technische universität
dortmund

Master's thesis

Synchronization in B-trees for Main Memory Databases

Patrick Westerhoff
March 2014

Gutachter:
Prof. Dr. Jens Teubner
Dr. Michael Engel

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl VI: Datenbanken und Informationssysteme
http://dbis.cs.tu-dortmund.de

Contents

1. Introduction **3**
- 1.1. B-trees . . . 4
- 1.2. Related work . . . 5

2. Implementation **7**
- 2.1. Tree structure . . . 7
- 2.2. Main operations . . . 9
  - 2.2.1. Search . . . 9
  - 2.2.2. Insert . . . 10
  - 2.2.3. Delete . . . 11
- 2.3. Other operations . . . 12
- 2.4. Optimizations . . . 13

3. Evaluation **17**
- 3.1. Test setup . . . 17
- 3.2. Individual tests . . . 18
- 3.3. Sequences . . . 23
- 3.4. Conclusion . . . 25

4. Synchronization **27**
- 4.1. Synchronization strategy . . . 27
- 4.2. Implementation . . . 29
  - 4.2.1. Interlocking read locks . . . 31
  - 4.2.2. Insertion paths . . . 31
- 4.3. Evaluation . . . 32
  - 4.3.1. High concurrency test . . . 34

5. Conclusion **37**
- 5.1. Future ideas . . . 37

A. Additional test results **39**

B. Enclosed digital medium **43**

List of Figures **45**

Listings **47**

Bibliography **49**

1. Introduction

With the world becoming more and more data-centric, the importance of databases is constantly growing. Every application nowadays employs some sort of database, being it a lightweight configuration store, or full-sized database engines running on a network of machines. As the amount of data increases, so does the importance of well performing databases.

Traditionally, databases were always optimized for I/O behavior. With hard disks always being a slow storage compared to fast random access memory, the amount of I/O operations was what affected the performance of databases the most.

With Moore's law still applying to the development in computing hardware, the performance of CPUs and memory is still increasing at an impressive pace. In addition, new technologies allow the manufacturing of components at an affordable price, making it easy to equip machines with even more memory. As such, it is not uncommon now if database machines have enough memory to keep the whole database in the main memory—or at least a significant part of it.

Given that main memory is performing better than hard disks by a very large margin, the interest in main memory databases is constantly increasing. Such databases primarily work within the main memory and usually only access the hard disks to add an additional level of persistence. The increased interest can especially seen with the rise of *NoSQL* databases. Those are database systems that manage the data differently than relational databases, for example as documents or key/value stores. The database engine *Redis*[1] is an example for a key/value store that has recently gained a lot of attention, becoming the most popular key/value store since early 2013 [6]. Yet, Redis runs primarily in the main memory and will only make use of the hard disk to store snapshots or to asynchronously backup the data.

Just like memory, processors continue to become more powerful too. However, today, most of that power comes in form of parallelism, making CPUs include more cores and getting better at running threads in parallel. Unfortunately, to fully utilize this parallelism, applications need to be thread-safe, which often results in synchronization becoming the bottleneck in high-performance systems.

For database engines, this means that simply moving the data into main memory and optimizing it for main memory access is not good enough. To get well-performing systems,

[1]Redis: `http://redis.io`

investments into synchronization have to be made. This does not only affect the data itself, but especially the management structures of the database engine itself.

One of these structures is the B-tree, or more accurately a B+-tree. B+-trees are usually the first choice when utilizing index structures in modern databases, making them very valuable for increasing the overall performance of a database. They are involved in many of a database's operations: They are used when data is fetched over an index, and they constantly need to update whenever new data is inserted, or existing data is modified. So to employ concurrency for the database engine, B+-trees, as the underlying index structure, need to be synchronized as well.

This thesis looks at in-memory B+-trees from a modern hardware perspective. Starting with a standard single-threaded implementation, the base structure is evaluated on modern hardware with different optimization approaches to further improve the performance and investigate the behavior on current hardware. Ultimately, the implementation is expanded for a concurrent environment utilizing a custom synchronization strategy. Finally, the developed thread-safe structure is evaluated in a concurrent workload.

1.1. B-trees

B-trees were originally invented by Rudolf Bayer and Eduard McCreight in 1971 during their work at Boeing Research Labs [2]. They are similar to binary search trees in that they represent sorted data using a tree structure. However, unlike binary trees, B-trees allow nodes to have more than two children, increasing the fan-out of single nodes and ultimately reducing the overall height of the trees. As nodes are commonly stored as individual database pages, this reduces the number of I/O operations low that are necessary to navigate in the tree and increases the overall speed. In addition, B-trees are designed to keep all leaves at the same height, providing a stable tree structure.

Later in 1979, Douglas Comer [5] covered B-trees again and also introduced B+-trees. B+-trees are a modification of the original B-trees where the data is only stored within the leaves, while inner nodes serve only for navigation. B+-trees are what databases and file systems commonly use as the index structure.

B+-trees have three basic operations: search, insert and delete. When searching for a key, the tree is recursively descended until the leaf level where the key—if it exists in the tree—is stored. This works just like in binary trees, except that to decide which path to go from one node, multiple keys have to be evaluated. To insert an element into the tree, the element is directly added to the leaf its key belongs to. If any node exceeds its capacity in that process, it will be split up resulting in two half-filled nodes. Deletion works inverse to that and will remove the element from the leaf. If a node gets below its minimum capacity threshold of 50%, then elements from the siblings will have to be moved over, or two nodes

will have to be merged. That way, nodes will always maintain a minimum number of keys. The individual operations will be covered in more detail in the implementation chapter 3. In general, B+-trees provide a very stable and fast structure for indexing which by design only rarely has to change its structure due to node splits and node merges.
Although B-trees and B+-trees are different entities, only B+-trees are really relevant for databases and the context of this thesis. As such, throughout this thesis, the terms will be used synonymously, and whenever B-trees are mentioned, B+-trees are actually being referred to.

1.2. Related work

Since their original introduction in the 1970s, a lot of effort went into optimizing B-trees as index structures in database engines. However, with databases traditionally being optimized for I/O operations, the existing approaches do not fit well for the purpose of in-memory databases. Especially with modern hardware, utilizing a high number of concurrency and a large main memory, the amount of previous work is highly limited.
In 2002, Chen, Gibbons, Mowry and Valentin [4] covered the topic of cache efficiency with B+-trees. They were still targeting traditional databases that store database pages on disk, but added focus on cache efficiency. Classic disk-optimized B+-trees are designed to have a node size that equals the size of a disk page, so that individual nodes can be transferred with a single I/O operation. This means that the tree order is usually very big, resulting in large node sizes and bad cache efficiency. In contrast, cache-optimized trees have a small tree order, resulting in small node sizes so that the nodes fit into a single cache line, making them very cache-efficient but equally bad for I/O performance.
So the idea they presented in [4] is to combine the best of both into "fractal prefetching B+-trees" that embed a cache-optimized variant of B+-trees into the nodes of traditional disk-optimized B+-trees. That way, they could transfer the outer nodes with a single I/O operation, but navigate in the inner node by utilizing the cache. In their tests, the solution for them worked very well and proved to be a good way to combine cache efficiency with good I/O performance.
Focusing more on the in-memory aspect, Rao and Ross [9] also evaluated the cache efficiency of B-trees. They introduced *Cache Sensitive B+-trees*, "a variant of B+-Trees that stores all the child nodes of any given node contiguously, and keeps only the address of the first child in each node" [9]. Instead of storing variable pointers to child elements individually, their idea is to store all child nodes in a sequential memory section, improving the cache efficiency, and requiring only a single pointer to the beginning of the memory section to access any child. They presented different variations of their idea and compared it to a standard B+-tree implementation, yielding a clear performance boost in favor of their proposed index structure.

The most recent work on the topic was done in 2013 by Levandoski, Lomet and Sengupta [7], targeting flash memory on multi-core processors. They presented “Bw-trees”, a latch-free variant of B-trees that ensures that “threads almost never block” [7]. Instead of blocking, their solution involves virtual addresses for tree nodes that are stored in a custom mapping table. Tree nodes can then be updated by prepending a delta record to it, using the atomic compare-and-swap instruction to update addresses in their mapping table. Eventually, multiple delta updates are consolidated into new nodes, keeping the structure flat. In their tests, they showed a large performance increase over standard B+-trees found in existing database engines.

2. Implementation

For this thesis, it was early decided to come up with an own implementation for B-trees. The primary reason for this is that the implementation process itself would point out chances for further optimization and also give an idea of what is possible later in terms of synchronization. The other main reason is that this would obviously improve the familiarity with the B-tree algorithms, the programming language, and ultimately the implementation itself.
This chapter covers the basic implementation written in standard C99 [3]. The multi-threaded version is discussed later in chapter 4.

2.1. Tree structure

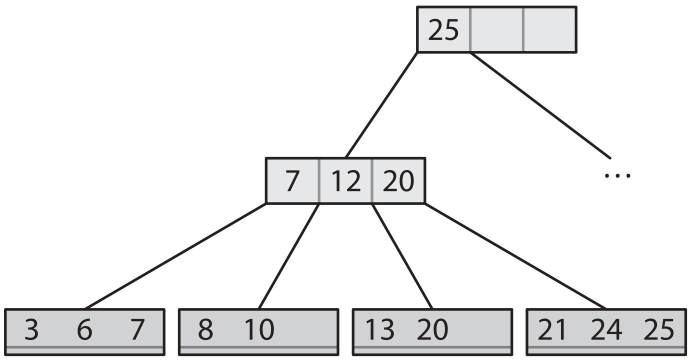

Figure 2.1.: An example B-tree. A key is always larger or equal to all keys within the child to its left. Keys that exist within inner nodes do not necessarily need to exist in leaves.

The implementation follows the tree structure from textbook B-tree implementations [8]. Inner tree nodes have a sorted list of keys which are interpreted as separators between the node's child trees. For a given key k, all keys m within the left subtree will be lower or equal to k, and all keys n within the right subtree will be larger than k: $m <= k < n$. While this relation defines a unique order on the tree, the existence of a key within an inner node does not guarantee for it to actually exist as an element within the tree. This is because when elements are removed, the keys in inner nodes are not updated. The tree elements themselves are stored in the leaves as key/value pairs of the actual data.
Inner B-tree nodes are represented by the `Node` structure. The tree order determines how many children each node can hold up to. As keys separate the children in a node, the

```c
typedef struct Node {
    int k;
    key_t keys[MAX_KEYS];
    void* children[ORDER];
    char childType;
} Node;

typedef struct Leaf {
    int k;
    key_t keys[MAX_KEYS];
    data_t data[MAX_KEYS];
} Leaf;
```

Listing 2.1: The `Node` and `Leaf` structures.

capacity for keys is one less than the tree order. The current number of occupied key slots is stored in the field k. This also determines the number of children, which is $k+1$. The field `childType` keeps track of what kind of children the node references. It can be either `CHILD_NODE` for other inner tree nodes, or `CHILD_LEAF` for leaves which are treated separately.

As this is a B+-tree implementation, leaves are the only nodes in the tree that contain actual data, which requires them to be treated separately. For consistency, they can have as many keys as inner nodes, and for each key, they can store one corresponding data entry. Leaves are represented by the `Leaf` structure.

```c
typedef struct BTree {
    char childType;
    void* root;
} BTree;
```

Listing 2.2: The `BTree` structure that stores a reference to the tree's root node.

The way B-trees work, they grow from the bottom. An empty tree starts with a single leaf as its only node, but as soon as that leaf exceeds its capacity, the tree expands into multiple leaves with a new inner node as their root. This growth continues as more elements are added to the tree, continuously causing new inner nodes to take the place of the tree root. As such, the node at the root is rather unstable, making it difficult to grasp the whole tree. To solve this, the `BTree` structure exists to hold the root node, providing stable access to the whole tree. The structure offers the flexibility to hold either an inner node or a leaf as the tree root. Similarly to the `Node` structure, it uses a `childType` field to encode whether the root is a leaf or a node.

To provide a stable access, the primary interface of this B-tree implementation operates solely on the `BTree` structures. The sole purpose of those high-level functions is to delegate the processing to the correct low-level function depending on the type of the tree root,

and to update the BTree structure if necessary. The low-level functions on the other hand operate directly on the Node and Leaf structures.
For the purpose of this thesis, the underlying type for keys, key_t, is just a plain integer. More complex types are imaginable but would unnecessarily bloat the implementation with comparison functions and generic type support. The type for data, data_t, was chosen to be a void pointer to simply allow referencing any kind of data. Another possibility would be to store the record identifier for a real database use instead.

2.2. Main operations

The three main operations on B-trees are search, insert and delete. All operations are implemented by recursively descending in the tree from the root to the leaves. The path that is used is determined by searching for a search key within each node. The search will yield an index which tells which subtree to navigate to next, or—when searching in leaves—whether or not an entry exists, or where it should be positioned at if it was.

```
int i = 0;
while (i < node->k && key > node->keys[i])
    i++;
```

Listing 2.3: Navigation using linear search for a key within nodes.

The key search is implemented using a linear search, where the index is being incremented as long as the search key is still larger than the node's key at that index. This ensures that, after the loop finished, the index points at the child which contains only keys that are lower or equal to the search key.

```
data_t btree_search (BTree* tree, key_t key);
void btree_insert (BTree* tree, key_t key, data_t data);
void btree_delete (BTree* tree, key_t key);
```

Listing 2.4: The three main operations: search, insert and delete.

2.2.1. Search

The purpose of the search operation is to look for a search key within the tree and return its data value if the entry exists in the tree. To complete the search, the navigational step from before is already most of the work that is required. After reaching the leaf and finding the appropriate index, all that is left is checking whether the key at that index equals the search key exactly or is just smaller than it. If an exact match was found, the data for that index can be returned. Otherwise, the search key was not found in the tree, and NULL is returned instead.

```
if (i < leaf->k && key == leaf->keys[i]) {
    return leaf->data[i];
}
```

Listing 2.5: Check whether or not the key at the found index matches the search key.

2.2.2. Insert

Inserting elements works similar first, by searching for the leaf where the key should be located if it was in the tree. If it actually exists in a leaf, the data entry can be updated if desired and no further work needs to happen. Otherwise, if the key was not found in the leaf, the actual insertion process begins.

If the leaf is not at maximum capacity, its elements are shifted to make room at the target index. The new element is then inserted there and the insertion is done. If there is not enough room in the leaf, then the leaf will have to split up. During that process, a new leaf is created, to which the right half of the original leaf is transferred to. The new element is then inserted in either the original or the new leaf depending on which side the target index belongs to. The leaf insertion then signalizes a split to the parent call, and passes back the new leaf and the splitting key. The splitting key is the key that should be placed between the two leaves in the parent node. It is equal to the right-most key of the *left* leaf, the original one.

The parent call is then an insertion of a new child node into an inner node, and works conceptually in the same way as an element insertion in a leaf. If the node is not at full capacity, the existing keys and children can be shifted to make room at the target index to insert the new child (and the splitting key). If the node does not have enough room for the new child, then it also has to split up. In that case, a new node is created which stores the same child type as the original node. The right half of the original node is then transferred to the new node and the new child is inserted at the correct position in either side. The node insertion then also signalizes the split to the parent call and passes the new node and the splitting key. Because a split of a node eliminates a separating key that would separate the right-most child of the left node and the left-most child of the right node, that exact key can be used as the splitting key between the left (original) node and the right (new) node.

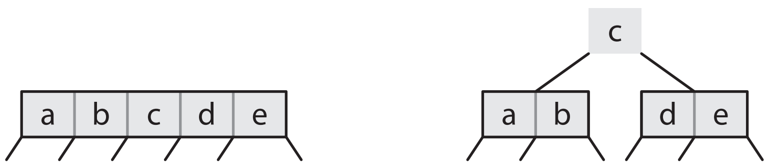

Figure 2.2.: During the insertion, a node split makes one key redundant which can then be used as the splitting key.

This node splitting process then continues on while ascending the tree again, until a node is found that has enough room to not require a split, at which point the insertion process ends. Otherwise, if the splitting happens to go up to the root node, requiring the root node to split up too, then a new inner node is created that stores the original root as the left child and the new node as the right child. That created node then becomes the new root, and the tree height is increased by one.

2.2.3. Delete

Deleting elements from the tree works inverse to the insertion process. While insertion makes it possible for nodes to exceed their capacity, requiring a node split, deleting may result in nodes going below their capacity threshold of 50%. In that case there are two options: Merging two adjacent nodes into a single one, or migrating individual children from an adjacent node. The former option is applicable if both nodes are at 50% capacity, while the latter requires the source node to have enough children left to not reach the threshold itself.

Deleting keys works similarly to inserting new elements. First, a search descends into the leaf and tries to find the key that should be removed from the tree. If the key could not be found, then no deletion is necessary and the process ends. Otherwise, the entry is removed from the leaf and the resulting gap is filled by shifting the other elements. If afterwards the leaf still contains enough elements, then all is well and the deletion process is completed. Else, it will signalize to the parent call that it is below the minimum threshold.

The parent call will then first take a look at the left and the right sibling of the current, underfilled node. If the sibling with the biggest number of children is above the minimum threshold, then it can be used to migrate children into the current node. If it is the right sibling, then the left-most child is moved into the current node; if it is the left sibling, the right-most child is moved into the current node. As it is only possible to get one below the minimum threshold, only one child has to be moved. After the node migration, all involved nodes are exactly at or above the minimum threshold, so no additional work needs to happen and the deletion process is completed.

If no sibling is filled enough to perform a migration, a node merge has to happen. This is the exact inverse of a node split and fills one node to its capacity limit, and removes the emptied node from the tree. To avoid having to shift children within a node, a node merge always happens from the current node into its left sibling. That way, the contents of the left node can stay at their original index, and all children and keys of the current node are simply appended to the left node. The splitting key that is necessary to separate the right-most child of the original left node and the left-most child of the current node is taken from the parent, which is again the inverse to what happened during the node split. If no left sibling exists, the whole view is shifted, so the right child is merged into the current node instead. After the node merge, the now empty node is removed, which can then cause the parent node to go below the threshold itself. If that is the case, the parent call is again notified and the same process happens again at the level above.

Just like with the insertion, this process continues to bubble up until the deletion succeeds without the node going below the minimum capacity threshold. At that point, the process is completed. If it happens to go up to the root, then the node is removed there instead. As the root has no lower limit of children, this can continue until there is only one child left. At that point, that child becomes the new root of the tree, and the previous root node is removed.

2.3. Other operations

Apart from the three main tree operations, the implementation includes a set of additional operations that deal with the management of the `BTree` structure, or provide some other utility functionality.

```
BTree* btree_create ();
void btree_destroy (BTree* tree);
BTree\textit{ btree_import (key_t} keys, data_t* data, int length);
void btree_print (BTree* tree);
```

Listing 2.6: The other utility operations.

The two most notable functions are `btree_create` and `btree_destroy` which are used to create an empty tree, or clean up and destroy an existing tree. The create function will allocate and initialize a `BTree` structure with an empty leaf as its root. As such it provides the minimum structure to successfully use the other functions that operate on the `BTree` structure. The destroy function is meant to destroy an existing tree and free all its allocated memory. It will recursively visit every node's children and destroy them, until ultimately the `BTree` structure is freed after making sure that no dangling nodes are left. As such, the create and destroy functions mark the start and the end of an individual B-tree.

Aside from creating an empty tree with the create function and inserting elements one by one, trees can also imported from a flat array. The `btree_import` function will take a sorted array of keys and an array of their respective data values, and will automatically create a B-tree from it. The resulting tree will be balanced to be effective for future operations on it. For this, all leaves and nodes created will in average be filled by 75%. This ensures that there is enough room for future insertions without immediately triggering node splits, but also that the nodes are filled enough to allow deletions without nodes going below their minimum threshold requiring node migrations or merges.

Finally, there is the `btree_print` utility function which will print a representation of the tree, allowing to see the exact structure of a tree instance. This is especially helpful to debug the other operations, keeping track of the changes that happen to the tree or individual subtrees, and to validate them in general. For example, after applying some optimization to the implementation, test trees can be printed and compared to ones generated before to make sure that the new implementation still produces the same results.

2.4. Optimizations

Based on the implementation discussed in the previous section, different optimization ideas were implemented and considered for the tests. These optimization ideas will be introduced in this section.
Profiling an early version of the implementation showed that a major part of the execution time was spent on the index search. This makes a lot of sense given that the tree navigation does exactly that, and all tree operations start with it to descend the tree. This gets even more important when considering that the probability for node splits during insertion, and node migrations/merges during deletion is by design very small. And as the index search is implemented as a linear search, it is not surprising that a considerable amount of time is spent on it.
As such, an early idea was to improve the key search to employ a faster search algorithm. To not introduce a too heavy overhead, an iterative binary search was chosen. As the search is not looking for an equality match but just for the largest key that is larger or equal than the search key, a standard implementation such as the one in the C standard library[1] could not be used. Instead, the binary search is implemented as shown in listing 2.7.

```
int i = 0;
int iMax = node->k;
while (i < iMax) {
    int mid = (i + iMax) / 2;
    if (key > node->keys[mid])
        i = mid + 1;
    else
        iMax = mid;
}
```

Listing 2.7: Navigation using binary search for a key within nodes.

The profiling also showed that a fair amount of time is spent on copying over children and keys when splitting or merging nodes. In each case, `ORDER / 2` iterations are necessary to copy the contents of a half node to another one. Depending on the target index during insertion, additional shifting of many children adds to the time too.

[1]bsearch: `http://www.cplusplus.com/reference/cstdlib/bsearch/`

To remedy this, the data can be moved with `memcpy`[2] instead. This would be feasible because the data being moved is always a whole block. For example when copying children 5–8 from one node to another (assuming a tree of order 8), those four child references would all be in consecutive memory, allowing them to be moved together. For the child shifting, which involves overlapping memory, `memmove`[3] can be used instead.

During the implementation of the B-tree, the code complexity to handle the different cases during node splits, migrations and merges was increasing very quickly. Taking the insertion of a child into an inner node, there are already 3 different cases, depending on whether or not the node has enough room for a simple insertion or which side the new element needs to be inserted to during a node split. Adding to that, for the side where the inserted element belongs, the elements to each side of the target index need to be handled differently—to keep the target slot free—requiring another branch. Listing 2.8 shows the branching complexity for the insertion process.

```
if (node->k == MAX_KEYS) {
    if (i < mid) {
        // move right half to new node
        // shift and insert to left side
    }
    else {
        // move right half < i to new node
        // insert to right side
        // move right half > i to new node
    }
}
else {
    // shift and insert
}
```

Listing 2.8: Original insertion complexity.

A common solution to avoid this level of complexity is to introduce an *overflow slot.* The basic idea is to allow nodes to hold one more key and child than their actual capacity allows. The insertion can then happen in two separate phases: First, the new child is added directly to the node (likely requiring shifting keys and children). Afterwards, if the node exceeded its capacity during that process, the node can be split independently. This especially means that the inserted child does not influence the split at all because by the time the split is done, the child was already inserted. Listing 2.9 shows the reduced complexity the introduction of an overflow slot would allow.

Unfortunately, the overflow slot would only benefit the insertion process. The complexity during deletion, which involves considering a merge from the left or the right sibling, and finally a merge of two siblings, would not change at all. Another more direct downside is

[2]memcpy: `http://www.cplusplus.com/reference/cstring/memcpy/`
[3]memmove: `http://www.cplusplus.com/reference/cstring/memmove/`

```
// shift and insert
if (node->k > MAX_KEYS) {
    // move right half to new node
}
```

Listing 2.9: Reduced insertion complexity with overflow slots.

that this would increase the size of every node in the tree by a key and child slot which will be only ever filled during a node split itself, of which only one happens at a time per insertion process. As such, the complexity reduction from this compared to the effectively wasted memory is questionable.

As the introduction of an overflow slot would require a complete rewrite of the whole insertion implementation, and because of the limited time for this thesis, this final optimization idea was not implemented and tested though.

3. Evaluation

The implementation presented in the previous chapter was throughly tested with different parameters. In the following chapter, the results of those tests are presented and evaluated. The results should give an idea about the efficiency of the implementation and further tell how to configure B-trees most efficiently for in-memory use.

3.1. Test setup

All tests were run on an Intel Core i7-4770, a 4-core (8 threads) Haswell processor with 3.4 GHz and 8 MB L3 cache. The machine is equipped with 32 GB RAM, and runs on a 64bit Debian. For the tests in this chapter, the code was compiled with the Intel C compiler (version 14.0.1).

For the measurements, the *Performance Application Programming Interface* (PAPI) library[1] was used. PAPI is a high-level interface that provides consistent access to accurate timing facilities, and to the CPU's performance counters. In addition to the elapsed real time (also often referred to as wall-clock time) of the test execution, the tests also keep track of the cache misses on the level 1, level 2 and level 3 cache, as well as the number of branch mispredictions to evaluate the branching complexity.

Each presented test is run for different tree orders, ranging from a small tree order of 5, to a very large order of 5,000. The tests are also run for the different optimization options, resulting in four different values: the original implementation, the implementation with binary key search for navigation, the implementation with memcpy for the transfer of keys and children, and finally a combined version using both binary search and memcpy. To receive stable results, each test is also run 5 times and the mean of the individual results is taken for the evaluation.

The tests can be separated into two different test sets. The first set consists of tests that benchmark each single B-tree operation individually. This allows evaluating the performance of a single operation without any impact from other operations or the natural distribution of the operations in a live environment. The second set of tests runs a sequence of mixed actions on the same tree. This is done to simulate a more practical situation where all three B-tree operations are used together on the same tree.

[1]PAPI: `http://icl.cs.utk.edu/papi/`

The individual insertion test will benchmark the insert operation by inserting a sequence of numbers into an empty tree. Before measuring the inserts, it shuffles the sequence to ensure that the inserts happen at a random distribution across the tree nodes.

As both search and delete require a tree to be filled before they can do anything, both tests start with a pre-initialized tree. They create the tree similarly to how the insertion test works, by inserting a sequence of numbers in a random order into an empty tree. The tests then shuffle the sequence again to get their own work set on which they run the search or delete operations. As the test work on the same set of numbers that were used to initialize the tree with, they are guaranteed to perform their operation successfully: Searches will find the key and return the data, and deletions will find the key and remove the element from the tree.

As implied, the sequence tests work quite differently to the individual tests. The sequence tests load an external *action sequence* and execute that on a tree. The sequences are simple text files that contain one action per line, with each action following the format `<type><number>`. The type can be either `s` for searches, `i` for insertions, or `d` for deletions. An action `d463` for example would mean *"delete the key 463 from the tree"*.

The used action sequences are previously generated by an external Python script. Depending on the configuration, the individual actions occur at different probabilities to match the practical use where searches for example are far more common than the modifying operations. The sequence also includes a fixed initialization sequence at the beginning which consists of only insertions to fill the empty tree with some elements before the actual tests start.

The test will then read the initialization sequence and add the elements to an empty B-tree, similarly how the initialization phase for the individual search and deletion tests work. Afterwards, the actual test starts, and the execution of the whole sequence is benchmarked. Just like the individual tests, the sequence files contain each sequence five times to provide stable results. After each sequence, which is signalized by a special stop action, the tree is reset and a new benchmark starts with a new initialization sequence on an empty tree.

3.2. Individual tests

In the following chapter, the results from the individual tests will be evaluated. As the insert operation is a generic operation that shows both the effects from navigating as well as modifying the tree, it will be evaluated first in more detail. As the results for both the search and the delete operation showed a very similar behavior, especially in terms of cache misses and branch predictions, they will not be covered with too much detail.

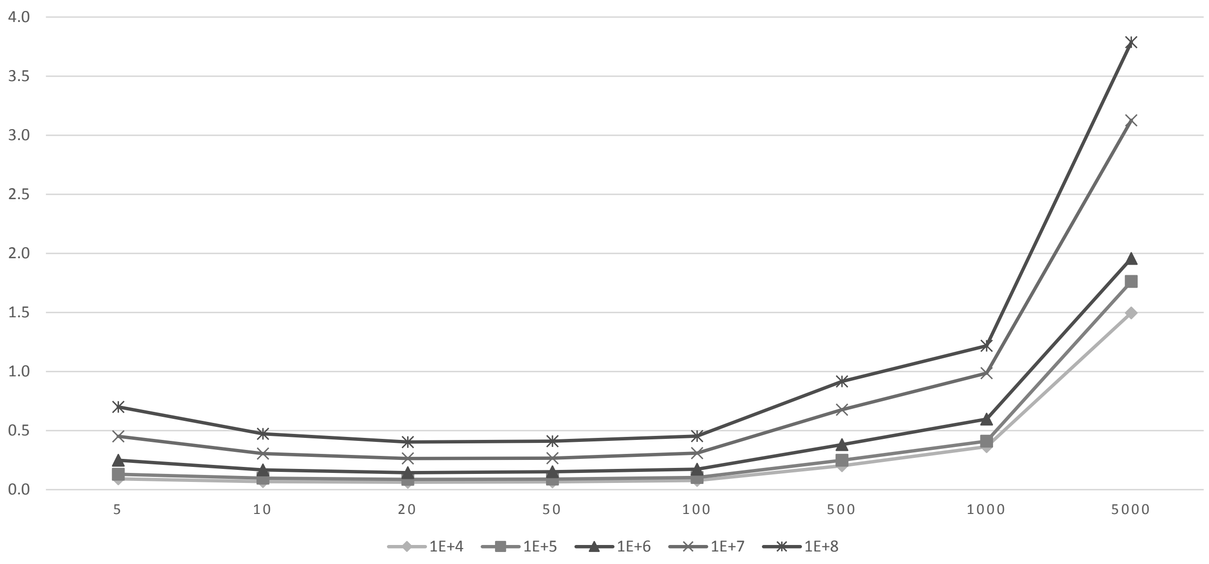

Figure 3.1.: Effect of different tree orders on the insert operation.

Figure 3.1 shows the effect of different tree orders on the insertion operations. The tree orders are arranged on the horizontal axis, ranging from 5 to 5,000, while the vertical axis shows the average time a single insert operation takes on the tree. For most of the different tree orders, the average time of a single insert is below 0.5 microseconds. For comparison, in [9] the average insertion time ranged from approximately 4 to 7 microseconds depending on which implementation they used. That shows that our simple implementation already performs very decently on current hardware.

In the figure, the different graphs represent different sizes of sequences that were added to an empty tree, ranging from 10,000 to 100,000,000 insertions into a single tree. Not surprisingly, the average insertion time increases with larger trees. This is reasonable given that as a tree gets larger and contains more elements, its height increases and navigating in the tree will require more steps than on a smaller tree. The graph also shows a surprisingly stable increase of the time, hinting that the implementation does not further degrade at certain sizes and tree orders.

As far as the tree order goes, the chart shows very clearly that larger tree orders of 1000 and especially 5,000 are more harming than benefiting. The easiest explanation for this is that larger tree orders means that there will be more keys within the nodes, increasing the effort to search within the key lists when navigating. That further confirms the observations from the early profiling as mentioned in chapter 2.4. To improve this situation, our proposed solution was to use a binary key search instead of linear key search. Figure 3.2 shows the results from that and the other optimizations.

Considering only the differences between the original implementation and the optimization using binary search, we can be sure that no other insert-related properties cause additional

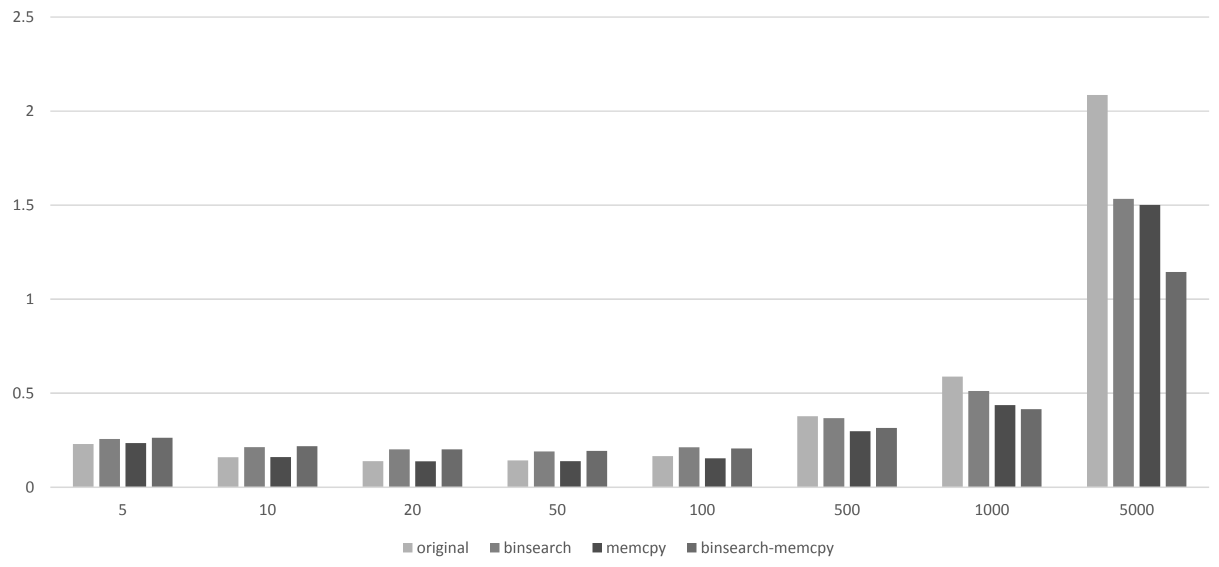

Figure 3.2.: Insertion with different optimizations (original, binary search, memcpy, and both).

effects, as the binary search optimization only changed the search within the nodes. As can be seen from tree orders above 500, the binary search helps a lot, reducing the insert time by one fourth for a tree order of 5,000. The graph also shows, that binary search includes a noticeable overhead for smaller tree orders, which is likely caused by the random access of keys when using binary search, instead of a simple sequential access when using linear search.

Figure 3.2 also shows the impact of the memcpy optimization. We can see that for increasing tree orders, the benefit of copying elements using memcpy gradually increases. And even for small tree orders, the change to memcpy does not negatively affect the execution time. The fourth optimization option combining both binary search and memcpy obviously results in combined effects: For small tree orders, the additional cost of binary search prevails while for larger tree orders the benefits of both binary search and memcpy help to improve the execution time substantially.

The impact of large tree orders can also be seen when looking at the number of L1 cache misses in figure 3.3. Starting with a tree order of 500, the amount of L1 cache misses starts to explode. Note that the tree order of 5,000 is not included in the graph, as each value is approximately 12 times larger than the cache misses for a tree order of 1,000. Other than that, with an increasing number of total insert operations into a single tree (horizontal axis), the graph shows a similar increasing behavior as noted in figure 3.1.

Comparing the different optimizations, we can see that the original implementation is maintaining the cache most efficiently. As suggested before, for the binary search, the random access pattern is likely the reason the caching fails.

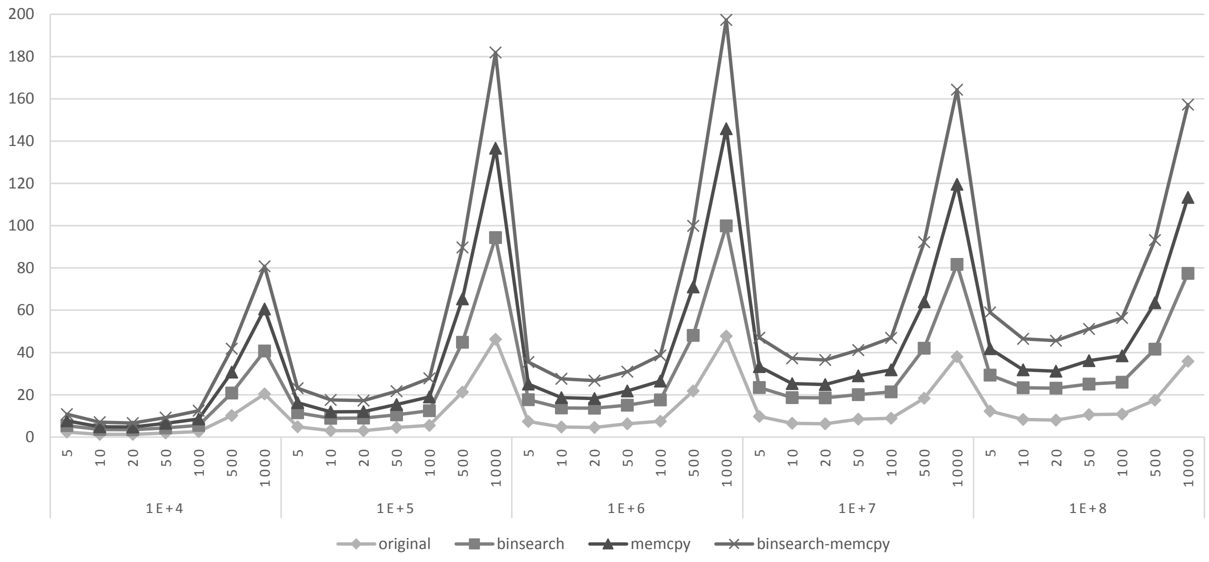

Figure 3.3.: Average L1 cache misses per insert operation (with increasing number of total insertions).

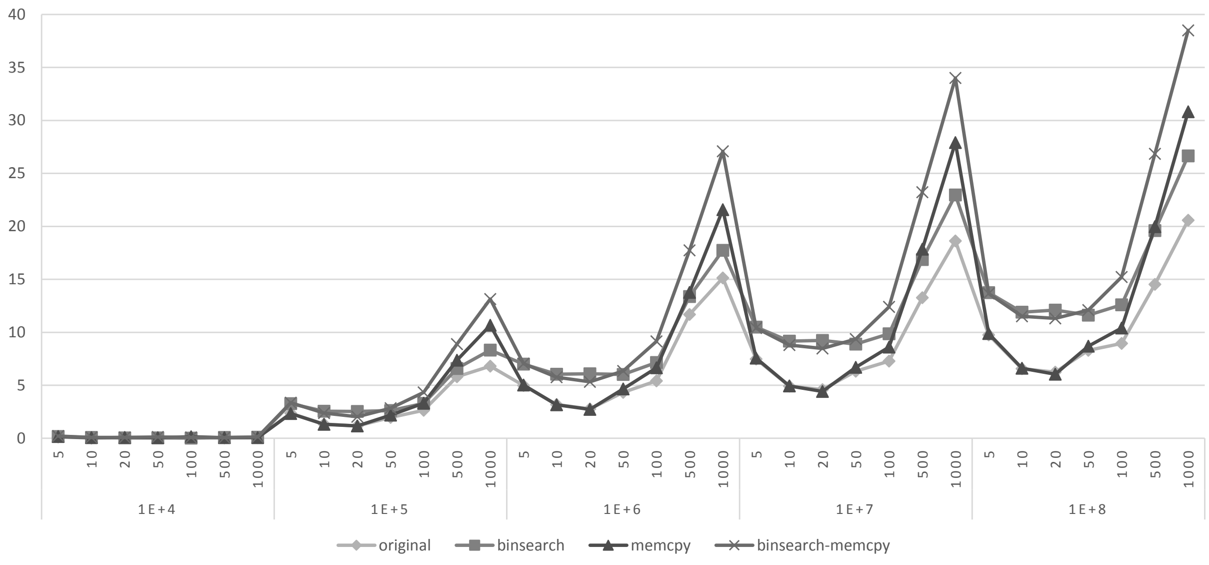

Figure 3.4.: Average L2 cache misses per insert operation (with increasing number of total insertions).

Looking at the L2 cache misses in figure 3.4, we can see the rocketing cache misses for tree levels above 100 again. Furthermore, we can see an improved behavior for the memcpy optimization here, which was suffering from more L1 cache misses compared the original implementation before.

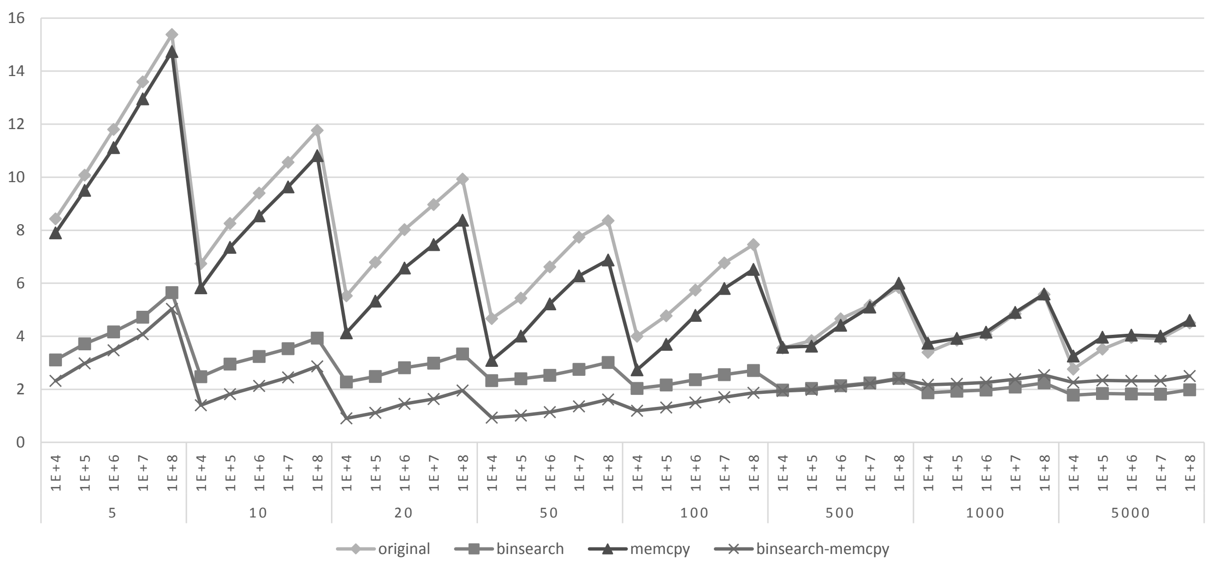

Figure 3.5.: Average branch mispredictions per insert operation (with increasing number of total insertions).

Figure 3.5 shows the average branch mispredictions per insert operation for different tree orders and again for different numbers of total insert operations on a single tree. The graph shows very clearly the differences between linear key search and binary key search, the latter resulting in far less branch mispredictions. This behavior is natural given that binary search by design makes less comparisons than linear search, as such resulting in a smaller probability of branch mispredictions. We can also see that, with an increasing tree order, the mispredictions get less common. This is because for larger tree orders, the search includes more comparisons of which most of all evaluate to false, meaning that the expected outcome becomes more stable as the number of comparisons increases.

In conclusion, with the results from just the insertion tests, we could already see that choosing tree orders above 100 are not a good idea for in-memory databases. The cost of cache misses and the increased search time is just too overwhelming. On the other hand, the desired benefits from using binary search only start appearing at those tree orders, which makes the optimization using binary search also impractical. And surprisingly, we could not yet see a clear improvement from using memcpy compared to our original implementation either.

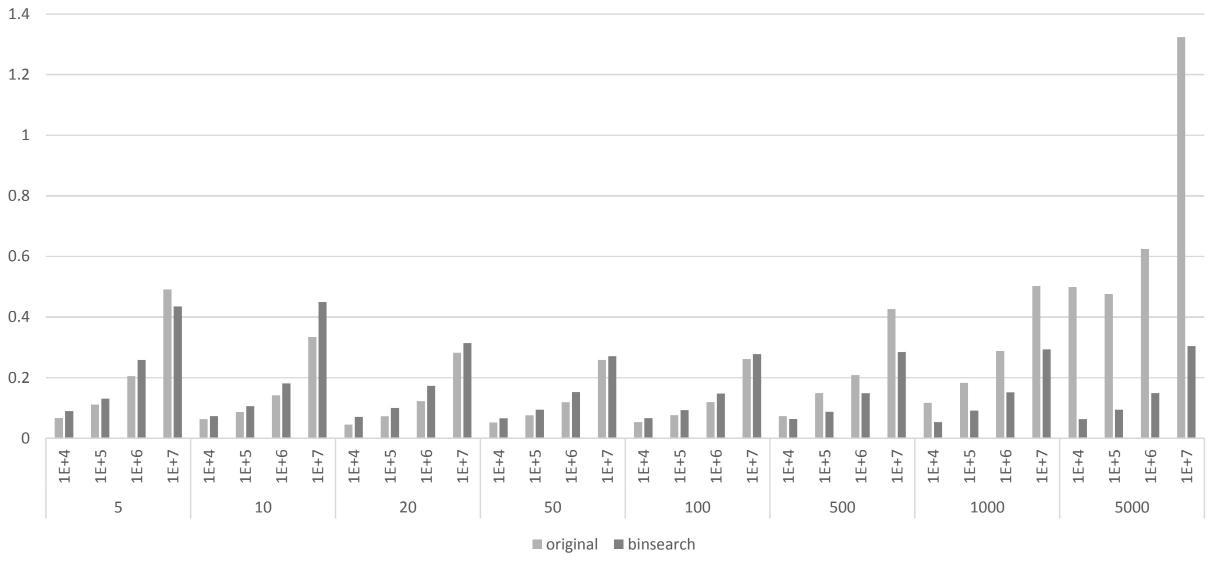

Figure 3.6.: Average execution time for a single search operation with varying tree sizes and orders.

Figure 3.6 shows the average time of a single search operation for different tree orders and tree sizes. As can be seen, even for larger trees containing 10,000,000 elements, the average time for a single search operation is around 0.3 microseconds. Again for comparison, in [9] the average time for a single search was about 2.5 microseconds.
Also, as we have previously concluded, the binary search optimization really proves to be impractical, yielding only better performance for tree orders beyond 100.

Figure 3.7 shows the average time a single delete operation takes to complete. The timing results bear a clear resemblance with what we have seen for the insert operation, also averaging below 0.5 microseconds per deletion.
Furthermore, next to the known negative behavior of binary search for tree orders below 500, we can again see that the memcpy optimization yields no noticeable benefits.

3.3. Sequences

As mentioned in section 3.1, sequence test combine all operations into a random sequence. As such, the average time can now only be determined per action, which can be either search, insert or delete.
Figure 3.8 shows the result from the first sequence group. The sequences were created with a 40% probability for search actions, 40% for insert and 20% for delete. As such, the sequences are heavily insert-oriented. The horizontal axis shows the sequence length, ranging from a 1,000 to 10,000,000.

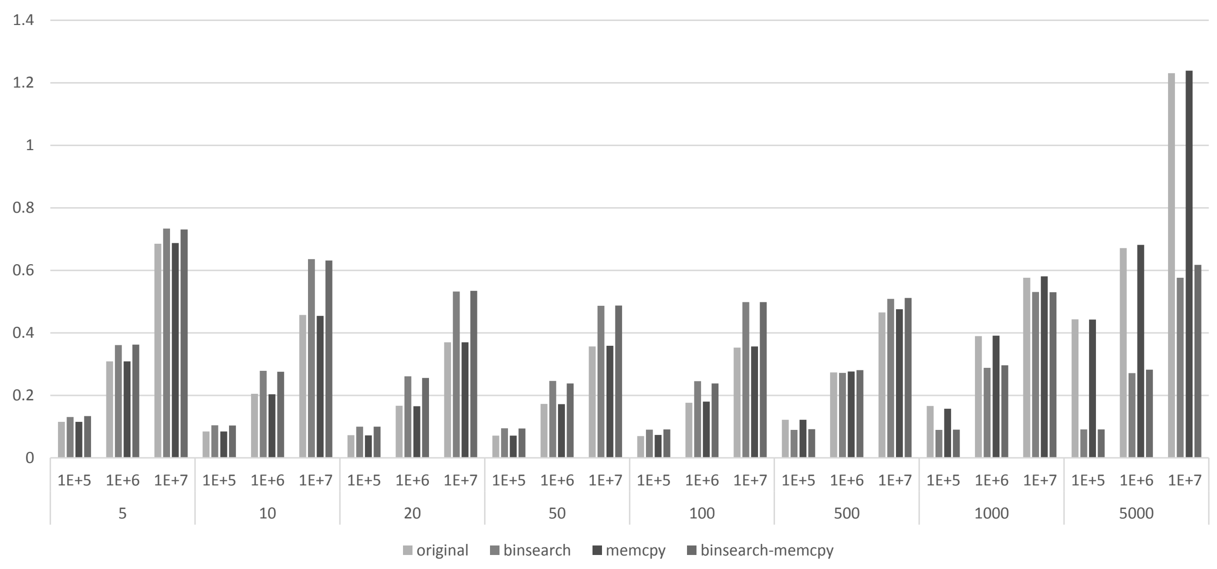

Figure 3.7.: Average execution times for a single deletion with varying tree sizes and orders.

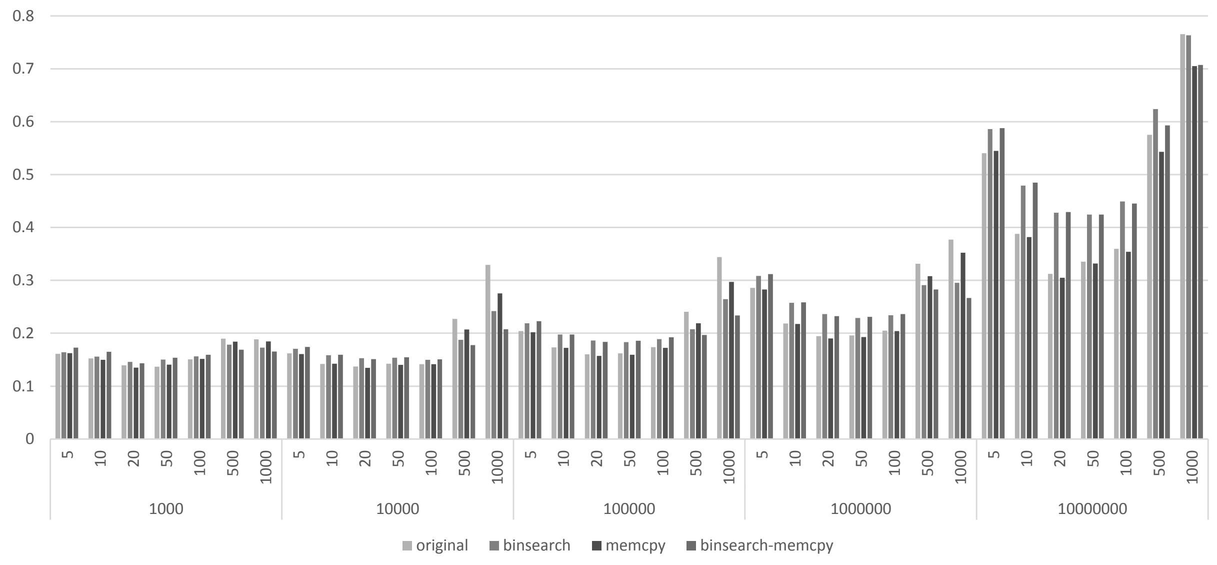

Figure 3.8.: Average execution time per action (40% search, 40% insert, 20% delete).

The graph again confirms that medium tree orders of 20 to 100 are the ones that perform the best in average, resulting in average execution times of 0.2–0.4 microseconds. We can also see that the tree optimizations again yield no improved results.

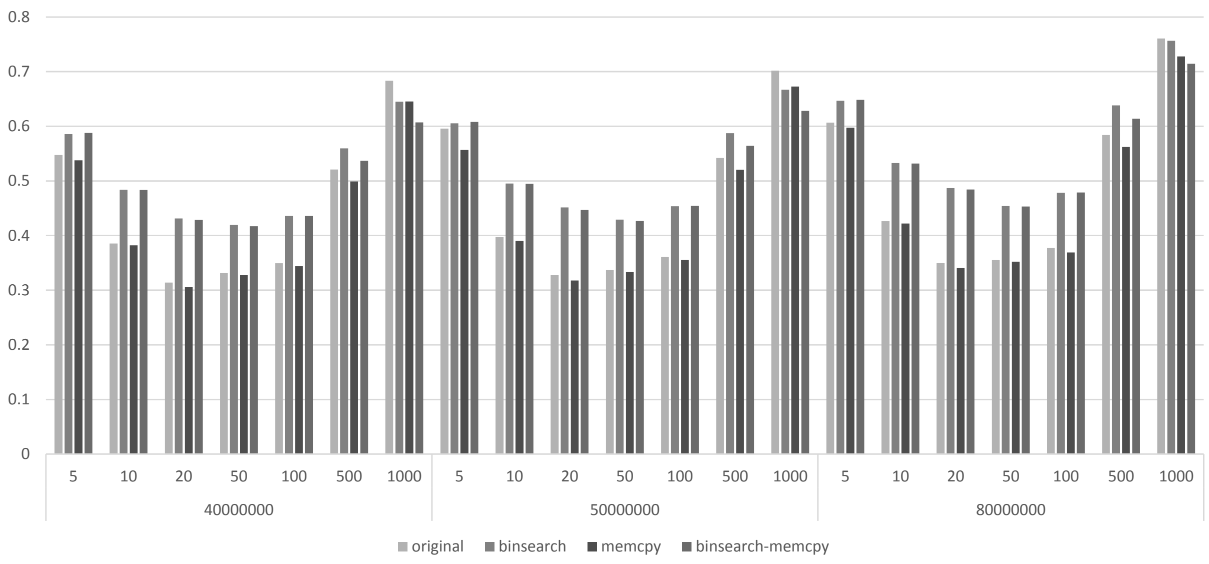

Figure 3.9.: Average execution time per action (65% search, 25% insert, 10% delete).

Figure 3.9 shows the result form the second sequence group. These sequences were created with a increased search probability of 65%. The probability for inserts was 25%, leaving a 10% chance to deletions. Also, the generated sequences were of a much larger length, which explains the overall increase of execution time, to around 0.5 microseconds per action for tree orders between 20 and 100.

3.4. Conclusion

Although we ran a large number of tests with different parameters, the end results did not show too many variations. One thing we could see in every result was the importance of the tree order choice. For a in-memory use, a tree order below 100 should be chosen to get the best performance and to stay cache-efficient.

Also, despite our efforts to further optimize the initial implementation, the test showed that neither of the ideas further improves the performance. The downsides of the binary key search were just too large to be compensated by the fundamentally more efficient search.

Even the memcpy optimization turned out to be less effective than hoped. The reason for this is likely because of how memcpy works internally. In its essence, all it does is loop through the source and target memory sections and copy over the values word by word.

However, this is very similar to what our original implementation did, hence the minimal performance differences.
Regardless of that, our implementation proved to be quite efficient after all, making it a good candidate to be evaluated in a concurrent environment in chapter 4.

4. Synchronization

Databases are usually highly concurrent systems. As such, the underlying index structures are required to employ some kind of synchronization to ensure thread-safety in a multi-threading environment.
A simple situation that shows where this is necessary for B-trees is during an insert operation. If the insertion causes a node to split at the same time a concurrent search decides to take the path into that node, then it is possible that the correct next path for the search is already moved to the newly created node from the node split, making it inaccessible. As such, the search would not be able to continue descending and would ultimately fail.
This chapter covers the changes made to the implementation introduced in chapter 2 to maintain thread-safety, the used synchronization strategy and finally an evaluation of the modified implementation.

4.1. Synchronization strategy

The operations on B-trees can be separated into read-only navigation parts and modifying parts that change the structure of the tree. During those modifying parts, only the modifying thread should have access to the affected nodes, to prevent other threads from navigating into broken areas of the tree.
This situation matches exactly the capabilities that *readers–writer locks* offer. A readers–writer lock, or RW-lock in short, allows multiple readers to concurrently access the resource, but also allows a single writer to acquire exclusive access for write actions on the resource. The exclusive write access prevents both readers and other writers to get access to the resource, so they are protected from ongoing modifications of the tree.
Write locks have to be acquired whenever an operation starts to modify the tree. The read locks have to be acquired during the navigation in the tree. However, a read lock does not need to stay acquired for the full duration of the navigation step. Once the decision has been made which node to visit next and a read lock is acquired for that node, the lock for the current node can be released. This results in interlocking read locks during the navigation.
As mentioned before, all operations start by navigating from the tree root at the top to the leaves, before they decide about any further action. For search operations, after reaching the leaf, no other nodes are touched again, making the search operation operate

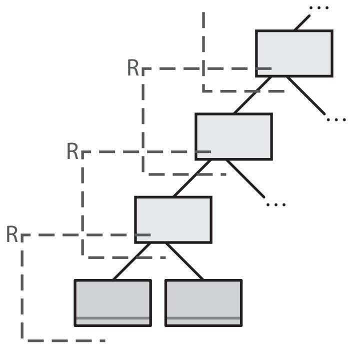

Figure 4.1.: The read locks are interlocking; as soon as the next read lock is acquired, the current lock can be released.

from top to bottom only. Modifying operations however usually start their actual work only once they reached the leaf by then ascending in the tree again. This makes modifying operations work in the opposite direction as navigation within the tree.

With a naive locking approach, navigation tasks would acquire read locks while descending down in the tree, and modifying tasks would acquire write locks on their way up from the leaf. This would however mean that locks would grow down from the root and also up from leaves, resulting in guaranteed deadlocks. To avoid this, write locks will have to be acquired in the same direction as read locks, from top to bottom.

The way this is solved is that modifying operations will work in multiple passes. In the initial pass, the operation memorizes the path it takes during the navigation and also keeps track of some meta information about the nodes it visits, for example their capacity. Once it reached the leaf, the information is then evaluated when deciding what actions to take next to complete the operation. Based on the collected meta information, it is then possible to deduce from which node level write locks are required. The execution then returns back to the top without doing anything else, and the next pass starts.

In the following pass, the operation then navigates down to the leaf again. Write locks are acquired for each visited node from the node level that was previously decided on. While descending the tree, each taken step is also compared with the stored path from the previous pass. If the operation reaches the leaf and the taken path matched the previous one, then all required locks were also acquired on the way down and the operation can complete its work, accepting this pass. Otherwise, the now taken path is remembered and evaluated instead, and the execution returns to the top to start another pass. On the way up, all taken write locks are also released again. This process repeats until a path matches the one from the pass before and the work can be completed.

By navigating in the tree at least twice, the first pass can be used as a “dry run” to find out exactly what changes need to be made before actually committing them. If two subsequent passes take the same path, then that means that the tree structure did not

change in between for the relevant nodes. This guarantees that the meta information from the previous pass is still correct and write locks acquired because of that are sufficient. Because acquired write locks are always released when bubbling up, passes that are turned into dry runs because of non-matching paths will also not cause any problems.
The only downside in this case is that write locks will be temporarily acquired without a real need for them. However in practice, modifying operations are relatively rare compared to read-only operations, and the probability for operations to impact the same part of the tree is likely small. Also, implementations can easily stop acquiring write locks as soon as the path is detected invalid to keep the number of redundant write locks as low as possible.
In conclusion, this locking strategy gives a very simple mechanism to provide synchronization without acquiring write locks too early, and without performing the operations optimistically with the option to roll back changes in case of conflicts as it is often done for database-backed B-trees.

4.2. Implementation

As discussed in the previous section, the locking strategy is built on RW-locks. To allow individual locks on all nodes, locks are added to all three structures introduced in chapter 2: `Node`, `Leaf` and `BTree`. Because the time a single lock will stay acquired is considerably short, the locks are implemented with simple spin locks [1]. The RW-locks are stored as 32 bit integers, of which the low-order word is a *counting semaphore* that counts how many threads are accessing as a reader. The high-order word is a *binary semaphore* that signalizes whether a writer has or wants exclusive access.
For a reader to acquire a read lock, it has to wait until a potential writer has left, and then increment the counting semaphore by one. To unlock, the counting semaphore is simply subtracted by one again.
When a writer wants to acquire a lock, it has to do two things: First, it has to acquire the binary lock to get write access. Once that happened, other writers and readers will have to wait for the writer to finish before they can acquire any other lock. The writer then has to wait for all readers that still hold a lock to release their lock. Afterwards, the writer has exclusive access and may continue. Releasing the write lock is equivalent to releasing the binary lock which will allow other waiting writers and readers to get access again.
Listing 4.1 shows the implementation of the RW-locks using built-in functions for atomic memory access[1]. The implementation works without a FIFO queue, which is commonly used to prevent writer starvation [1]. Writer starvation is a state in which writers are continuously prevented from acquiring the write lock by readers that still hold read locks. In this implementation, this is primarily avoided by having the write locking mechanism

[1]Built-in functions for atomic memory access: `http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html`

```c
void readLock (volatile int* rwlock) {
    // wait while a writer has the write lock
    while (__sync_fetch_and_add(rwlock, 1) >= RW_W_LOCK) {
        __sync_fetch_and_sub(rwlock, 1);

        // pause while write-locked
        while (*rwlock >= RW_W_LOCK) {
            _mm_pause();
        }
    }
}

void readUnlock (volatile int* rwlock) {
    __sync_fetch_and_sub(rwlock, 1);
}

void writeLock (volatile int* rwlock) {
    // wait to acquire the write lock
    while (__sync_fetch_and_add(rwlock, RW_W_LOCK) >= RW_W_LOCK) {
        __sync_fetch_and_sub(rwlock, RW_W_LOCK);
        while (*rwlock > RW_W_LOCK) {
            _mm_pause();
        }
    }

    // wait until all readers left
    while(*rwlock != RW_W_LOCK) {
        _mm_pause();
    }
}

void writeUnlock (volatile int* rwlock) {
    __sync_fetch_and_sub(rwlock, RW_W_LOCK);
}
```

Listing 4.1: Implementation of RW-locks using the atomic `__sync_fetch_and_add` and `__sync_fetch_and_sub` functions.

work in two phases: A writer trying to acquire the write lock will immediately get it as soon as no other writer has the lock. The then acquired write lock prevents new readers from acquiring a read lock for themselves. This prevents new readers from acquiring a lock, so the number of readers cannot increase.

In the context of B-trees, there is another factor that will prevent writer starvation in its entirety: Read locks are solely used for navigation in the tree. Furthermore, the locks are acquired in an interlocking pattern, so they are released shortly after they have been acquired. In sum, the time is a single read lock is acquired is very small, making it very unlikely that read locks will cause writer to starve even remotely.

On the other hand, modifying operations are generally rare enough—especially in a locally close part of the tree—that write locks likely will not result in reader-starvation either.

4.2.1. Interlocking read locks

The previously mentioned interlocking read locks are used for the navigational parts of the B-tree operations. They follow a rather straight-forward idea: The read lock is originally acquired on the root node before the navigation starts to descend. For each step then a read lock exists on the current node, guaranteeing safe access the node's fields. Once the next node that needs to be visit has been determined, a read lock is acquired on it. Afterwards, the read lock on the current node is released and the step to the next node is performed. Listing 4.2 shows this procedure for the search operation on a node after the index has been determined.

```
Node* childNode = (Node*) node->children[i];
readLock(&childNode->lock);
readUnlock(&node->lock);
return btree_search_node(childNode, key);
```

Listing 4.2: Interlocking read locks; the new read lock is acquired before the current one is released.

At the leaf-level the lock is then released immediately before the data is returned, after all data has been read from the leaf itself.

4.2.2. Insertion paths

The following section will detail the implementation of the presented locking strategy for the insertion operation. For the evaluation of this thesis, the focus was on the search and insert operations, which is why the implementation for the delete operation is not further covered.

As mentioned at the beginning of this chapter, the locking strategy works in multiple passes and utilizes a memory of the taken path and additional metadata. For the insertion process, the `Path` structure represents this memory.

```
typedef struct Path {
    int* path;
    int height;
    int lockFrom;
    bool valid;
} Path;
```

Listing 4.3: The `Path` structure.

The `path` field is an array of indexes which represents the path that was taken in the previous pass. The `height` is set to the tree height and is primarily used to detect changes at root level which cause the whole path to get invalidated. The `lockFrom` field stores the height from which write locks should be acquired based on the knowledge from the previous pass. Finally, the `valid` boolean stores whether or not the current path is still considered valid. All non-first passes are considered valid from the beginning and become invalid when a taken step does not match the path from the previous pass.

The insertion process begins with the high-level function where a read lock is acquired on the `BTree` structure to ensure that the data can be read safely. A new path is allocated and the first descent starts. During the navigation, each taken index is added to the `path` array to remember it for the future.

Once the process reaches the leaf, there are two things that can possibly happen: If the leaf would need to split, it only signalizes that to the parent call. Otherwise, if the leaf does not need to split up, the insertion is simply completed. This is possible because by design, during insertion leaf nodes are always acquired with a write lock instead of a read lock. As leaves are considerably small compared to the rest of the tree, and because they have no further children, blocking the leaf with a write lock is not a huge problem. In addition, the odds for the leaf to be able to complete without having to split up are quite good. The leaf marks the path as valid then signalizing that the insertion is completed, which then ends the process back at the root level.

Otherwise, if the leaf signalized a necessary split, then the node capacities are evaluated on the way back up. If a node is at full capacity after receiving the signal from the level below that a split is necessary, it also signalizes a split to the above level. Otherwise it sets `lockFrom` to the current level.

Once the execution reaches the root again, the path is evaluated. If it is marked as valid, then the insertion completed, so it can end the process. Otherwise, the next pass is started, this time with `valid` being preset to true. The process then descends in the tree again, verifying the step from the previous pass on every level. If the validation fails, then `valid` is immediately set to false to signalize that the path is no longer valid and that this cannot be the final pass. Otherwise, write locks are acquired from the level that is specified in `lockFrom`. If the leaf is reached while the path is still considered valid, then all required locks were acquired and the insertion process can run which then works identical to the original implementation described in chapter 2.

If the path is invalidated on the way down, then the current pass is downgraded to a dry run, doing the same things as the first pass. Write locks that were potentially acquired are again released on the way up without modifying anything.

The whole process will continue running pass after pass until a stored path stayed valid when it reached the leaf. In that case, the insertion process ends.

4.3. Evaluation

To evaluate the synchronized version of the implementation, the known test setup from chapter 3.1 was used. In a concurrent environment, individual tests are not helpful when all operations are the same and as such keep interfering with each other (when only running insertions), or simply ignore each other (when running searches). As such, the previously introduced sequence approach is used again to simulate a more realistic scenario. The sequences were generated with a 80% probability for searches, and 20% for inserts. In practice, an insert on every fifth action is still rather unlikely, but for the purpose of this evaluation, the value was chosen to explicitly trigger competing insert actions.

Although multiple threads run during the tests, only a single sequence is used. This ensures that every thread is working on the same data set, and that each thread will not just work in its own area somewhere in tree. To make this even clearer, the generated sequence is evenly distributed across the threads. For example with four threads, the first thread gets action 1, 5, 9 etc. That way, eventually existing dependencies from within the sequence file are respected.

When running the test program, it will first load the sequence completely into memory, and create a tree from the initialization actions. It then distributes the sequence evenly across a number of threads and then the threads are executed. The measured time only includes the time it took the threads to work off their part of the sequence. A wait lock is used to make sure that all threads start working at the same time, and once all threads completed their work, the timing ends.

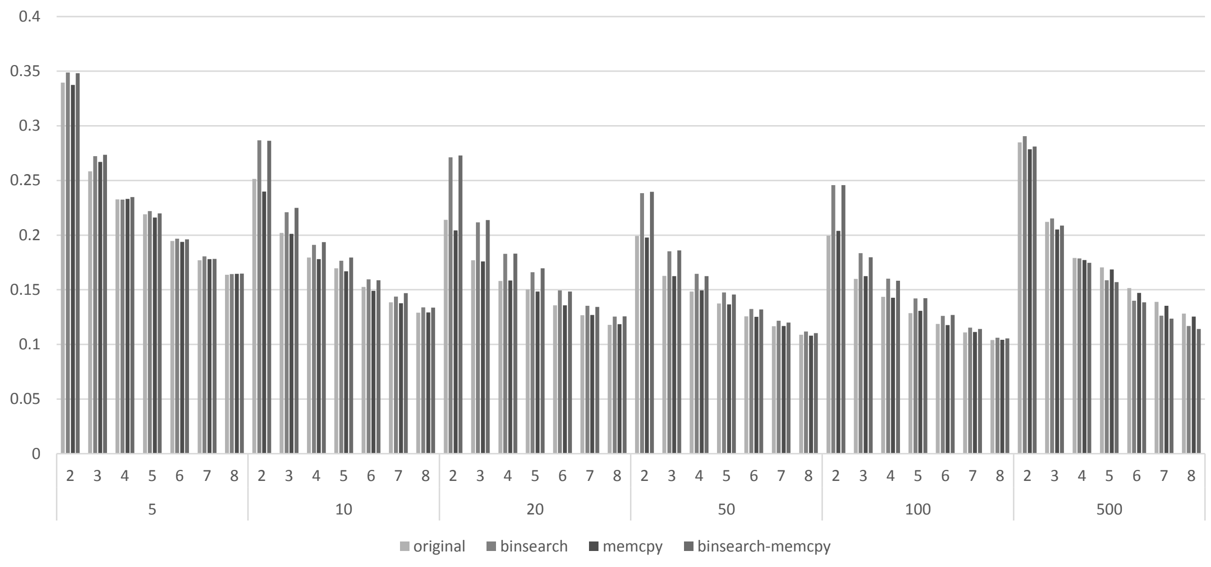

Figure 4.2.: Average execution time per action with multiple threads.

Figure 4.2 shows the average execution time for a single action on two to eight threads. For the sake of completeness, the different optimization options are included one more time, although they show the same pattern we already discovered in chapter 3.

For more than three threads we already got below 0.2 microseconds for a single action, which is a very good result given that the used sequence contained 100,000,000 actions, resulting in a very large tree. For comparison, a similar sized sequence was shown in figure 3.9 for which we had an average of about 0.35 microseconds per action using the single-threaded implementation.

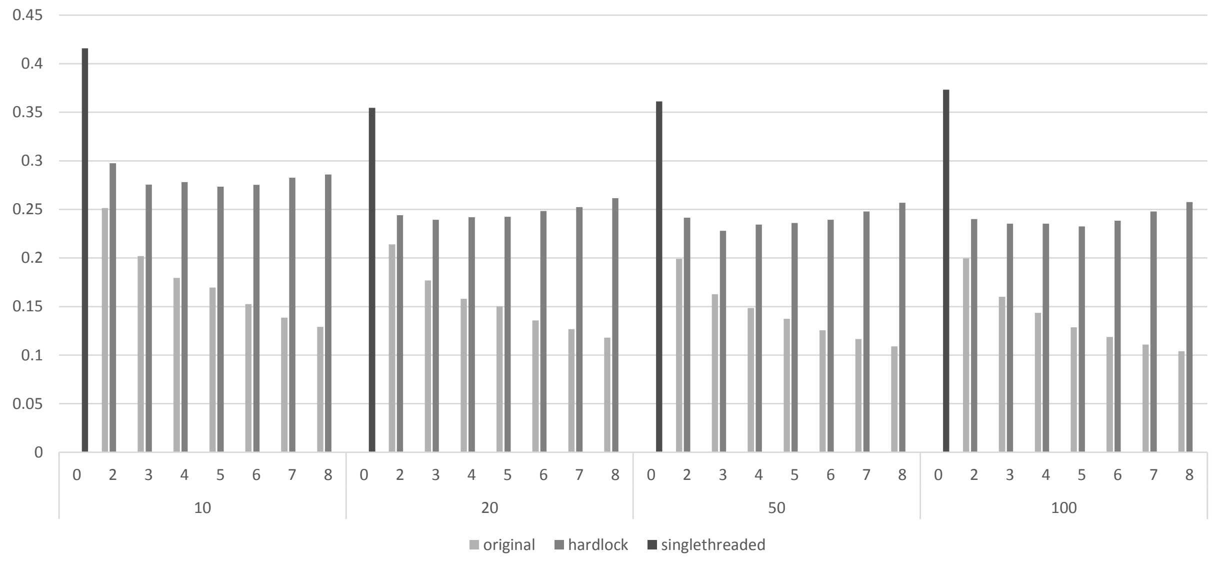

Figure 4.3.: Comparison of synchronization methods.

Figure 4.3 shows a comparison of the implemented synchronization strategy, simplistic "hard locking" and the single-threaded implementation for the same sequence.
The shown hard locking results are from an early version of the implementation that used simple tree-wide locks for the whole tree. As such, it is not surprising that the performance of that implementation degrades as the concurrency is increased, as the probability for simultaneous insertions and as such the demand for the write lock increases. It still performs better than the single-threaded implementation, as the concurrent search operations can still work concurrently.
Results for different sequence sizes are shown in figure 4.4. While the overall performance still degrades as the sequence increases in size, the effect gets negligible as the concurrency increases.

4.3.1. High concurrency test

In addition to the tests presented in the previous chapter, the tests were also run on a two-socket machine, allowing for a test with higher concurrency. The machine used was equipped with two Intel Xeon E5-2690 processors at 2.9 GHz, offering a total of 32 threads, and 64 GB memory.
For these tests, the same setup was used, except that the source code was compiled with the GCC compiler instead.
Figure 4.5 shows the results from the test. As we can see, the average execution time is higher than what we have seen in figure 4.4 for the other machine. A possible reason for

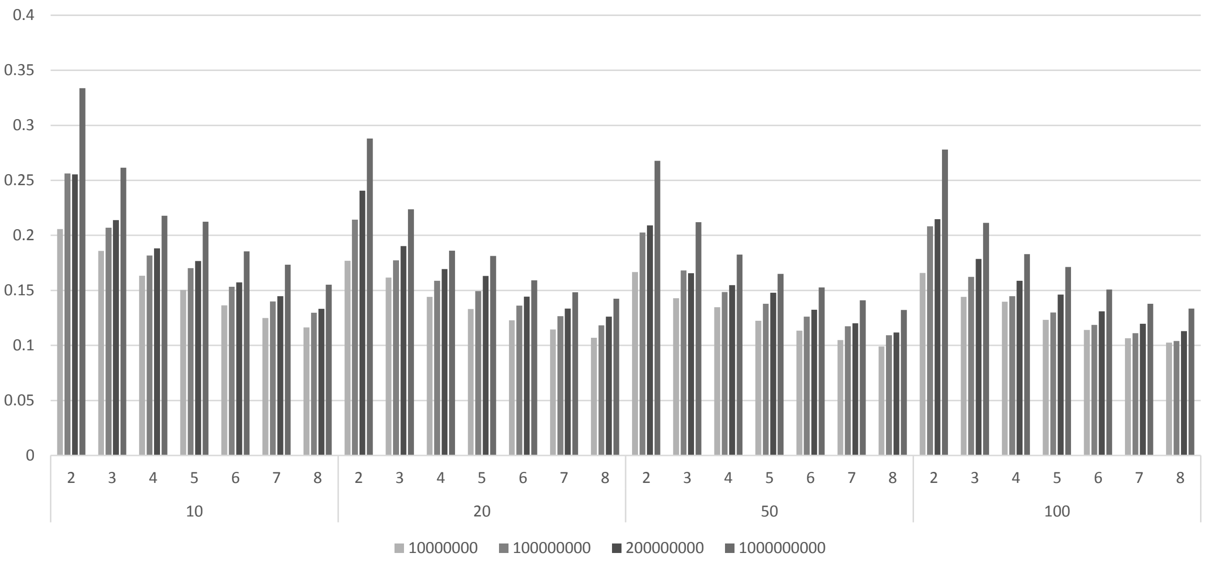

Figure 4.4.: Average execution time for different sequence sizes.

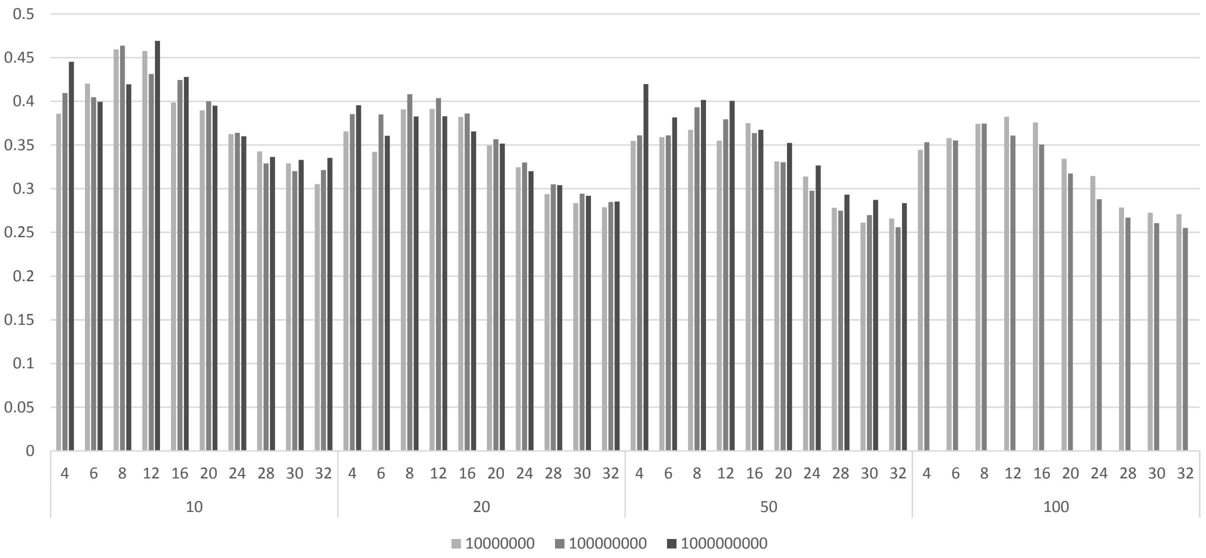

Figure 4.5.: Average execution time for different sequence sizes with a high level of concurrency.

this could be the slower processor that is running on this machine. Unfortunately, we do not have any single threaded results with which we could compare these values to see how much the improvement with the concurrent execution is.
However, we still reach average execution times below 0.3 microseconds even for very large tree sizes. We can also see that the performance continues to improve as the concurrency increases, showing that the implementation is able to work even with very high concurrency.

In conclusion, the tests on both machines showed that the presented synchronization strategy is performing really well, and is already able to reduce the average execution time per operation by half for just a small number of threads. The results also show that with an increasing concurrency, the performance is constantly increasing. That proves that more concurrent operations will not result in more blocking which would instead slow down the execution.

5. Conclusion

In this thesis, we started with a textbook implementation of a B-tree and evaluated it on modern hardware. The test results showed that our implementation was already performing very well, showing impressive average times for single operation of less than 0.5 microseconds. We tried out different ideas to further optimize the implementation but as it turned out, none of them were really that useful. The idea to replace the linear key search for a binary search added so much overhead, that the benefits it theoretically promised did not appear at all. The memcpy optimization idea was performing better but did not really distinguish itself from the original implementation, yielding very similar results.
The sequence tests showed that even with a load of mixed operations, the implementation shows a stable performance that even is not too much affected by greatly increasing tree sizes, remaining below 0.5 microseconds all the times.
Based on this implementation, we introduced a custom locking strategy for modifying operations. The operations work in multiple passes to first figure out what locks are required and then acquire the locks. The integrated validation protects against intermediate changes, restarting the process in case of conflicts.
In the evaluation of the synchronized version of the implementation, we could see that already two threads are able to reduce the load below impressive 0.2 microseconds. As the concurrency increases, this value can be reduced even more, and at no time, the concurrency seems to create a negative effect due to too much locking. This shows that the optimistic strategy we chose was indeed a good idea and is definitely able to improve the performance.
In conclusion, the presented implementation proved to perform really well and is definitely a good foundation to base future work on.

5.1. Future ideas

Because of the limited time for this thesis, many ideas could not be tried out. Given that all the optimization ideas we implemented turned out to be rather ineffective, it would be interesting to see how well the overflow slot mentioned in chapter 2.4 would perform. Given that throughout the tests we assumed deletes as rarer than inserts, the downside of only affecting insertions might not be so problematic. And given the number of branch mispredictions as shown in figure 3.5, there is definitely some room for improvement, so

reducing the overall complexity of the insert operation for the cost of another slot in nodes could result in a better performance.
One major idea was to use the transactional memory that is available with Intel's Haswell microprocessor architecture. In October 2013, GitHub user Austin Seipp published and example use of transactional memory as a GitHub Gist [10]. In his example, he adds transactional memory intrinsics around a traditional spinlock implementation and is able to elide locks almost completely, showing the capabilities of transactional memory as a means for synchronization. It would be interesting to see how the performance of the implementation improves with transactional memory, especially since the probability for actually blocking locks is rather low for B-trees.
Of course, the related work mentioned in chapter 1.2 do offer some ideas too. While they mostly focused different aspects, the core ideas are still applicable for our purpose.
Especially the ideas from the CSB+-trees introduced in [9] could be integrated well with the synchronized tree implementation of this thesis. As they were primarily focusing on cache efficiency for the structure itself, a combination with the synchronization results of this thesis could be promising.
The Bw-trees in [7] also showed very good ideas with their own way to manage synchronization. As they managed to go completely without locking, it would be interesting to see how their solution performs on current hardware, to compare it with the locking strategy of this thesis.
Ultimately, the ideas have shown that there is a lot potential for improvement for B-trees in the context of in-memory databases, and it will be interesting to see what will happen in the future.

A. Additional test results

The following chapter shows some additional test results that were previously not included. The results were omitted from the chapters 3 and 4.3 because they would not result in new findings.

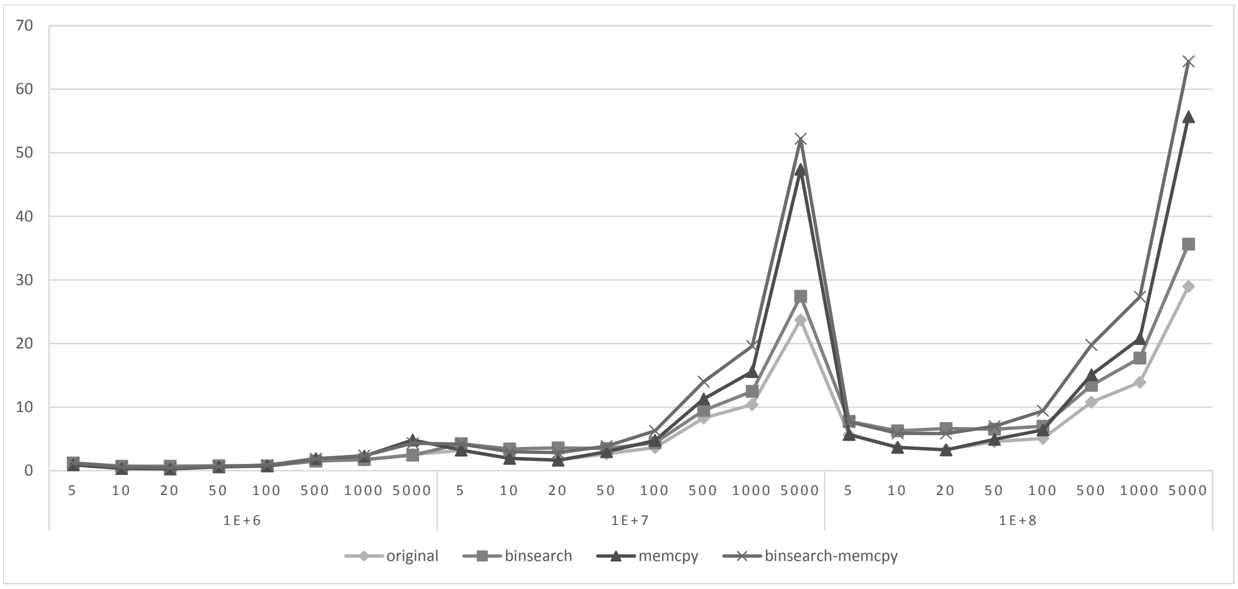

Figure A.1.: Average L3 cache misses per insert operation (with increasing number of total insertions).

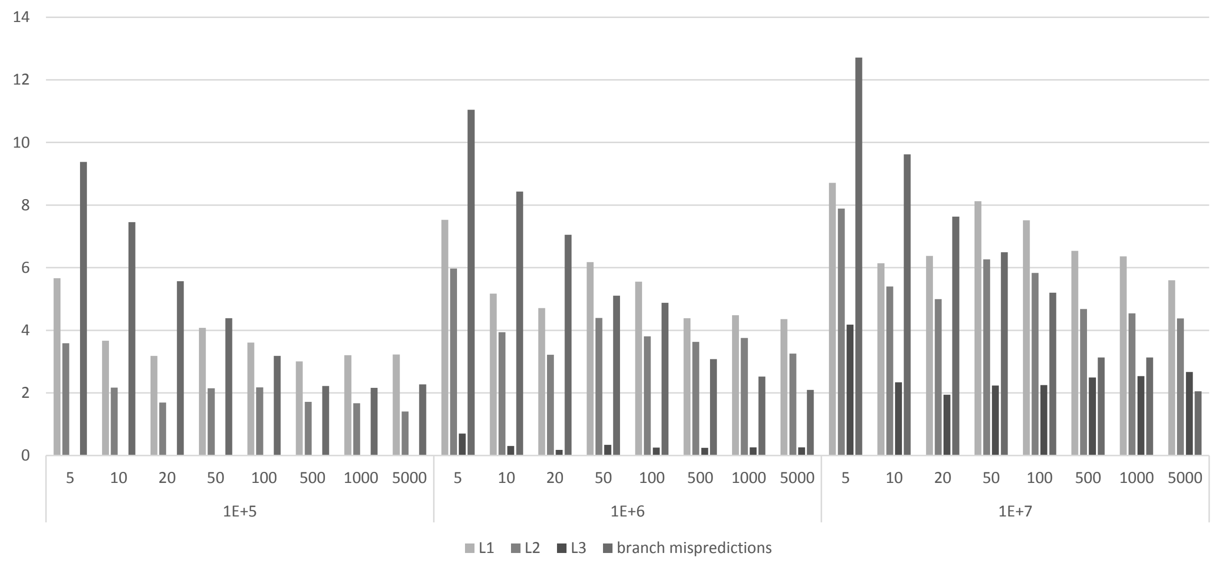

Figure A.2.: Average number of cache misses and branch mispredictions for a single search operation (with different tree orders and tree sizes).

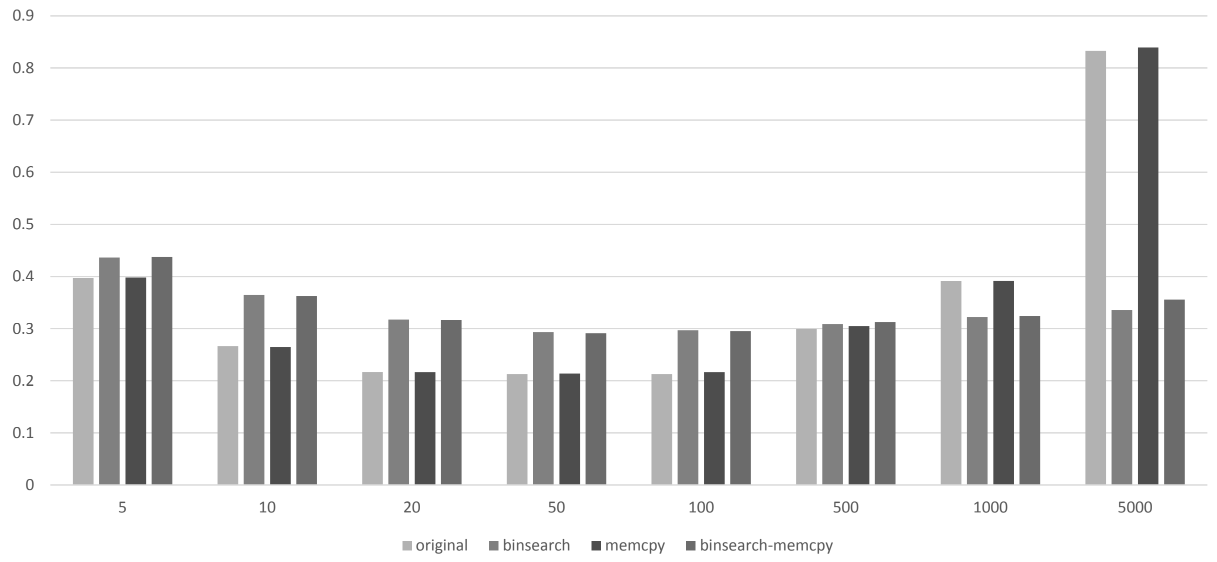

Figure A.3.: Comparison of optimizations for the delete operation.

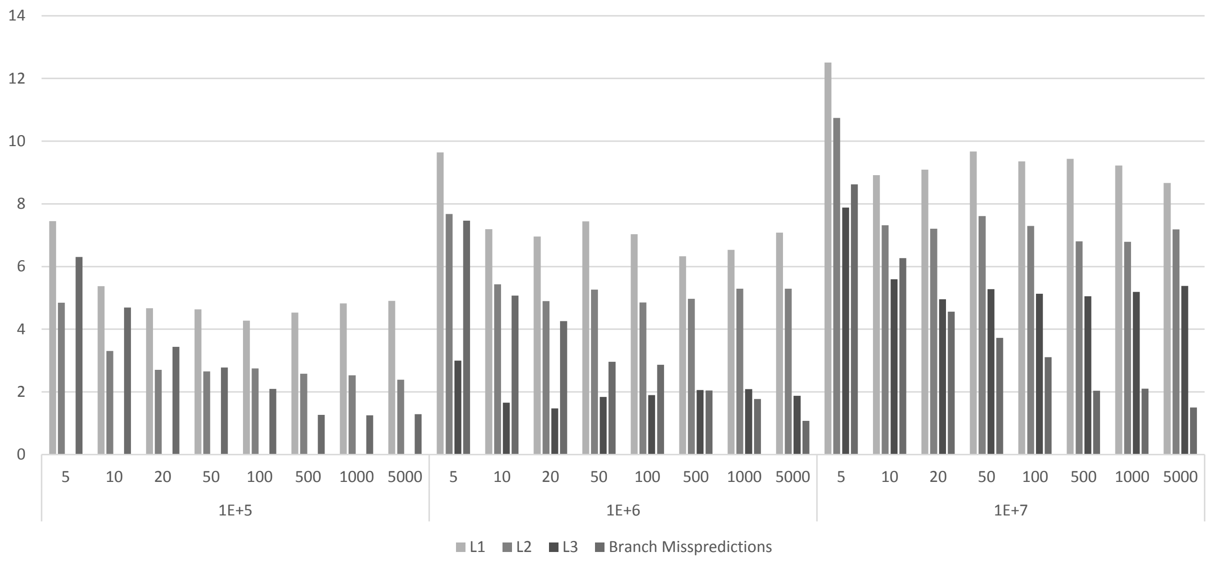

Figure A.4.: Average number of cache misses and branch mispredictions for a single delete operation (with different tree orders and tree sizes).

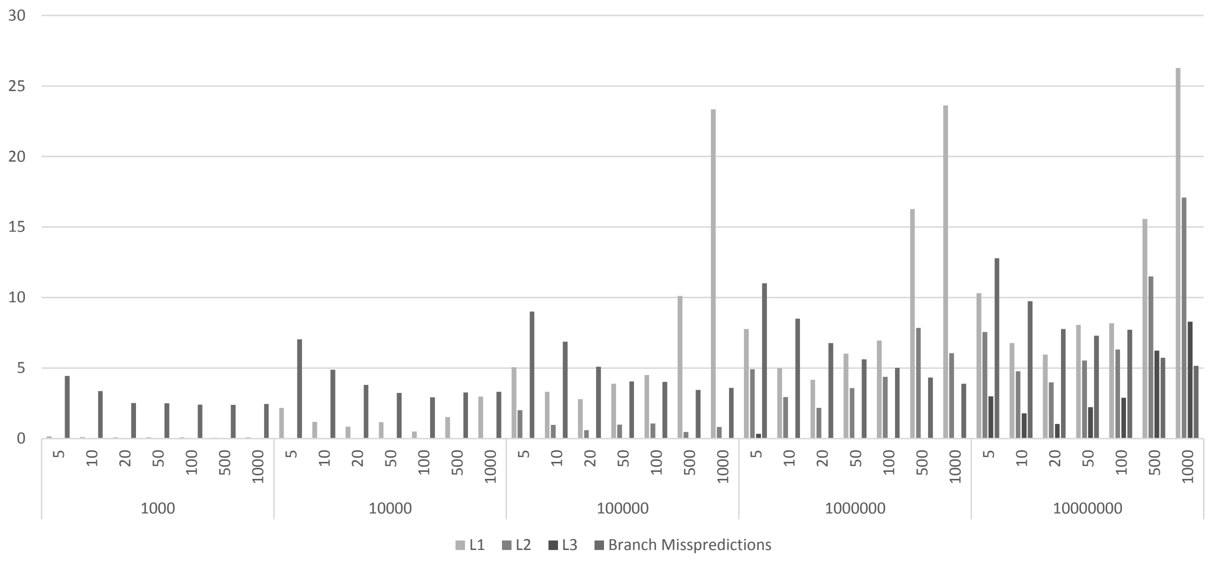

Figure A.5.: Average number of cache misses and branch mispredictions for a sequence (40% search, 40% insert, 20% delete).

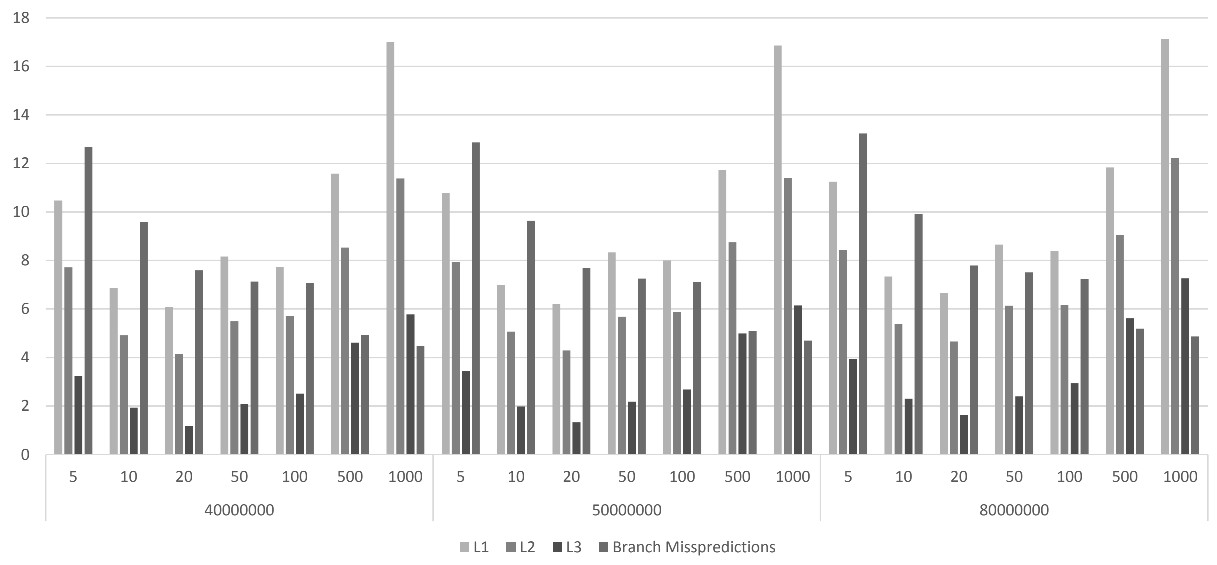

Figure A.6.: Average number of cache misses and branch mispredictions for a sequence (65% search, 25% insert, 10% delete).

B. Enclosed digital medium

The enclosed disk contains a digital version of this thesis, the single-threaded implementation of the B-tree and the updated implementation with synchronization. Both versions include the available optimizations behind compiler flags, and they also come with the programs used to run the tests.
In addition, the final test results presented in this thesis are included as a browsable Microsoft Excel file.

List of Figures

2.1. B-tree structure 7
2.2. Node split 10

3.1. Insert: Different tree orders 19
3.2. Insert: Comparison of optimizations 20
3.3. Insert: L1 cache misses 21
3.4. Insert: L2 cache misses 21
3.5. Insert: Branch mispredictions 22
3.6. Search: Average execution time 23
3.7. Delete: Average execution time 24
3.8. Sequence (40/40/20 probabilities) 24
3.9. Sequence (65/25/10 probabilities) 25

4.1. Interlocking read locks 28
4.2. Concurrent evaluation: Average execution time per action 33
4.3. Concurrent evaluation: Comparison of synchronization methods 34
4.4. Concurrent evaluation: Increasing sequence sizes 35
4.5. Concurrent evaluation: High concurrency test 35

A.1. Insert: L3 cache misses 39
A.2. Search: Cache misses and branch mispredictions 40
A.3. Delete: Comparison of optimizations 40
A.4. Delete: Cache misses and branch mispredictions 41
A.5. Sequence (40/40/20 probabilities): Cache misses and branch mispredictions 41
A.6. Sequence (65/25/10 probabilities): Cache misses and branch mispredictions 42

Listings

2.1. Node and Leaf structures . . . 8
2.2. BTree structure . . . 8
2.3. Linear key search . . . 9
2.4. Main tree operations . . . 9
2.5. Search key equality check . . . 10
2.6. Other tree operations . . . 12
2.7. Binary key search . . . 13
2.8. Insertion complexity . . . 14
2.9. Insertion complexity with overflow slots . . . 15

4.1. Readers-writer lock implementation . . . 30
4.2. Interlocking read locks in search . . . 31
4.3. Insert path structure . . . 31

Bibliography

[1] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, January 1990.

[2] Rudolf Bayer and E. McCreight. Organization and maintenance of large ordered indexes. In Manfred Broy and Ernst Denert, editors, *Software Pioneers*, pages 245–262. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[3] ISO/IEC 9899:1999; the C programming language standard. `http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=29237`, 1999. [Online; accessed 5 March 2014].

[4] 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*, SIGMOD '02, pages 157–168, New York, NY, USA, 2002. ACM.

[5] Douglas Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[6] Db-engines ranking of key-value stores. `http://db-engines.com/en/ranking/key-value+store`, 2014. [Online; accessed 5 March 2014].

[7] David B. Lomet, Sudipta Sengupta, and Justin J. Levandoski. The bw-tree: A b-tree for new hardware platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 302–313, Washington, DC, USA, 2013. IEEE Computer Society.

[8] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, third edition edition, 2002.

[9] 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*, SIGMOD '00, pages 475–486, New York, NY, USA, 2000. ACM.

[10] Austin Seipp. Transactional memory on haswell. GitHub Gist, `https://gist.github.com/thoughtpolice/7123036`, 2014. [Online; accessed 5 March 2014].