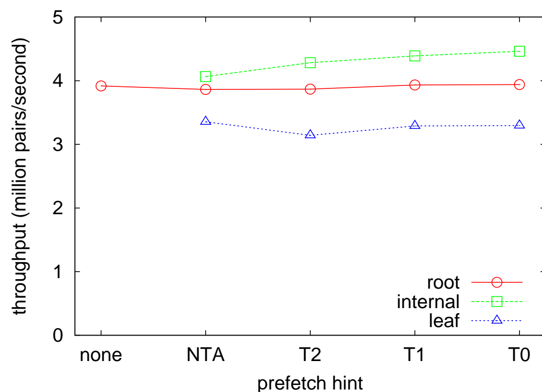


Figure 12: Throughput of get operations when identifiers were prefetched

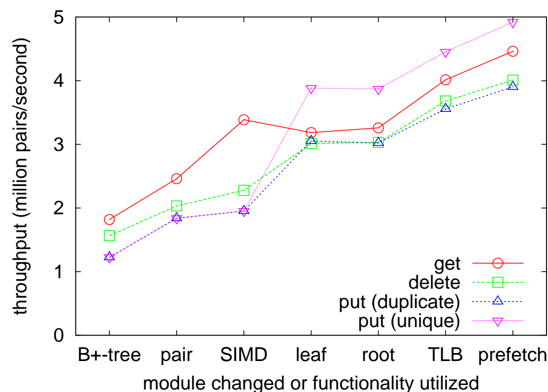


Figure 13: Throughput achieved by the modularization

to be in L1 cache.

Throughput of get operations increased by approximately 10% when identifiers on internal nodes at the level 1 were prefetched. The squares in the figure represent throughput when identifiers at the level 1 were prefetched and when those on the root node were prefetched using T0 hint. We can see in the figure that throughput increased when the identifiers at the level 1 were prefetched in L3 cache. There were 2017 nodes at the level 1 and the total size of the identifiers on the nodes was up to 1MB, which was much larger than the size of L2 cache. Therefore the identifiers at the level 1, which were read much more frequently than those at the level 0, incurred cache misses and stronger hints increase throughput. Prefetching on leaf nodes decreased throughput as the triangles show. This is because identifiers on internal nodes were excessively evicted when identifiers on leaf nodes were prefetched.

The modularization contributed to the performance improvement by prefetch because a part of identifiers should be prefetched. The following reasons account for the contribution: (1) identifiers on the root node were basically cached, (2) identifiers on internal nodes were excessively evicted when identifiers on leaf nodes were prefetched, (3) identifiers on internal nodes at a lower level may be out of cache and pollute cache little even when they are prefetched. We note that throughput of put and delete operations also increased when identifiers at the level 1 were prefetched.

Figure 13 summarizes this research, and demonstrates that the modularization of B+-trees brought two- or threefold performance improvement. Throughput of get, delete, and put operations are shown in the figure. The horizontal axis denotes modules that we changed from the B+-tree or hardware functionality utilized. We first changed the place where the pairs were stored from nodes to log-structured area, and then prepared nodes that could utilize SIMD instructions.

The modularization contributed to performance improvement because the selection of algorithms and the node size increased performance. We changed leaf nodes from sorted to unsorted ones to increase throughput of put and delete operations with minimizing deterioration of throughput of get operations. Making all nodes unsorted is clearly inappropriate because equality search is performed only on leaf nodes. Throughput of put and delete operations can be increased more by enlarging the size of leaf nodes if the uniqueness of keys is assured, even though that of get operations was decreased. We changed the search algorithm on the root node from binary to interpolation search. This increased throughput as long as the keys were uniformly distributed even though throughput of put and delete operations slightly decreased. The large node size account for the increase and the decrease in throughput, and there is room for further improvement as the number of pairs increases.

We allocated large pages to leaf and internal nodes independently to reduce TLB misses. The increase in throughput per large page when large pages were allocated to internal nodes was larger that when large pages were allocated to all nodes. Software prefetch increased throughput when we prefetched identifiers on internal nodes in the three-level modular B+-tree. In contrast, prefetching identifiers on leaf nodes decreased throughput. Therefore, the modularization contributed to performance improvement by the utilization of large pages and prefetch.

5. CONCLUSION

We modularized B+-trees and show effective selections of algorithms and the size of nodes at each level. As a result, two- or threefold performance improvement was achieved over a typical implementation of a B+-tree. The best selection in this paper formed three-level B+-trees, in which linear search was performed on small unsorted leaf nodes and on small sorted internal nodes except the root node, and interpolation search was performed on the large sorted root node. In addition, the three-level modular B+-trees enable us to utilize large pages and software prefetch only at levels specified. Evaluation results suggest that allocation of large pages to nodes at a high level takes precedence over that at a low level. Prefetching identifiers only on internal nodes increased throughput while prefetching those on all nodes decreased throughput. While utilizing techniques proposed in existing works in each module, we propose the method to make use of the techniques in a single tree and effective selections of modules.

The modularization increased throughput of get operations from 1.819 to 4.463 million pairs/second (2.453 times faster), and that of delete operations from 1.564 to 4.010 million pairs/second (2.564 times faster). The increase in throughput of put operations was larger than that of get

and delete operations. When the uniqueness of keys must be examined in run-time, throughput increased from 1.225 to 3.903 million pairs/second (3.186 times faster). When the uniqueness was assured statically, throughput increased from 1.225 to 4.920 million pairs/second (4.016 times faster). Therefore, the modularization was more effective for processing write-intensive workloads than read-intensive ones. As the next step of this research, we are planning to automate the process of selecting types of nodes because there is a wide variety of workloads and hardware, which have been still changing.

6. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and D. Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP*, pages 1–14, 2009.

[3] B. Atikoglu, Y. Xu, and E. Frachtenberg. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, pages 53–64, 2012.

[4] R. Bayer. *B-trees and databases, past and future. In Software pioneers*. Springer-Verlag New York, Inc., 2002.

[5] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[6] R. Bayer and K. Unterauer. Prefix B-trees. *ACM Transactions on Database Systems*, 2(1):11–26, 1977.

[7] P. Bohannon, P. Mcllroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *SIGMOD*, pages 163–174, 2001.

[8] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *SIGMOD*, pages 235–246, 2001.

[9] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B+-trees: Optimizing both cache and disk performance. In *SIGMOD*, pages 157–168, 2002.

[10] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR*, 2011.

[11] T. M. Chilimbi, J. R. Larus, and M. D. Hill. Improving pointer-based codes through cache-conscious data placement. In *Technical report 98, University of Wisconsin-Madison, Computer Science Department*, 1998.

[12] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[13] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: the Complete Book*. Prentice Hall Press, 2009.

[14] Google Inc. `http://code.google.com/p/cpp-btree/`, February 2013.

[15] G. Graefe. B-tree indexes for high update rates. *SIGMOD Record*, 35(1):39–44, 2006.

[16] G. Graefe. B-tree indexes, interpolation search, and skew. In *DaMoN*, 2006.

[17] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *DaMoN*, 2007.

[18] G. Graefe and P. Larson. B-tree indexes and CPU caches. In *ICDE*, pages 349–358, 2001.

[19] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Record*, 26(4):63–68, 1997.

[20] J. Gray and G. F. Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. In *SIGMOD*, pages 395–398, 1987.

[21] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache conscious B+-trees. In *SIGMETRICS*, pages 283–294, 2003.

[22] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual. In *Order Number: 248966-026*, 2012.

[23] T. Johnson and D. Shasha. Utilization of B-trees with inserts, deletes and modifies. In *PODS*, pages 235–246, 1989.

[24] C. Kim, J. Chhugani, N. Stish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, pages 339–350, 2010.

[25] J. Lee, H. Kim, and R. Vuduc. When prefetching works, when it doesn't, and why. *ACM Trans. on Archit. and Code Optim.*, 9(1):2:1–2:29, 2012.

[26] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *VLDB*, pages 294–303, 1986.

[27] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *PVLDB*, 3(1):1195–1206, 2010.

[28] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *SOSP*, pages 1–13, 2011.

[29] Oracle Corporation. *Oracle Berkeley DB, Writing In-Memory Applications*. `http://docs.oracle.com/cd/E17076_02/html/index.html`, 2011.

[30] Y. Perl, A. Itai, and H. Avni. Interpolation search - a log logn search. *Communications of the ACM*, 21(7):550–553, 1978.

[31] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB*, pages 78–89, 1999.

[32] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *SIGMOD*, pages 475–486, 2000.

[33] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project voldemort. In *FAST*, 2012.

[34] A. Yao. On random 2-3 trees. *Acta Informatica*, 9:159–170, 1978.

[35] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.