technische universität
dortmund

Masterarbeit

Using Modern Synchronization Mechanisms in Databases

Mark Schröder

April 2016

Gutachter:
Prof. Dr. Jens Teubner
Michael Kußmann

Technische Universität Dortmund
Fakultät für Informatik
Datenbanken und Informationssysteme (6)
http://dbis.cs.tu-dortmund.de

Contents

Chapter 1

Introduction

With ever-increasing amounts of main memory available to databases, large parts of their data structures can be stored there, including the indices. This is reflected by the growing number of major, commercial, in-memory database management systems like SAP HANA, IBM Blu Acceleration, Oracle Database In-Memory, and Microsoft Hekaton. As early as 1999, components of modern CPUs and memory have been identified as viable areas to optimize database management systems towards [Ail+99]. Today, adapting indices for in-memory databases is an area of active research, with new designs like *Adaptive Radix Trees* being developed only recently [LKN13].

Updating data files and index structures in OLTP (*On-Line Transaction Processing*) scenarios requires a significant synchronization overhead of using locks and latches. In case of B-tree indices, several different synchronization mechanisms have been developed to allow for as much concurrent operations on the database as possible. Besides fine-grained locking schemes like lock coupling, lock-free data structures have been proposed. In addition, CPU makers have introduced hardware transactional memory. In this short introduction, we will describe why concurrency is an important issue in databases and what synchronization mechanisms exist to tackle this issue. We will then present our goals with regard to these two issues and outline the structure of the thesis.

1.1 Concurrency and Synchronization in Databases

There are many different areas in which databases benefit from having multiple processes or threads running concurrently. In order to process transactions in parallel, not only do accesses to the data files themselves have to be synchronized. Also shared

internal data structures like indices have to be searched and modified in a thread-safe way.

To this end, there are multiple synchronization mechanisms employed in databases. *Locks* are used to synchronize transactions and help to provide the known guarantees of atomicity, consistency, isolation and durability. For internal data structures, *latches* have traditionally been used for synchronization. The driving demand for concurrency and the unique challenges of in-memory databases have lead to a plethora of algorithms and implementations of latches:

- When it comes to implementing latches as simple spinlocks, the goal is to reduce the cost of waiting for a latch. To reduce traffic on the memory bus in multi-core and multi-processor systems, the *test-and-set* approach has been refined into the *test-and-test-and-set* approach. On x86 systems, several different atomic instructions may be used to implement these approaches, each requiring different numbers of CPU cycles. On top of these hardware primitives for latch acquisition, backoff strategies are used to further reduce the amount of CPU time spent spinning.

- In addition to simple exclusive latches, more elaborate latch types have been developed. *Shared/exclusive* latches (also called *read/write* latches) allow a distinction to be made between threads which only read data and threads which might also modify data. In this way either multiple reading threads or a single modifying thread can access the data. Even further refinements allow *upgrading* an already-held shared latch to an exclusive latch.

- Fine-grained latching schemes like latch coupling in B-trees have been developed to facilitate locking as few nodes as possible, thereby reducing latch contention.

- Moreover, B-trees have been improved towards less reliance on latches by using more optimistic latching schemes. B^{link}-trees require at most one latch to be held at all times, by navigating down the tree optimistically. The OLFIT (*optimistic, latch-free index traversal*) algorithm allows to even read the contents of nodes optimistically without latches, at the risk of having to repeat the read until no more modifications are made in the meantime.

- Recently, entirely lock-free data structures like Bw-trees have been introduced. They combine optimistic execution of operations with atomic compare-and-swap instructions to allow even modifications to be attempted concurrently.

- Transactional memory, implemented in both hardware and software, has long been studied as a way to implement lock-free data structures without having to develop intricate algorithms to guarantee thread safety. In recent years, especially the hardware-supported implementations by Intel and others promise to solve these issues while delivering competitive performance.

1.2 Goals of the Thesis

The overarching goal of this thesis is to evaluate the effects that Intel's implementation of hardware transactional memory can have on the performance of B-tree operations. The feature comprises two sets of instruction set extensions, *Hardware Lock Elision* (HLE) and *Restricted Transactional Memory* (RTM). The latter is the more interesting one, as it permits to explicitly mark the beginning and end of a transactional region of code, and to handle transactional aborts in an application-specific way. Possible performance improvements achieved by using Intel's *Transactional Synchronization Extensions* (TSX) over existing B-tree designs should be measured using realistic inputs.

1. First, a textbook implementation of a B-tree, which includes latches, is to be created. Based on that, refined implementations like cache-sensitive B-trees (CSB-trees) [RR00], B^{link}-trees [LY81] and the OLFIT algorithm [Cha+01] are made. The trees should be able to store integer keys and values, and provide search, insert, and delete functions.

2. Measuring performance requires a benchmark. Our benchmark will consist of a small program written in C, which parses files containing lists of operations, starts multiple threads to execute these operations and measures time consumption and throughput of operations. The files containing the operations are created with a Python script.

3. The main goal is to use Intel's RTM instructions to improve performance of B-tree operations. This means identifying memory access operations that span small enough of an address range to be executed atomically by the CPU, resulting in a carefully implemented B-tree variant with competitive performance.

1.3 Structure of the Thesis

The thesis is organized as follows:

B-trees are the index data structure used in database management systems that this thesis will focus on. The second chapter contains a summary of what B-trees are, and the several operations that they support. It will also cover different variations of B-trees, namely B^{link}-trees, cache-sensitive B-trees, Bw-trees, and the OLFIT algorithm. Implementations of some of these B-tree variants will later be benchmarked.

The third chapter gives a short survey of transactional memory techniques and implementations.

The fourth chapter describes the *Transactional Synchronization Extensions* in detail. Along with descriptions of the two components, *Hardware Lock Elision* and *Restricted Transactional Memory*, we will provide code examples and microbenchmarks intended to make the usage and limitations of TSX more palpable.

In the fifth chapter we will describe our own B-tree implementation that uses TSX for increased performance. We will call it the *Bx-tree*.

The sixth chapter contains benchmarks comparing the performance of different types of B-trees, all implemented in C. Areas of focus will be a first look at different B-trees using latches, a comparison of those trees with implementations where latches are elided using Hardware Lock Elision, and finally an analysis of the Bx-tree.

The seventh and final chapter contains a review of the work done and an outlook on prospective research.

Chapter 2

B-Trees

In this chapter we will give a short summary of what B-Trees are. We will begin with an overview of the general layout of B-Trees and a distinction between the various definitions used in literature. After that, we will describe the common operations of searching, inserting, deleting, and updating data, that a B-tree supports. Then we will present issues and solutions regarding concurrency in B-trees. Finally, we will present variations of the original idea of B-trees that should be better suited to multi-threaded scenarios and modern hardware. These are the B^{link}-tree (Section 2.4), the Bw-tree (Section 2.5), the CSB-tree (Section 2.6), and the OLFIT algorithm (Section 2.7).

B-trees have originally been described in [BM72]. They are used in many different databases, from SQL databases like PostgreSQL [PSQL15] to NoSQL storage engines liked WiredTiger [Wik15].

2.1 B-Trees, B^+-Trees and B^*-Trees

B-trees are data structures that can be used for indexing arbitrary data. In databases in particular, they are the basis for *indices* that map keys of different data types to *record identifiers* (RIDs). There are two types of nodes in B-trees: regular nodes (which include the root and inner nodes) and leaves. The leaves of a tree contain every key and its associated value, for example the RID. Inner nodes serve to guide the search from the root to the leaves, as in a binary search tree. In fact, B-trees can be seen as a generalization of binary search trees: an inner node of degree d contains d keys and $d + 1$ pointers to child nodes (Figure 2.1).

The word B-tree is used in literature both as the name for a specific data structure, the original B-tree, and as the name for a class of variants thereof. Usually, the following distinction is made:

B-tree The original B-tree, described in [BM72], stores values in all nodes and is thus a true generalization of a binary search tree.

B⁺-tree It stores values in the leaves only [Com79]. In addition, the leaves are connected to each other in the form of a linked list, allowing efficient sequential access to the values for use in range queries.

B*-tree It is similar to a B⁺-tree, but requires nodes to be at least $\frac{2}{3}$ full, making inserts and deletes a bit more complicated.

In this thesis we will focus on B⁺-trees, but refer to them as B-trees, as is customary in literature.

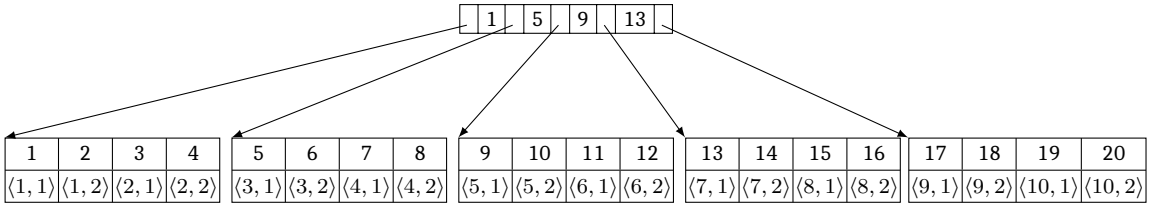


Figure 2.1: A B-Tree with $d = 4$. The upper node is an inner node with keys and pointers; the lower nodes are leaves containing keys and record identifiers. RIDs are represented as pairs of a database page number and an offset.

2.2 Operations in B-Trees

Maintaining an index requires support for several operations, namely search, insert, delete, and update. In the following section we will briefly describe these operations and challenges that are faced when implementing them.

2.2.1 Search

Search consists of identifying the correct branch to descend down in an inner node and identifying the correct value in a leaf. Two types of search that can be solved efficiently are searching for the value that a single key maps to and searching for the values mapped by a range of keys, because leaves in B-trees are usually connected

with their siblings to allow sequential scans. Searching for a key inside a node is discussed further in Subsection 2.2.5.

2.2.2 Insert

Insertion consists of searching for the place where a new key needs to be inserted and then doing the actual inserting. Inserting a value into a leaf may require splitting nodes up until the root. In general, the insertion process entails two costly operations: inserting an entry into a node, which means moving the keys and pointers or RIDs from the insertion point to the right; and splitting a node, which means copying the upper half of the keys to a new node. Splitting also requires inserting a pointer to the newly created node into the parent of both nodes, next to the pointer to the old node.

2.2.3 Delete

Deletes are handled in one of several ways:

- An entry in the leaves can be marked as deleted, but remain in the tree. This allows to later undo the delete in case a transaction needs to be aborted.
- The entry can be physically deleted from the page, freeing up space in the node.
- The entry can be physically deleted from the page, and in case the page *underflows*, i.e., becomes less than 50 % full, the tree structure can be restored. Depending on the state of adjacent nodes, this can be handled in two ways:
  - Keys from a sufficiently full neighbor may be copied over.
  - The node's contents may be merged into its neighbors if there is enough room, allowing the node to be deallocated.

2.2.4 Update

An update is necessary when the mapping between keys and values changes. In the context of databases, this occurs, for example, when the value in an indexed column of a tuple changes, i.e., a RID becomes associated with a new key. Performing an update can be handled by deleting the obsolete entry and inserting a new entry.

2.2.5 Search inside Nodes

Searching for a key inside a B-tree node presents an interesting choice: as the keys are sorted, efficient search algorithms like binary search may be employed. However, there is an important difference between searching in leaves vs. inner nodes and in searching vs. inserting:

- When searching for a key inside a leaf, it only matters whether the key exists or not, and, in the former case, at what position. This is the classical definition of the search problem, for which binary search is a solution.
- When trying to insert a key into a node, and if the key does not already exist, the insertion point must be found. This is sometimes also defined as the problem of the lower bound, that is, to determine the position of the left-most key which is equal to or greater than the search key.
- When traversing the tree, finding the correct child pointer to follow requires searching for the rightmost key that is equal to or less than the search key. If such a key does not exist, the leftmost child pointer is followed.

In addition to these slightly different problem definitions, there exists also a plenitude of solutions to the search problem, that may or may not be optimized for specific cases or certain hardware:

- The slowest solution is certainly linear search.
- Even optimized variations of linear search, for example those using SIMD[1] instructions, may provide the optimal performance in some cases.
- Binary search is the obvious solution.
- Ternary search is sometimes discussed as an alternative to binary search that requires fewer iterations or recursive calls. However, it requires provably more comparisons than binary search.
- Exponential search probes the array from left to right with exponentially increasing steps. When a lower and upper bound for the index have been established, another search algorithm (usually binary search) is used to process the remaining interval. Exponential search is especially suited for situations where the search key is expected to appear near the beginning of the array.

[1] *Single-Instruction-Multiple-Data*-instructions, for allowing greater parallelism

- Interpolation search can be viewed as a generalization of binary search: a pivot element partitions the current range into two remaining search spaces. However, the pivot element is not chosen as the middle of the search space. Instead, the keys are assumed to be distributed evenly and the pivot element is chosen based on the value of the search key. This allows large performance increases when the keys are in fact evenly distributed, but in the worst-case the algorithm may degrade to linear search.

Because the choice of search algorithm is not obvious and depends on the size of the array to be searched, we implemented a microbenchmark comparing some of these solutions.

2.2.6 Microbenchmark: Search inside Nodes

In order to find optimal search algorithms for the B-trees implemented in this thesis, we implemented and evaluated the following approaches:

- Forward linear search
- Backward linear search
- Forward linear search with a manually unrolled loop (four iterations in one)
- Forward linear search using AVX2 instructions[2] to compare four keys in parallel
- Binary search with two comparisons per iteration, allowing early return when the exact key is found
- Binary search with one comparison per iteration

In Figure 2.2 we present the results of the benchmark. The task consisted of searching in an array of the given size. The time is the average time per search, measured over 4,000,000 searches. The code has been compiled with GCC[3] at optimization level -O2. The level has been chosen as a compromise for the following reasons:

- Without optimization, GCC does not inline functions, adding the function call overhead to every search.

[2] Advanced Vector Extensions 2, an instruction set extension containing SIMD instructions for x86 CPUs

[3] GNU Compiler Collection

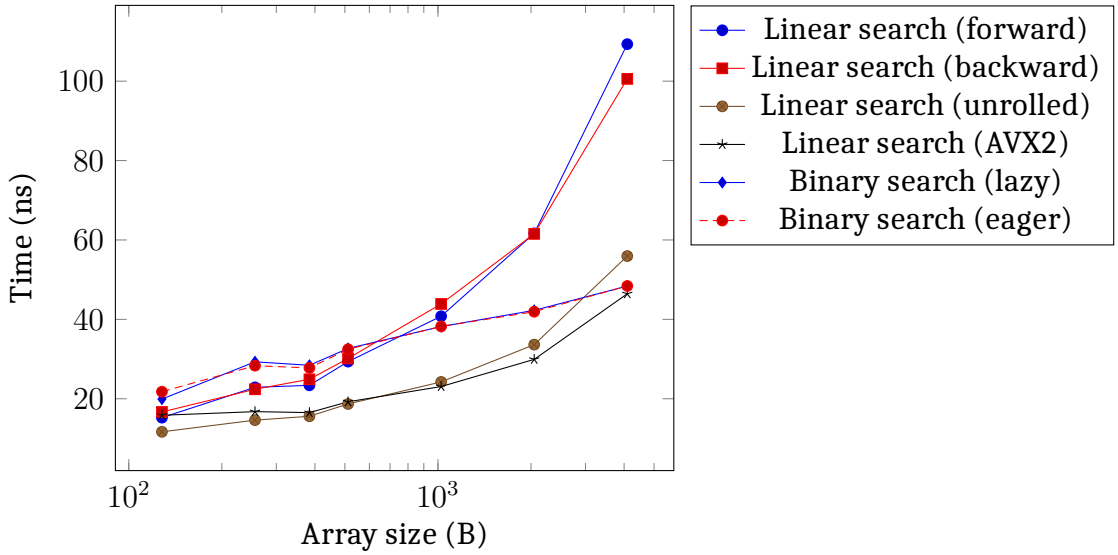


Figure 2.2: Search performance with -O2

- At optimization level -O3, GCC automatically uses SIMD instructions like AVX2 for loop, making the differences between the several implementations less apparent.

The benchmark shows that unrolling loops and using SIMD instructions provides a valuable speedup compared to naive linear search. Only at larger array sizes can binary search catch up from the overhead added by the more complicated code structure and random memory access patterns.

For our B-tree implementations however, we have chosen binary search, because at the highest optimization level it outperforms all other searches all the time, as GCC presumably removes much of the overhead and uses SIMD instructions as well.

2.3 Concurrency in B-Trees

To increase performance, it is desirable to grant multiple processes and/or threads access to shared data structures in a database at the same time. In this regard, a distinction is made between the synchronization of transactions and synchronizing parallel accesses to index structures. Both cases are handled by using locks to provide for mutual exclusion (except when usage of lock-free data structures is possible, see Section 2.5), but in the case of internal data structures like B-trees, locks are more often than not referred to as *latches* (see Subsection 2.3.1). This makes the different purposes more evident, but also reflects the fact that latches are usually implemented in a more lightweight fashion. Here are some more noteworthy differences:

- Locks control access to the logical contents of a database. This can mean objects stored in the database, like single records or whole tables, but also "concepts" like a lock on a range of keys, preventing for example inserting records with a key inside that range. Latches control access to the physical contents like the memory or disk pages on which data is stored.

- Locks are often stored inside a single, shared data structure, accesses to which have to be synchronized as well. It may even be necessary that ownership of a lock survive a crash, in order for transactions to be rolled back after the crash. Latches are rather stored locally with the data that they guard, for example they can reside on the page that they grant access to, while that page is loaded in memory.

2.3.1 Latches

Latches are lightweight locks. They are usually implemented as *spinlocks*, meaning a thread continuously checks in a loop whether the latch is free or not, and attempts to acquire the latch as soon as it becomes free. This works well for latches that are held for a short amount of time, and when no more threads exist in the system than can run on the hardware at the same time. If a thread holding a spinlock is suspended, all other threads waiting for the latch have to spin in the loop for at least as long as it takes for the latch owner to be scheduled again. Different techniques to balance the scalability of operating system mutexes and the fast grant of access of spinlocks, taking into account system load, have been studied in [Joh+10].

2.3.2 Latch Coupling

There are different ways to synchronize multiple threads operating in a B-tree, ensuring that every thread sees only a consistent view of the B-tree at any time.

- Certainly the most naive approach is to use a single global latch, ensuring only one thread can access the tree at any time. This leads to serial execution of all operations and is certainly the baseline when it comes to performance.

- A more fine-grained scheme is to lock nodes from the root downwards and release locks that are no longer required. Acquiring locks in a deterministic order is a typical solution to avoid deadlocks. This locking scheme naturally leads to *latch coupling*.

In order to reduce lock contention, latch coupling has been introduced as a way to avoid more coarse-grained locking while still providing freedom from deadlocks. In fact, it has been proven that latch coupling with latches stored inside the nodes is deadlock-free [WV01, p. 340].

2.3.3 Latches and Splits

When inserting a value into a tree, latch coupling is not enough to guarantee thread safety. Latch coupling avoids deadlocks because all locks are acquired in the same order. When a split causes inserts to be propagated up the tree, the latches can obviously not be acquired from the bottom up. There are however several ways to overcome this:

- While readers use latch coupling, writers may lock all possibly affected nodes from the root down. This way, a writer can start locking its path while multiple readers are still navigating through the tree.
- When using shared/exclusive latches, writers may use *upgrade latches* from the root down. This way, readers can still begin a search while a writer is already navigating through the tree, however other writers are locked out. When an insertion takes places, the latches have to be upgraded to exclusive locks.
- Writers may hold latches beginning at the lowest node that is safe from being split, i.e. the lowest (inner) node that is not full.
- Splitting proactively, meaning that an inserting thread encountering a full node on its descent splits it, allowing the latch on its parent node to be released.

2.3.4 Microbenchmark: Latch Implementations

In order to find the optimal lock for use in the later B-tree implementations, we implemented a microbenchmark to compare the overhead of different lock implementations.

The benchmark lets multiple threads execute a binary search as described in Subsection 2.2.6 to search 64-bit integer values in a shared array with a total size of 4096 KiB[4]. Each thread obtains the keys to search for from another array of variable length. This array is also shared and has an associated current index which points to the next key to search for. The array to be searched is protected by a latch/lock, with

[4] 1 KiB = 1024 B

the implementation of said latch being the code that is benchmarked. The latch is technically not necessary because the threads only search the array and do not write into it, but this has two beneficial side-effects:

- It simulates the search inside a B-tree node that is performed during tree traversal.
- It allows a first analysis of Intel's transactional memory implementation (see Chapter 4) in an optimal case, namely executing read-only transactions.

As the threads search for the values contained in the second array, the current position in that array has to be synchronized. Both arrays could be protected by the same latch, but in order to focus on the binary search aspect we opted to use an atomic *fetch-and-add* instruction: each thread reads the current index and atomically increments the index for the other threads, until the index reaches the end of the array.

The overhead of the respective latch implementations is difficult to measure exactly, so we looked at the total time as a metric, because the amount of work performed is the same for all benchmark runs, only the lock implementations differ.

We have implemented latches in the following ways:

- A spinlock using the C type `atomic_flag`.
- A spinlock where the state is stored in a variable of the C type `atomic_bool`.
- A custom spinlock written in assembly, using Intel's *Hardware Lock Elision* (see Section 4.2).
- The tried and trusted spinlock of Pthreads, `pthread_spinlock_t`.
- A heavy-weight lock, a mutex (`pthread_mutex_t`). This one is included to show the difference between spinlocks and locks that suspend threads.

The result of these experiments can be seen in Figure 2.3.

The most important takeaways are the following:

- Using the mutex is much more expensive with multiple threads. However, the performance does not degrade with more threads. The single hike at two threads is the cost of waking up one of the waiting threads, which does not increase when more than two threads are waiting.

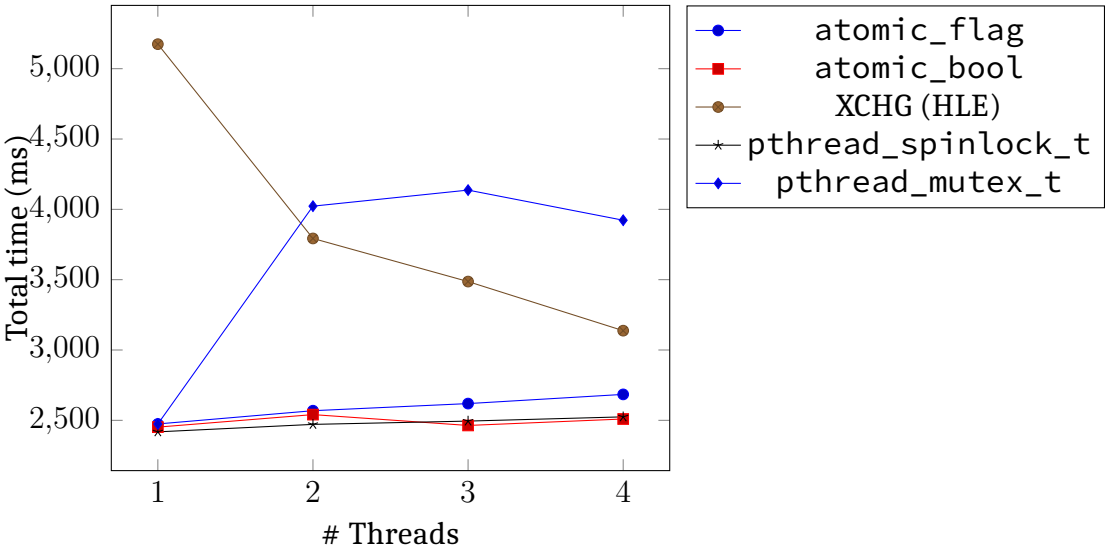


Figure 2.3: Comparison of different (spin)locks

- *Hardware Lock Elision* is costly, but it is the only synchronization mechanism that favors more threads, as they can run and perform work at the same time. However, the amount of work done inside the locked region (the binary search) is not enough to justify the cost attached to using transactional memory.

- The `atomic_flag` has slightly more overhead attached to it than the other spinlocks, possibly because we have used only its *test-and-set* instruction to acquire the lock, whereas with the `atomic_bool` we have employed a *test-and-test-and-set* scheme.

2.4 B^{link}-Trees

The main motivation for B^{link}-trees is to allow a more scalable locking scheme than latch coupling to be used. B^{link}-trees allow all operations to be performed without ever holding more than one latch at the same time. To achieve that, every node also stores a *high key* and a pointer to its right sibling. The high key is equal to the key in the parent node that points to the right sibling, i.e., any entry with a key smaller than the high key would be stored in the current node. The insertion algorithm is modified as follows:

1. The correct leaf is identified and locked.

2. The new entry is inserted.

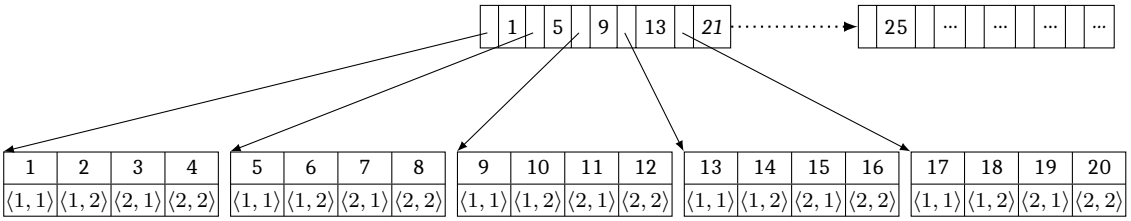


Figure 2.4: B^{link}-Tree with $d = 4$

3. In case of an overflow, a new right sibling is allocated and filled in the usual way.

4. The pointer to the right sibling and the high key are set. Thus, in the case of a split, the newly created right sibling is accessible before the respective pointer is inserted in the parent. At this point, the tree is in a consistent state again and the current node is unlocked.

5. A pointer to the right sibling may be inserted into the parent now or any time later.

2.5 Bw-Trees

Bw-trees [LLS13] have been developed mainly to improve the potential for concurrency. They are a variation of B^{link}-trees with a similar logical structure, however the actual implementation has two major differences:

- Nodes do not store pointers to other nodes. They store *logical page identifiers*. These page identifiers map to the nodes' physical location in memory through a mapping table.

- Page identifiers do not necessarily resolve to complete nodes, but to delta information that describe the changes made to a node. Every delta entry, which might be an insert or delete, points to the previous change, in the form of a chain, until finally a delta entry points to the actual node.

This has a number of consequences for the way searches and modifications are performed:

- Searches have to access each node through its page identifier, locating the node through an entry in the mapping table. Searching inside a node may require navigating through a lengthy chain of deltas until a more efficient binary search can be performed inside the actual node.

- Inserts can potentially do their work without looking at a complete node, as the information about the insert can just be prepended to the delta chain.

- In order to maintain acceptable performance, from time to time the changes encoded in the delta chain need to be applied to a node. This is called *consolidating* the node. Because the point of the delta chain is to allow updates to be performed without acquiring a latch, consolidation can not be done in-place. Instead, a new node with the changes applied is produced. This new node is then inserted into the mapping table using the same logical page identifier as the old node, replacing it. Similar to B^{link}-trees, a split is not performed by locking multiple nodes, but by making the new sibling accessible early through its own page identifier from the split node.

- Changing the mapping in the table is performed atomically with compare-and-swap (CAS) instructions. Thus, the design is entirely lock-free. It is, however, an optimistic approach: while a thread attempts to install the result of a lengthy consolidation into the table, another thread might cause the CAS to fail, meaning the work has to be discarded.

- Because consolidation is performed by creating a new node and installing it in the mapping table, the old node should at some point be handed over to a *garbage collector* to be reused or to have its memory freed. Because a thread might still be performing a search inside the node, however, the concept of *epochs* has been developed: every thread and node is considered part of an epoch and nodes are only garbage-collected, when no thread of the same epoch is running.

2.6 Cache-Sensitive B-Trees

To better utilize the available space in inner nodes, *cache-sensitive* B-trees (CSB-trees) [RR00] were developed. Instead of storing pairs of keys and pointers to child nodes, a node contains only keys and a single pointer to all children, which are placed contiguously in memory. The main motivation of CSB-trees is to allow more efficient search inside nodes, which benefits search operations as well as inserts and deletes. The changed memory layout however comes with a number of drawbacks:

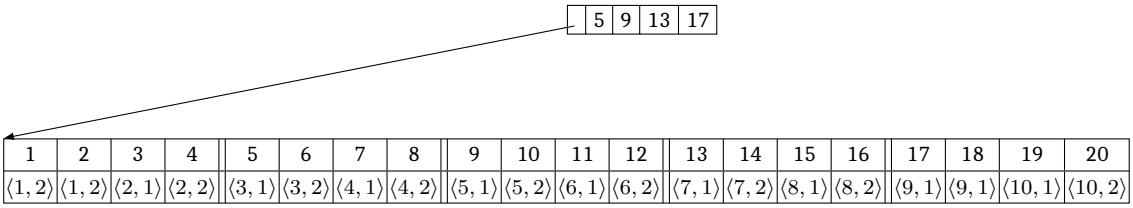


Figure 2.5: CSB-Tree with $d = 4$

- Inserting an entry into an inner node is more costly, as instead of moving pointers to children to the right, all the child nodes right of the insertion point have to be moved.

- Splitting an inner node is more costly as well, because instead of keys and pointers, keys and half of the child nodes have to be moved to the new right sibling.

- The locking scheme is more complicated, because modifying a node (inserting an entry or splitting) means that at least some of the child nodes have to be locked during the operation, as they may have to be moved around in memory. This means, in contrast to latch coupling, more than two nodes might be locked at the same time by a thread, which reduces the potential for concurrency.

2.7 The OLFIT algorithm

The OLFIT (*optimistic, latch-free index traversal*) algorithm has been proposed in [Cha+01]. It is a variant of the B^{link}-tree that does not use latches when reading the contents of a node, either during search or when traversing down the tree during an insert or delete. Instead, each node has a version number that is incremented each time the content of the node changes. Latches are only held during the modification of a node. Read access to a node works as follows:

1. The version number of the node is stored locally.

2. The accessing thread spins until the latch is free and the version number of the node matches the one stored locally, updating the latter whenever there is a mismatch.

3. The access (search in the key array, and possibly reading the associated RID in a leaf) is performed.

4. The same check of latch and version number as above is performed. If the access was successful, the thread proceeds to the next node. Otherwise, it repeats the access from step 2 onward.

Chapter 3

A Glimpse at Transactional Memory

Transactional Memory has first been proposed as a hardware-supported synchronization mechanism to allow using lock-free data structures [HM93]. It should make the following two guarantees, analogous to constraints on transactions in databases:

Atomicity Meaning either all changes of the transaction become effective at once, when the transaction is committed successfully, or none of these do.

Serializability Meaning every set of transactions appears to outside observers to be executed serially, and it appears to be executed in the same order for every observer.

In this chapter, we will look at several proposed hardware and software implementations of transactional memory, in order to better understand its semantics and different ways to implement it and possibly resulting challenges. The proposals are:

- The original proposal for *Hardware Transactional Memory* (HTM) by Herlihy and Moss [HM93], utilizing the CPU's caches and cache coherence protocol.
- The original proposal for *Software Transactional Memory* (STM) by Shavit and Touitou [ST95].
- A proposal for implementing STM in Concurrent Haskell by Harris et al. [Har+05].

3.1 Hardware Transactional Memory

In his original proposal, Herlihy defined a set of instructions that could be used to leverage transactional memory in software:

LT (Load transactional) Load a value as part of a transaction.

LTX (Load transactional exclusive) Load a value as part of a transaction with the intention of modifying it later.

ST (Store transactional) Store a value modified during a transaction.

COMMIT Commit the modifications of a transaction to shared memory.

ABORT Discard the modifications of a transaction.

VALIDATE Check if the current transaction has been aborted (an *orphan transaction*) or is still valid.

A transaction can be successfully committed if no other write in the system conflicts with any read or write in the transaction and no other read in the system conflicts with any write in the transaction. When a transaction is committed, the modifications of the write set become visible to the system, the write set being any value written with a ST instruction as described above.

The validate instruction checks for such conflicts and discards the modifications (just as with an explicit abort instruction as described above) if necessary. This is motivated by the fact that transactions are not automatically aborted as soon as a conflict is detected. Therefore, the validate instruction can be used to avoid errors in orphan transactions. Other instructions do not affect the read or write set, so values can be loaded or modified during (in parallel with) the transaction.

One proposed scheme for simple transactions is to

- load the necessary data,
- validate that no conflicts have occurred until then,
- store the modifications,
- commit the transaction.

The transaction duration in this proposal should be shorter than a *scheduling quantum*, i.e. the time between two context switches.

The proposed implementation uses a small, dedicated, fully-associative cache to store the write set of a transaction. A commit causes the write set to become visible by forwarding all requests to locations in the write set to this cache instead of the existing, non-transactional cache. The cache coherence protocol is used to implement conflict detection. This is done in an *attacker-loses* fashion, meaning a transaction that tries to write to a location that is part of another transaction's data set is aborted.

3.2 Software Transactional Memory

Transactional memory has been implemented in a number of programming languages. There are several experimental compiler- and library-based implementations for C and C++ and also many official or unofficial implementations in languages that rely on virtual machines, for example CLR-based languages like C# and F# or JVM-based languages like Java and Scala.

A simple method of implementing software transactional memory is to make a shared data structure accessible only via a single pointer, then copying that data structure to some private memory location, making changes to it, and finally swapping it with the original data structure using the pointer [Her93]. The implementation requires hardware support to make this swap atomic. However, such support is needed anyway to implement locks and also most lock-free data structures. On x86 CPUs, this is possible with an atomic CMPXCHG[1] instruction. This approach, while allowing concurrent read access to a shared data structure, has at least two major drawbacks:

- Copying a large data structure may be too expensive (especially if only small modifications are made).
- Concurrent write accesses are not possible.

Another approach is the implementation of STM in Concurrent Haskell [Har+05]. Most functional programming languages clearly separate functions with side effects, for example modifying shared memory or I/O, from other functions. STM in Haskell marks functions used in transactional code in a similar way. In transactions, functions doing I/O are never allowed, because input/output operations cannot be reverted. Purely functional code in these regions, on the other hand, need not be supervised or logged in any way, because it is known that it never has an effect on shared memory, so there are no results that would need to be undone.

In this implementation, all variables that must be accessed transactionally in a safe way have a dedicated type. Larger transactions are composed of smaller transactions and the smallest transactions are simple read and write operations of these variables.

Beyond the Haskell API for transactions, some parts of transactional memory are implemented in C as part of the Haskell runtime. There, each of the transactional variables has a log on the heap associated with it. The log stores the value from before

[1] compare and exchange

the start of a transaction and the value to be committed at the end of a transaction. In case of an abort, the logs of the variables involved are simply discarded. In case of a commit, the runtime checks if all variables in the transaction refer to the same original variables, i.e., no other thread has modified the variables since the beginning of the transaction, and then applies the changes. This is safe because the threads involved are *green threads*, which are managed by the runtime. They are only switched out at safe points and only one thread ever executes these C function at one time. This, of course, limits the amount of concurrency achievable on a multi-core or multi-CPU system.

After looking at some generic hardware and software implementations of transactional memory, in the next chapter we will focus on Intel's Transactional Synchronization Extensions and see how they correspond to the concepts presented here.

Chapter 4

Intel Transactional Synchronization Extensions

In this chapter we will give an overview of Intel's implementation of hardware transactional memory, the *Transactional Synchronization Extensions* (TSX)[INTC12c]. We start with recalling some basics about the x86 architecture. Then we will describe the two main components of the Transactional Synchronization Extensions, *Hardware Lock Elision* (Section 4.2) and *Restricted Transactional Memory* (Section 4.3). We will conclude this chapter with some microbenchmarks to better assess the capabilities of these instruction set extensions.

According to Intel, the goal of their implementation is to keep a programming model based on coarse-grained locks, but provide performance that is closer to using highly scalable operations as is possible with fine-grained locks [INTC12a]. Intel envisions TSX as an alternative to algorithmically more complicated concepts like fine-grained locking schemes, lock-free data structures and optimistic locking.

The Transactional Synchronization Extensions were first made available by Intel as part of the Haswell line of processors [INTC12b] and have been present in every CPU generation since. However, in Haswell and some Broadwell processors the extensions have been deactivated with a microcode update because of stability issues [INTC16, p. 52]. Support for these extensions has been enabled in major projects like the GNU project's GCC since version 4.8 [GNU13] and glibc since version 2.18 [Mil13].

4.1 x86 Architecture Basics

To understand the way the new Intel instructions work, one has to understand two main aspects of the x86 architecture: instructions and instruction prefixes on the one hand and the way caches work on the other hand.

4.1.1 Prefixes and Instructions

Instructions can be preceded by up to four prefixes that change the way an instruction is executed. The prefixes are split into four groups, and up to one prefix out of each group may be used at the same time [INTC15b]:

Group 1 includes the LOCK prefix (see subsection 4.2.1) and prefixes for repeating operations on strings.

Group 2 includes the management of segment overrides and on some architectures branch hints.

Group 3 includes operand-size overrides to switch between 16- and 32-bits from their default size.

Group 4 includes address-size overrides. Like operand-size overrides, this enables using 16- or 32-bits width.

4.1.2 Caches

Modern CPUs use multiple caches to speed up memory accesses. Intel uses a hierarchy of at least three levels of caches:

1. Separate L1 caches for instructions and data, respectively. These are typically 32 KiB in size each.

2. An L2 cache of typically 256 KiB per core.

3. a larger L3 cache that is shared between all cores and ranges from 2 MiB[1] to 8 MiB in size on consumer CPUs.

Since multicore CPUs have individual L1 and L2 caches per core, features like a cache coherency protocol are necessary even in small consumer CPU models. Intel uses the *MESI* protocol for coherency. The MESI protocol is so named because it assigns

[1] 1 MiB = 1024 KiB

to each cache line one of several states: Modified, Exclusive, Shared, and Invalid, with the following meanings:

Modified The cache line has been modified on the core and the changes are not yet reflected in the L3 cache.

Exclusive The cache line is loaded exclusively in this core. If another core loads it, it becomes shared.

Shared The cache line is loaded in multiple cores.

Invalid The cache line, which was previously shared, has been modified on another core and thus the data is obsolete.

Although not declared publicly by Intel, it is widely speculated that the cache coherency protocol is used to keep track of transactions. A cache line that is only read from in a transaction would be marked exclusive or shared. A cache line that is written to would be marked modified. And when a cache line is invalidated, the transaction would be aborted. Furthermore, this would mean that conflicts are handled in a so-called *attacker-wins* fashion, were in particular readers can abort writing transaction by causing eviction of one of their cache lines [Kan12].

4.2 Hardware Lock Elision

Hardware Lock Elision (HLE) is a backwards-compatible pair of instruction prefixes. They are intended to be used with the existing LOCK instruction prefix. The backwards-compatibility stems from the fact that they are silently ignored by CPUs that do not support TSX. This is possible because while these prefixes are represented in the same way as two existing prefixes, these existing prefixes have no meaning when used together with the LOCK prefix.

4.2.1 The LOCK Instruction Prefix

LOCK is a prefix that can be used before certain instructions to implement atomic operations. It prevents other cores from accessing the same memory as the following instruction. This allows for example to implement an atomic test-and-set function, where the LOCK prefix guarantees that the Bit-Test-and-Set (BTS) instruction is executed atomically. Figure 4.1 shows schematically how this can be done:

1. LOCK is used to execute the following instruction (BTS) atomically.

2. BTS (Bit-Test-and-Set) sets the specified bit (in this case bit 0) in *lock* and stores the previous value of the bit in the *Carry Flag* (CF) register.

3. ADC (Add with Carry) stores the sum of 0, *result* and CF in *result*.

```
bool *lock = /* initialize lock */
bool result = false;
asm volatile("lock; bts $0, %[lock]"
             : [lock] "=m" (*lock) : : "memory");
asm volatile("adc $0, %[result]"
             : [result] "=m" (result) : :);
```

Figure 4.1: Trying to acquire a lock

4.2.2 The New HLE Prefixes

HLE provides two new instruction prefixes, XACQUIRE and XRELEASE:

XACQUIRE must precede the LOCK prefix.

XRELEASE is used before an instruction and a memory address that was previously accessed with XACQUIRE.

With these two backwards-compatible prefixes, locks can be elided in the following way:

- If a conflict occurs, the CPU aborts the transaction and executes the same code again, but without the HLE prefixes.
- If no conflict occurs, the code is executed as if there were no locks in use at all.
- The lock-variable is not actually written to. However, to ensure backwards-compatibility, during the transaction the lock-variable appears to have been set.

Backwards-compatibility in this context means that the HLE prefixes can be used in a library for locking, even when the library is used by a caller that is not aware of HLE and tries to examine the state of the lock after acquiring it.

The prefixes can be used in assembler code directly. For example, the lock in Figure 4.1 can easily be elided by adding the prefix in the following way:

```
asm volatile("xacquire; lock; bts $0, %[lock]"
             : [lock] "=m" (*lock) : : "memory");
```

The new prefixes can also be used in conjunction with GCC's built-in atomic functions. These are similar to the atomic functions introduced with C11[2], however they were introduced earlier and are GCC-specific. These functions, just like the ones in C11, allow the user to specify a *memory ordering*. For implementing spinlocks, the memory orderings *acquire* and *release* are usually used, which demand that every instruction before the lock-acquiring instruction and after the lock-releasing instruction be executed before resp. after, without the compiler nor the hardware reordering instructions. These memory orderings are specified using flags in the code (named `__ATOMIC_ACQUIRE` and `__ATOMIC_RELEASE`). To enable elision of the lock with HLE, GCC provides the additional flags

- `__ATOMIC_HLE_ACQUIRE` and
- `__ATOMIC_HLE_RELEASE`.

For example, to implement a spinlock, the `__atomic_exchange_n`-instruction can be used in the following way:

```
while (__atomic_exchange_n(&lock, true,
    __ATOMIC_ACQUIRE | __ATOMIC_HLE_ACQUIRE));
```

Assuming the variable lock is a boolean variable, the spinlock is elided through the use of the flag. Similarly, when unlocking an elided lock, a flag has to be used to tell the compiler to use the corresponding XRELEASE instruction:

```
__atomic_store_n(&lock, false,
    __ATOMIC_RELEASE | __ATOMIC_HLE_RELEASE);
```

[2]ISO/IEC 9899:2011

4.3 Restricted Transactional Memory

Restricted Transactional Memory (RTM) is a set of new instructions to allow developers explicit control over the beginning and end of transactions, whereas with HLE transactions can only begin and end with instructions that have the LOCK prefix. This allows for much more flexible usage of transactional memory and with no loss of performance compared to HLE, because the same underlying hardware mechanisms are used. RTM consists of three instructions XBEGIN, XEND, and XABORT, that are described in the following. However, because new instructions are introduced, binaries containing these instructions are not backwards-compatible to older processors.

4.3.1 Instructions

XBEGIN The XBEGIN instruction attempts to start a transaction. It takes as an argument the address of a region of code to execute as a fallback path (in case of an abort).

XEND The XEND instruction tries to commit a transaction.

XABORT The XABORT instruction aborts the current transaction. It takes as an argument an 8-bit code as the reason for the abort.

When a transaction is aborted, an error code is stored in register EAX. This means that the fallback path can examine the error code and decide whether to retry the transaction or follow some other path. The error code is a 32-bit integer encoding information about the cause of the abort (Table 4.1).

Bit	Meaning
0	Abort caused by XABORT command
1	CPU believes retry might be successful
2	Conflicting memory access
3	Transaction too large
4	Debug or breakpoint exception occured
5	Abort in nested transaction
23:6	Reserved
31:24	Argument passed to XABORT

Table 4.1: Contents of EAX after an abort

4.3.2 Intrinsics

The header file <immintrin.h> provided by the Intel compiler and by GCC contains the following intrinsics to help developers:

- `int _xbegin();` attempts to start a transaction. The return value is one of:
  - `_XBEGIN_STARTED` Transaction started successfully
  - `_XABORT_EXPLICIT` Transaction aborted by `_xabort()`
  - `_XABORT_RETRY` Retry of transaction may be successful
  - `_XABORT_CONFLICT` Transaction aborted due to data conflict
  - `_XABORT_CAPACITY` Maximum transaction size exceeded
  - `_XABORT_DEBUG` Transaction aborted due to debug breakpoint
  - `_XABORT_NESTED` Transaction nested too deeply
- `void _xend();` attempts to commit a transaction.
- `void _xabort(unsigned int);` aborts a transaction, storing the specified abort cause in EAX.

To give a better understanding of the assembler instructions, consider how the intrinsic `_xbegin()` is implemented:

1. The XBEGIN instruction is executed, with the abort handler being the region of code immediately after XBEGIN.
2. A return value indicating success is stored in EAX.
3. In both the transaction and the fallback path, the same region of code is executed next, namely using the contents of EAX as the return value of the intrinsic.

This makes the use of intrinsics rather straightforward: If a transaction aborts for whatever reason, the control flow appears to the programmer as if the intrinsic had returned an error code immediately and the transaction had never even started.

The header also provides the macro definition `_XABORT_CODE(x)` that retrieves the bits of the return value reserved for the custom abort code. Thus, a check for this custom abort code would look as follows:

1. Aborting the transaction: `_xabort(42);`

2. Checking for the custom code:

```
int result = _xbegin();
if (result & _XABORT_EXPLICIT) {
    if (_XABORT_CODE(result) == 42) {
        ...
    }
    ...
}
```

The main advantage of RTM over HLE is the ability to define the fallback code path. This is mainly useful for retrying failed transactions or abandoning the execution of a transaction altogether.

4.4 Other Features and Comparison

In both HLE and RTM another instruction, XTEST, is available. It allows checking whether the CPU is currently executing a transaction. The main difference between HLE transactions and RTM transactions is the handling of lock variables:

- Inside an HLE transaction, the lock will appear to be set, but is not, in fact. This makes locking libraries backwards-compatible with old code using the library, because if the old code reads the state of a lock during the transaction (instead of, for example, using the XTEST instruction), the lock will appear to be in the expected state.
- Inside a RTM transaction that only reads the lock, the lock won't appear to be set and is not, in fact.

From a user's point of view there are two important differences between transactional memory as implemented by Intel and Herlihy's proposal as discussed in the previous chapter:

- There are no explicit load, store, and validate instructions. Every variable read (written) during a transaction becomes part of the read set (write set).
- There are no orphan transactions, since they are aborted as soon as a conflict is detected.

4.5 Limitations

Each CPU core tracks memory accesses at the granularity of a cache line (64 B). The CPU uses the cache coherency protocol to synchronize transactions between cores. This brings with it a number of limitations:

- There are some instructions that must not appear in transactional regions, namely:
  - CPUID
  - PAUSE
  - XABORT
- There are other instruction that might cause a transaction to abort, depending on the specific hardware implementation, for example:
  - System calls
  - Instructions interacting with special registers
  - Instructions controlling the TLB and caches
  - Interrupts
  - I/O instructions
- Transactions can use too much memory, making the CPU unable to track the read or write sets.
- Conflicting accesses to the same cache line can occur, even if the accessed memory areas do not overlap.
- Accessing different cache lines that map to the same cache set leads to aborts, because the caches are set-associative.

The following are pitfalls to be considered:

- Intel does not guarantee that a transaction will ever succeed. This requires a non-transactional fallback code path to be designed, even if one were willing to try the transactional path until the transaction has succeeded. Not providing a fallback path could lead to infinite loops.
- Any exception or error in the processor that occurs in a transactional region causes the transaction to abort and prevents an exception handler from being executed. For example, de-referencing a null pointer or dividing by zero would

cause an abort, and if the error incidentally does not occur in the fallback path, an error in the program logic might go unnoticed. In combination with the case above, if no fallback path is implemented and an error in the program logic causes every transaction to fail, then an infinite loop would happen.

4.6 Microbenchmarks

Having described the new instructions, we will now present three experiments in hopes of giving a better understanding of the possibilities and limitations of transactional code.

The first experiment tries to measure the maximum amount of data that can be read during a transaction. The second experiment tries to measure the maximum amount of data that can be written during a transaction. The third experiment measures the maximum amount of nesting within transactions.

All these experiments where conducted on an Intel Core i7-6700T CPU, part of the Skylake generation [INTC15a]. It provides four physical cores with 32 KiB L1 cache per core and 256 KiB L2 cache per core.

4.6.1 Maximum Read-Set Size

The maximum read-set size is assumed to be bounded by the size of the L2 cache [Kan12]. In current generation Intel CPUs, this amounts to 256 KiB. In the experiment, increasingly larger amounts of memory are allocated, aligned to the cache-line size. Then, during a transaction, all the data in the memory is read, after which the transaction commits. The experiment is repeated until the user aborts the program. The code in Figure 4.2 is the core of the benchmark, attempting to commit a transaction that reads *size* bytes. The bytes are expected to be initialized with the value 1. The maximum size of a read transaction achieved in practice was slightly above the L2 cache size, so the hardware seems to be using more than just the L2 cache to track read sets. Figure 4.3 shows the abort rates for different read-set sizes.

4.6.2 Maximum Write-Set Size

The maximum write-set size is assumed to be bounded by the size of the L1 cache. In current generation Intel CPUs, this amounts to 32 KiB. In the next experiment, we again allocated increasingly larger amounts of memory, aligned to the cache-line size. During a transaction, every byte is then set to 1 to maximize the write-set, as can be

```
static
bool run_test(int8_t* data, size_t size)
{
    if (_xbegin() == _XBEGIN_STARTED) {
    for (size_t i = 0; i < size; ++i) {
        if (data[i] != 1)
            _xabort(0);
        };
        _xend();
        return true;
    } else {
        return false;
    }
}
```

Figure 4.2: Measuring maximum read set size

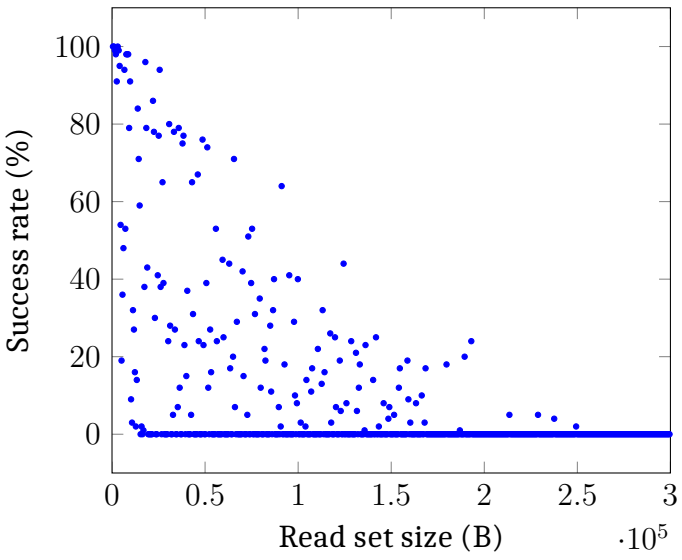


Figure 4.3: Success rate when testing the maximum read set size

```
static
bool test_max_write_set_size(int8_t* data, size_t size)
{
    if (_xbegin() == _XBEGIN_STARTED) {
        for (size_t i = 0; i < size; ++i)
            data[i] = 1;
        _xend();
        return true;
    } else {
        return false;
    }
}
```

Figure 4.4: Measuring maximum write set size

seen in Figure 4.4. Figure 4.5 again shows the success rates for a particular run of the benchmark. The maximum size of a transaction achieved in practice was around 25 KiB.

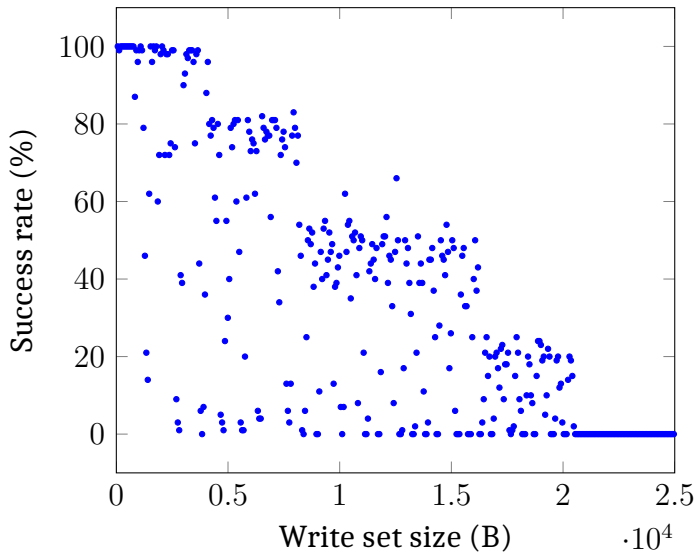


Figure 4.5: Success rate when testing the maximum write set size

4.6.3 Maximum Nesting Level

As transactions can be nested, we tried to check the maximum number of times a transaction can successfully be nested. The function in Figure 4.6 differentiates between transactions aborted because of too deep nesting and transactions aborted for other reasons. Whenever a transaction with a particular nesting level succeeds, a higher level can be tried. If a transaction aborts with error code _XABORT_NESTED, the maximum nesting level has surely been found. In contrast to the previous experiments, this allows to draw stronger conclusions about the limitations of the hardware. The maximum nesting level in practice was found to be 7.

```
int test_max_nesting_level(unsigned int level)
{
    unsigned int result;
    for (unsigned int i = 0; i < level; ++i) {
        if ((result = _xbegin()) == _XBEGIN_STARTED);
        else if (result == _XABORT_NESTED)
            return NESTING_FAILURE;
        else
            return NESTING_UNKOWN;
        }
    for (unsigned int i = 0; i < level; ++i)
        _xend();
    return NESTING_SUCCESS;
}
```

Figure 4.6: Measuring maximum nesting level

4.6.4 Evaluation

Similar microbenchmarks have been conducted by [Goe+14]. We can confirm their findings regarding the write-set sizes. Surprisingly, at least since the Skylake generation, the read-set size seems to be no longer bounded by the L2 cache. Additionally, they measured the maximum duration a transaction can last and found that limit to be about ten million CPU cycles. Finally, they measured the overhead and poten-

tial benefits of TSX compared to a traditional spinlock. They benchmarked multi-threaded removals from a queue, synchronized with a spinlock, a compare-and-swap instruction and an RTM transaction (with unlimited retries). The results were a theoretical overhead of 45 % of RTM compared to the spinlock, but already at low contention RTM provided a net benefit in execution time of over 30 % and over 50 % at high contention, beating even the compare-and-swap instruction.

In the next chapter, we will discuss how we used the features TSX provides in a custom B-tree implementation.

Chapter 5

The Bx-Tree

The main goal of the thesis is to investigate in what ways TSX, and Restricted Transactional Memory in particular, can be used to increase B-tree performance in multi-threaded scenarios. In this chapter, we will present the Bx-tree, an enhancement of the B^{link}-tree, in which we have incorporated RTM. We will begin with a general description of the tree and its node layout, which is in its core an implementation of the B^{link}-tree. Then we discuss how the basic search, insert, and delete functions are implemented. Finally, we describe the related findings in [MLS15] that focused on B-trees and the Bw-tree.

5.1 General Description

The Bx-tree is derived from the B^{link}-tree and implemented in C. Because with Intel TSX a non-transactional fallback path is always needed, the latching scheme of the B^{link}-tree serves as that fallback. This means the tree is protected by a latch in case a new root has to be inserted, and every node is protected by a latch. All latches are implemented as simple spinlocks with atomic operations.

The height of the tree is stored inside and incremented on a root split, instead of calculating it dynamically when needed (e.g. for printing statistics).

After performing the benchmarks to find the optimal node size (see Section 6.2), the size has been set to 1024 B, which allows for storing 62 keys and values in leafs, and 61 keys with the accompanying 62 pointers in inner nodes.

Every transaction is retried up to a maximum retry count specified at compile time, a sensible value for which was found to be 3 (see Section 6.6).

Because B^{link}-trees perform splits not in a single step, but instead insert new nodes first through the right-sibling pointer, the parent nodes have to be remembered when

traversing down the tree. This could be handled by inserting the child pointers immediately afterwards, as is done in B-trees. In this case, assuming a recursive algorithm is used for traversal, the information about parents is part of the call stack. To allow for greater flexibility as to when to perform the insert, we chose to implement a custom stack, allocated on the heap, just for storing the nodes visited during a traversal. This stack is used only when necessary, i.e. when traversing the tree to find a leaf during inserts. After first allocating and later freeing a new stack for each operation, we identified this behavior as having a noticeable impact on overall performance. Therefore, we now store the stack along with the tree using a thread-local key (part of the Pthreads API), so one stack per thread is used for the entire duration of a benchmark.

Another memory allocation issue is the creation of new nodes when splittings leafs or inner nodes. Because library functions like `malloc`, causing system calls, cannot be used inside transactions, a possible future refinement could be to implement a simple memory management solution for nodes in userspace.

Search inside nodes is performed using the binary search implementation described in Section 2.2.6. The reasoning behind this choice is that binary search scales better with node sizes greater than 1024 B, and because we compiled the benchmarks with the highest optimization level.

5.2 Node Layout

Table 5.1 shows the layout of a 1024 B node of the Bx-tree. The type describes whether the node is an inner node or a leaf. The high key is the smallest value accessible from the right sibling, or `INT64_MAX` if there is no right sibling. The keys are stored in an array whose length is determined at compile time based on the size of the node. The pointers to children are stored in a "simulated" array: the pointers are stored contiguously inside the node, but instead of an array variable a pointer to the second of those pointers is stored. This allows to access the second pointer, which is covered by the first key, with the array notation `children[0]` and the first pointer, which is not covered by a key, as `children[-1]`. The major benefit of this approach is that it corresponds well to a (binary or other) search implementation that returns -1 in this case (see Section 2.2.5).

Component	Size	Type
Type	1 B	`bool`
Latch	1 B	`atomic_bool`
Size	2 B	`uint16_t`
High key	8 B	`int64_t`
Right sibling	8 B	`Node *`
Keys	488 B	`int64_t[61]`
Children	8 B	`Node **`
Space for child pointers	496 B	
Total size	1024 B	*12 B unused*

Table 5.1: Layout of inner nodes

5.3 Algorithms

In this section we will give some details about the way standard B-tree operations are implemented with TSX.

Search is performed in two transactions: first, the search descends from the root down to the first matching leaf and the transaction is committed (see Subsection 5.3.1). This way, the transaction is not aborted if the leaf is just about to be modified by another thread. In the second transaction, the search moves to the right, making up for any splits that might have occurred between the two transactions. In this second transaction, the value (RID) belonging to the key is then also retrieved.

Insertion uses the same function as search to transactionally navigate down the tree, only at the same time adding the visited nodes to the stack. In a second transaction we then move to the right if necessary and insert the value, splitting the leaf if necessary (see Subsection 5.3.1). The insert of the new right sibling into the parent, and every subsequent insert into parent nodes or split of parent nodes, is performed in its own transaction (including the move to the right).

Deletions are also performed in two transactions: the leaf search and then the move to the right and removal of the entry.

The insertion, split and removal functions use the C standard library functions of `memcpy` and `memmove`. We have identified these to be crucial to the performance:

when trying to replace the implementation provided by glibc with an own implementation as well as with another C standard library, the performance was approximately halved.

Having described the general idea of the Bx-tree operations, we will now exemplarily focus on how the traversal and insert operations are implemented.

5.3.1 Navigating Down the Tree

Navigating, or traversing, down the tree, is arguably the most important operation of a search tree. In contrast to B-trees, where the different latching schemes for search and insert require custom implementations of the traversal algorithm, in B^{link}-trees the implementation can be shared because only one latch is ever acquired at any time.

In this implementation, the result of the traversal is *a* leaf, which might be either the target leaf or a left to the left thereof, in case the target leaf has been split after it was identified in the parent. The procedure is depicted in Algorithm 1. It receives as arguments a starting node (usually the root), the key to look for, and a stack which is optionally used in inserts. The algorithm contains one transactional region, which is repeated up to MAX RETRY COUNT times. Aborts are handled in the block beginning at line 21.

A retry is usually advised after conflicts. The retry counter is then incremented and a low-level pause command is issued, so as to reduce traffic on the memory bus and not provoke the event that caused the abort to immediately occur again. In cases such as size aborts, a node being explicitly locked, or after other spurious aborts, a retry is not advised and the function returns *null* immediately. This signals to the caller that the fallback path using latches should be taken.

At certain points during the traversal (lines 7 and 10, after a new node is encountered), the state of latches is checked. This serves to make the latch part of the read set of the transaction. If the latch is locked, the transaction is explicitly aborted (exemplified through the use of the word *assert*). If the latch is not locked at the beginning of an access, but becomes locked after that, the conflict is detected by the hardware, which then induces an abort.

Otherwise, the transaction traverses down the tree until a leaf is encountered. This leaf is then the result.

Algorithm 1 Navigating to a leaf close to the target

```
1:  function FindLeaf(node, key, stack)
2:      retries ← 0
3:      current ← node
4:      while retries < MAX RETRY COUNT do
5:          if transaction successfully started then
6:              while current is not leaf do
7:                  assert current's latch is not locked
8:                  while key ≥ node's high key ∧ current's right sibling is defined do
9:                      current ← current's right sibling
10:                     assert current's latch is not locked
11:                 end while
12:                 if stack is defined then
13:                     push(stack, current)
14:                 end if
15:                 index ← lower bound(current's keys, key)
16:                 current ← currents's children[index]
17:             end while
18:             commit
19:             return current
20:         else
21:             if retry advised then
22:                 retries ← retries + 1
23:                 pause
24:             else
25:                 return null
26:             end if
27:         end if
28:     end while
29:     return null
30: end function
```

5.3.2 Inserting an Entry

Part of the insertion procedure is depicted in Algorithm 2. It receives as parameters a leaf, the key and RID to be inserted, and a variable to store the result of the actual insert into. This result is *null* after a regular insert and a tuple (key, pointer to node) in case a split has happened. The return value of the function is either *true* or *false*, indicating whether the insert was successfully performed in a transaction, or has to be retried with latches.

Most importantly, the algorithm starts by moving to the right, which is necessary if the leaf returned by the previous algorithm has been split in the meantime.

Algorithm 2 Inserting an entry into a leaf, potentially moving to the right first

```
1:  function InsertIntoLeaf(node, key, rid, result)
2:      retries ← 0
3:      current ← node
4:      while retries < MAX RETRY COUNT do
5:          if transaction successfully started then
6:              assert current's latch is not locked
7:              while key ≥ currents's high key ∧ current's right sibling is defined do
8:                  current ← current's right sibling
9:                  assert current's latch is not locked
10:             end while
11:             result ← AddOrSplit(current, key, rid)
12:                                  ▷ tuple (new key, new right sibling) in case of split
13:             commit
14:             return true
15:         else
16:             if retry advised then
17:                 retries ← retries + 1
18:                 pause
19:             else
20:                 return false
21:             end if
22:         end if
23:     end while
24:     return false
25: end function
```

5.4 Related Work: the Bw-Tree and TSX

The most in-depth related analysis of B-tree designs and TSX was done by Makreshanski, Levandovski, and Stutsman [MLS15]. They mainly applied HLE and RTM in two different trees:

- A B-tree with a global latch, serializing all accesses.
- The Bw-tree, developed at Microsoft, where RTM is used to implement a multi-word compare-and-swap function.

For the B-tree, the result with HLE was almost optimal scalability in certain read-only workload (the optimum being no synchronization at all), but a significant drop in performance with larger trees, mainly due to cache associativity conflicts and transaction size. In the worst case, performance with the global elided latch dropped below the level of performance when the latch is not elided, i.e. when all operations are performed in sequence. Due to these circumstances, they concluded that TSX is not adequate for augmenting B-trees with latch coupling.

When comparing the B-tree with a global latch (elided) and the non-transactional Bw-tree, it was found that read access performance in the Bw-tree is largely unaffected by even high numbers of concurrent writes, because the copy-on-write mechanic prevents conflicts between readers and writers in general. In the B-tree with HLE, aborts caused by writes lead to a drop in read performance of as high as 80 % with high numbers of concurrent writes.

Lastly, Makreshanski et al. implemented a multi-word compare-and-swap instruction for their Bw-tree using RTM. The main advantage of this approach was a reduction in code complexity while maintaining acceptable performance. The reduced complexity stems from the fact that the non-transactional implementation of accessing the page mapping table was replaced by access through a global elided latch (with RTM providing a variable number of retries). The performance, albeit slightly reduced, remained within reach because the transactions were small (constant size), so the global lock was rarely taken.

Chapter 6

Benchmarking the Trees

In this chapter we will describe how we have performed benchmarks and what insights they revealed. At he beginning we will briefly describe the implementation of the benchmarking framework and the environment the benchmarks were run in. Then we will compare different B-tree variants using the framework. This entails three different areas of focus: first, the fundamental differences between the "traditional" B-tree variants, i.e. the B-tree, the B^{link}-tree, and the CSB-tree, will be analyzed. This also includes finding the optimal node sizes for these trees. Second, the trees, which rely on latching as a synchronization mechanism, are used in conjunction with Hardware Lock Elision. Third, the Bx-tree is included in the analysis, with an emphasis on tuning different variables for maximum performance.

For reference, we have also implemented the following variants of B-trees:

- The regular B-tree using latch coupling for synchronization
- The B^{link}-tree
- The cache-sensitive B-tree
- The B^{link}-tree using the OLFIT algorithm for synchronization

In these implementations, keys must be unique. Inserting an entry with a duplicate key leads to an error. This was a choice made purely for benchmarking purposes, because skipping the insert would mean that the operation does not bear the true cost of an insert and allowing a key to be associated with multiple values would have complicated the node layout to much.

6.1 About the Benchmark

In this section we describe how the benchmark is implemented. The benchmark consists of two parts: data generation, done in Python ahead of time; and storing and accessing the data in B-trees, done in C.

6.1.1 Data Generation

The data is generated by a Python script and then written to a text file. "Data", in this case, is a sequence of operations to be performed later. An operation has a type, which is either *Search*, *Insert*, or *Delete*. In the case of inserts, the operation specifies the 64 bit key and the 64 bit (unsigned) value to be inserted into the tree. Search and delete also specify the key and the previously stored value, in order to allow checking the correctness of the tree implementation. The script creates at least one file, which consists solely of insert operations, and is used to fill the tree with its initial values. Then, arbitrarily many other files can be created that contain all three operations. An *update rate* can be given to allow simulating read-heavy or write-heavy workloads. Each file created this way contains the same number of operations as the initial file, so the number of files determines how long the benchmark will run. "Updates" are split evenly into inserts and deletes, so the size of trees after initial creation stays roughly the same. In each sequence of operations (except the first, of course) only keys inserted in one of the previous sequences are searched or deleted, and only keys not previously inserted can appear in insert operations. Thus, splitting the workload into multiple files creates barriers that allow arbitrarily long benchmarking without ever inserting duplicates or searching or deleting non-existent keys. This, in turn, guarantees that these operations are never cheaper than expected, by e.g. not actually moving keys on insert or delete.

6.1.2 The Benchmarking Framework

After the data files have been generated as above, they are used for benchmarking the tree. Each file is parsed and loaded into memory first. The operations are stored sequentially and are then handed over to a variable number of threads to be executed. For each file, a new set of threads is created using the POSIX Threads API (*Pthreads*). The threads each consume one operation after another from the sequence. For synchronization, an atomic counter is used to indicate the next operation to be executed. The threads exit when the whole sequence is consumed.

CPU	Core i7-6700T
Base clock	2.8 GHz
L1 data cache	4×32 KiB
L2 cache	4×256 KiB
L3 cache	8 MiB
RAM	8 GiB DDR4-2133

Table 6.1: Hardware used in the benchmarks

6.1.3 Evaluation

All benchmarks were performed on the hardware described in Table 6.1. All time measurements were done using the API in `<time.h>` defined by POSIX, with the `CLOCK_REALTIME` clock, which provides nanosecond resolution. Time is measured from just before the first thread is started until just after the last thread has exited. The total time for the benchmark is the sum of these execution times per file. The initial insertion of data into the tree is reported separately, so read-only workloads can be simulated.

6.2 Optimal Node Size

In this benchmark we have tried to determine the optimal node size for each B-tree type. In disk-based database systems, 4 KiB or 8 KiB are often used as node sizes, partly because they map well to the size of memory pages of the operating system.

However, the optimal node size is highly dependent on the insert ratio, because moving data inside nodes or splitting them is much more expensive with larger nodes. To better ascertain sensible sizes, we ran the benchmark in two different ways, once with only read operations and once measuring the time to fill a tree with the initial data. If the optimal node size for update rates of 0 % (read only) and 100 % is somewhat similar, it should perform equally well also for mixes of operations, as search performance is the best-case scenario. If the optimal node sizes in these cases differ a lot, a choice has to be made: either deciding on a fixed size, possibly after running more benchmarks with different rates; or choosing a size depending on the most likely scenario. In the real world, one would probably go the latter route and use the optimal page size for the expected workload.

Another important variable is the number of threads running in the benchmark. Smaller nodes directly translate to more fine-grained locking, which leads to less con-

tention of latches. However, it also means more random accesses and less potential to benefit from the compiler's search optimization.

6.2.1 Filling the Trees

This is a write-heavy benchmark. We measured the time it takes to fill a tree with 4,000,000 entries. We used only one thread for this, in order not to have the effects of latching mix with other effects at play that we tried to observe. The results can be found in Figure 6.1.

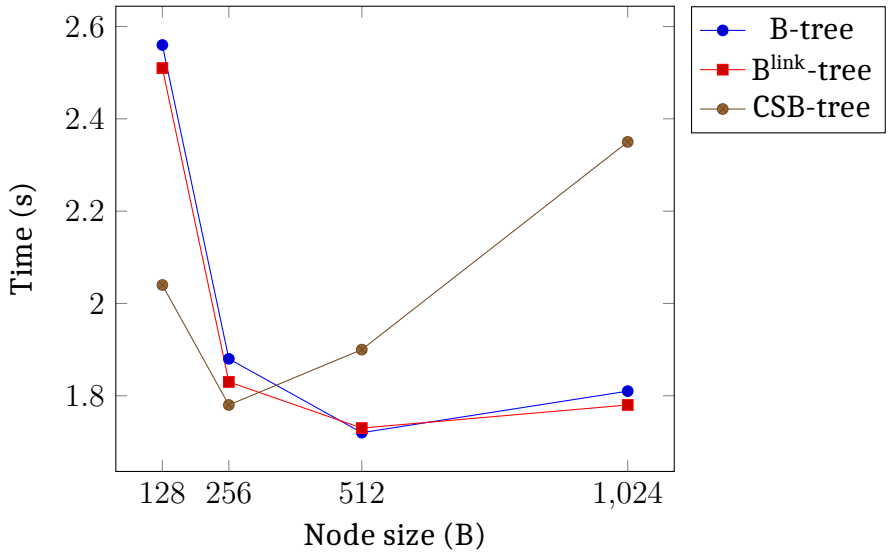


Figure 6.1: Time to fill a tree with 4,000,000 entries

While this it not a particularly realistic scenario, except for the case of creating an index on a large, existing table, it is nonetheless interesting for two reasons:

- The more write-heavy an actual workload is, the closer it comes to this fringe scenario.
- It provides interesting insights into the cost associated with inserts, because no searches or deletes are executed.

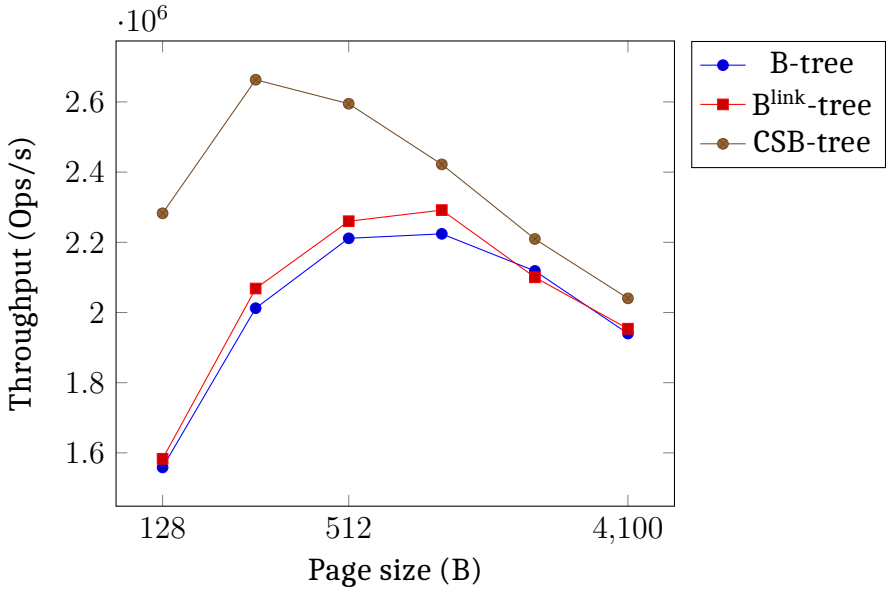
As the results show, for B-trees and B^{link}-trees, the optimum lies at 512 B. We did not try a much broader range of sizes than the one pictured, because the general trend is evident even there. The case of 128 B is a huge deviation from the other sizes. This is probably the result of having to allocate memory for new nodes much more often with smaller node sizes. Also, with a node size of 128 B, the second half of the node

is a prefetched cache line, but contains mostly pointers that are not used during the traversal in inner nodes, something that does not happen with any larger size.

For the CSB-tree, it is clear that the cache-optimized layout provides for very different performance characteristics. With larger node sizes, the curve is much steeper because each insert into an inner node requires moving a lot more child nodes around in memory.

6.2.2 Read-Only Benchmark

Figure 6.2 contains the results for a read-only run. The respective tree is pre-filled with 2,000,000 entries and then 10,000,000 search operations are performed. The keys are uniformly distributed and a single thread searches only for keys contained in the tree.



Entries	Operations	Update rate	Threads
2,000,000	10,000,000	0 %	1

Figure 6.2: Searching in a tree with 4,000,000 values

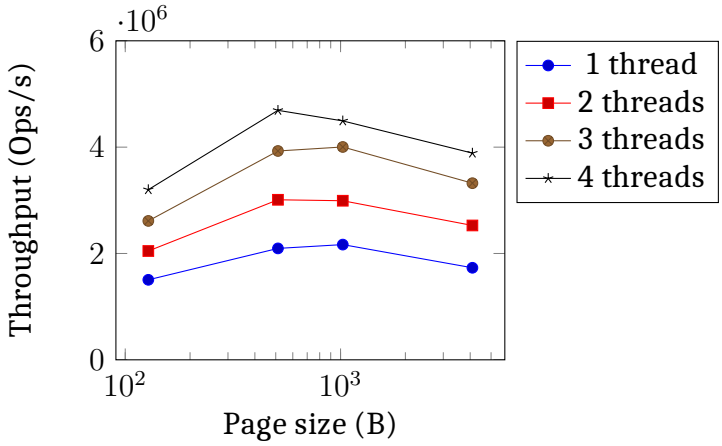
The most important observations are these:

- In this single-threaded case, B-tree and B^{link}-tree behave very similarly.
- CSB-trees provide much higher read performance.
- There is an optimum for all trees with more or less medium-sized nodes. It is not immediately evident why this is the case. We suspect the following two mechanisms to be at play:

- With very small nodes, the search inside the nodes does not have time to "become" efficient. Instead, after a very quick search with all the overhead of initializing the respective variables, a pointer is followed to the next node, which is likely to incur a cache miss.
- With very large nodes, one of the purposes of B-trees, which is to have a large fan-out, is subverted. Instead of reducing the search space dramatically with every node visited, for most of the time it is "only" halved (with a binary search, which is still better than a linear search in this case).

6.2.3 Common Case: Mixed Reads and Writes, Multiple Threads

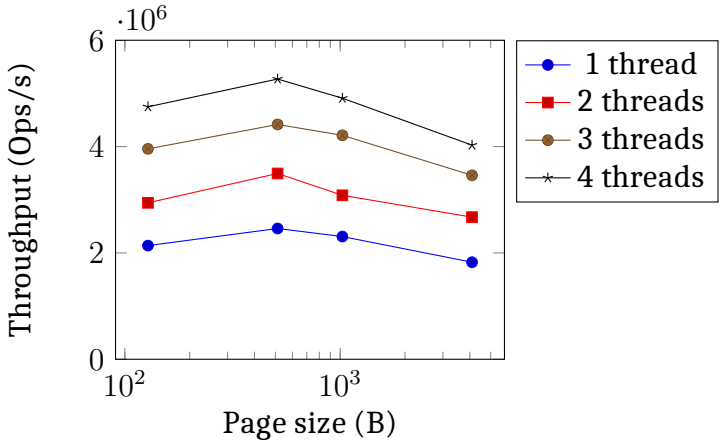
To gain more confidence in the measured optimal values, we decided to run a third benchmark that includes writes in addition to reads. Also, we varied the number of threads now, in order to check whether having more latch contention shifts the optimal value towards smaller or larger nodes. Both are reasonable assumptions: with smaller nodes, the latching is more fine-grained, allowing more parallelism. With larger nodes, the overhead for acquiring and releasing the latch becomes less prevalent compared to the search time inside the node. The results can be found in Figure 6.3 and 6.4.



Entries	Operations	Update rate	Threads
4,000,000	20,000,000	25 %	*variable*

Figure 6.3: 25 % write rate; left: B-tree

Unfortunately, the results revealed little new. The optimal values stayed mostly the same. The only differences seems to be a slight preference for the B-tree towards smaller nodes when moving to four threads.

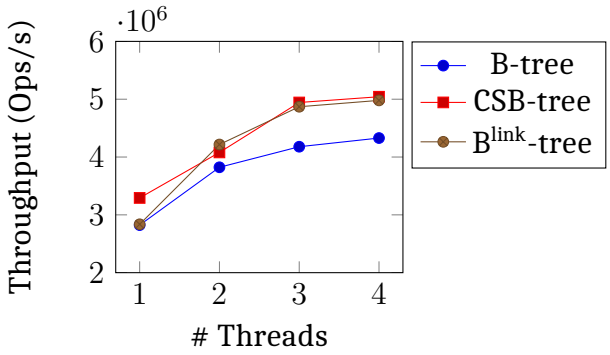


Entries	Operations	Update rate	Threads
4,000,000	20,000,000	25 %	*variable*

Figure 6.4: 25 % write rate; CSB-tree

6.3 Limitations of Non-Transactional B-Trees

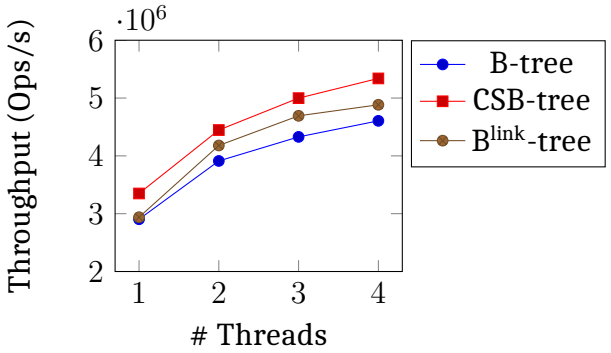
A common scenario in databases is a workload of reads with some intermittent writes. So show the limitations of traditional B-trees, we look at the scalability of these trees when increasing the number of threads. Figure 6.5 and Figure 6.6 show the results of this benchmark.



Entries	Operations	Update rate	Threads
4,000,000	20,000,000	0 %	*variable*

Figure 6.5: Left: read-only workload

In this read-intensive workload, the CSB-tree performs visibly better than the regular B-tree because of the more efficient space utilization inside the nodes. The B^{link}-tree, having almost the same layout, scales better with more threads than the B-tree, while the B-tree and the CSB-tree are nowhere even close to having a linear speedup.



Entries	Operations	Update rate	Threads
4,000,000	20,000,000	10 %	*variable*

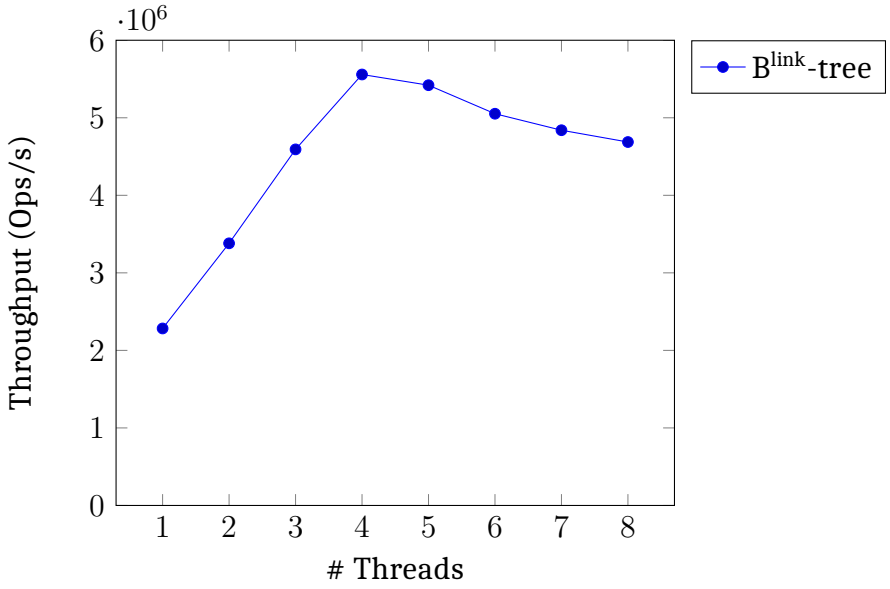
Figure 6.6: right: 10 % write rate

Analyzing the benchmark with the profiling tool *gprof* suggests that a significant cause for these scalability issues is the lock function.

6.4 Effects of Hyper-Threading

In this benchmark we looked at the effects of Hyper-Threading. This is Intel's name for the capability of running two threads on the same physical processor core at the same time, optimizing the usage of functional units. This is especially beneficial when the two threads do different work, because they are less likely to compete for functional units. Scalability is however expected to be less than when adding more physical cores, and the two threads sharing the L1 and L2 caches can have negative effects on performance as well.

For this benchmark, we evaluated the B^{link}-tree only, because it provides the best scalability, so any performance degradation through Hyper-Threading should be better visible than with the other trees, where the tree design itself hampers scalability as well. The results can be found in Figure 6.7. We calculated a speedup of approximately 40 % to 50 % per core compared to the baseline of a single core, and a slowdown of approximately 5 % for each additional thread added after all physical cores (four) have been exceeded. We believe the reason for this is mainly the sharing of caches between the threads, so when two threads running on the same core operate in different parts of the tree, they can use only about half of the shared caches. Also, but probably to a lesser extent, when running as much threads as physically possible at the same time, they will get swapped out more often in favor of other processes unrelated to the benchmark.



Entries	Operations	Update rate	Threads
4,000,000	20,000,000	0 %	*variable*

Figure 6.7: Hyper-Threading in a B^{link}-tree

6.5 Using Hardware Lock Elision

In this benchmark, we studied the effects of using latches with HLE enabled. For this, we look at the three trees that rely heavily on latches for synchronization: the B-tree and CSB-tree with latch coupling and the B^{link}-tree. We look at several different scenarios:

- A read-only workload (Subsection 6.5.1). HLE should provide a benefit in this case because there are no true data conflicts between the threads. The only way conflicts could arise is when transactions are aborted for other reasons like size, forcing to actually acquire the latches and thus starting a cascade where the locking and unlocking of latches causes aborts in the threads who have elided them.

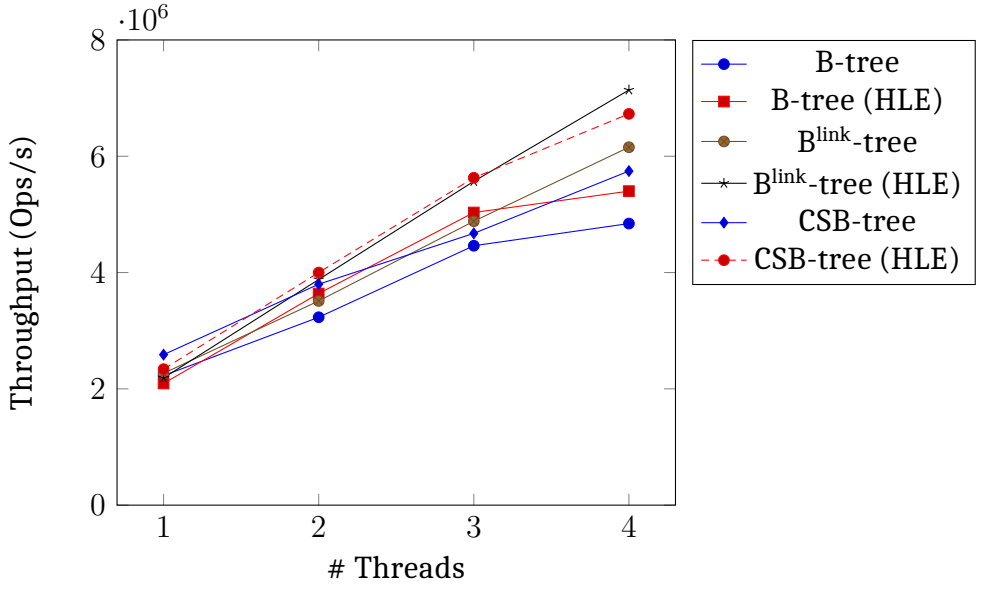
- A write-heavy workload (Subsection 6.5.2). There are a number of effects at play here that might cause HLE to be either a benefit or a detriment. The chance of conflicts is increased, because writes to a node certainly overlap with some other reads from or writes to the same node. On the other hand, write operation take longer, leading to higher lock contention and thus making the elision of those locks more beneficial. We expect this to have less of an effect on B^{link}-trees, were latches are only held when a write actually occurs, but it might be

visible with latch coupling, were latches during write operations are held from the highest safe node downwards.

- B^{link}-trees in collaboration with Hyper-Threading (Subsection 6.5.3).
- Comparing B-trees (and CSB-trees) with different tree sizes. As larger trees in combination with latch coupling inevitably lead to larger transactions, we expect the abort rate to increase, resulting in any performance differences between HLE-enabled and normal binaries to be more pronounced (Subsection 6.5.4).
- Comparing different node sizes with B^{link}-trees. While we have seen in Section 6.2 that there is a sweet spot for the node size at 1024 B, using HLE could shift that to larger sizes because the transaction overhead is reduced while at the same time not sacrificing throughput due to latch contention.

6.5.1 Read-Only Workload

In this benchmark, 20,000,000 read operations are performed in trees which contain 4,000,000 entries. The results can be found in Figure 6.8.



Entries	Operations	Update rate	Threads
4,000,000	20,000,000	0 %	*variable*

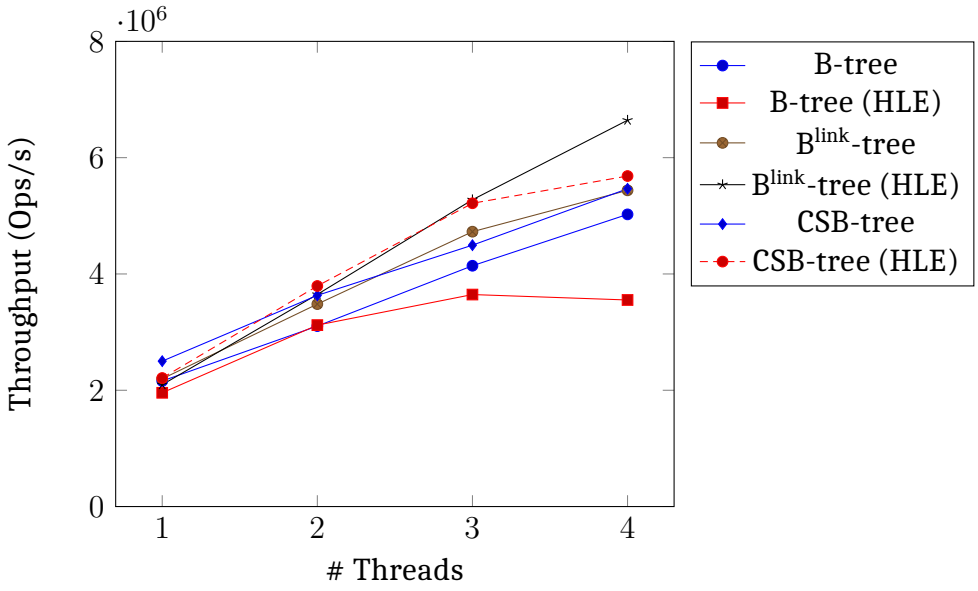
Figure 6.8: Using HLE

This benchmark shows HLE to have a performance benefit in every type of tree when running more than one thread. Especially amenable to HLE is the B^{link}-tree,

where it leads to unparalleled throughput. Also, adding more threads with the HLE-enhanced B^{link}-tree does not even seem to have diminishing returns.

6.5.2 Write-Heavy Workload

In this benchmark, 20,000,000 operations, 25 % of which are writes, are performed in trees storing 4,000,000 entries: The results can be found in Figure 6.9.

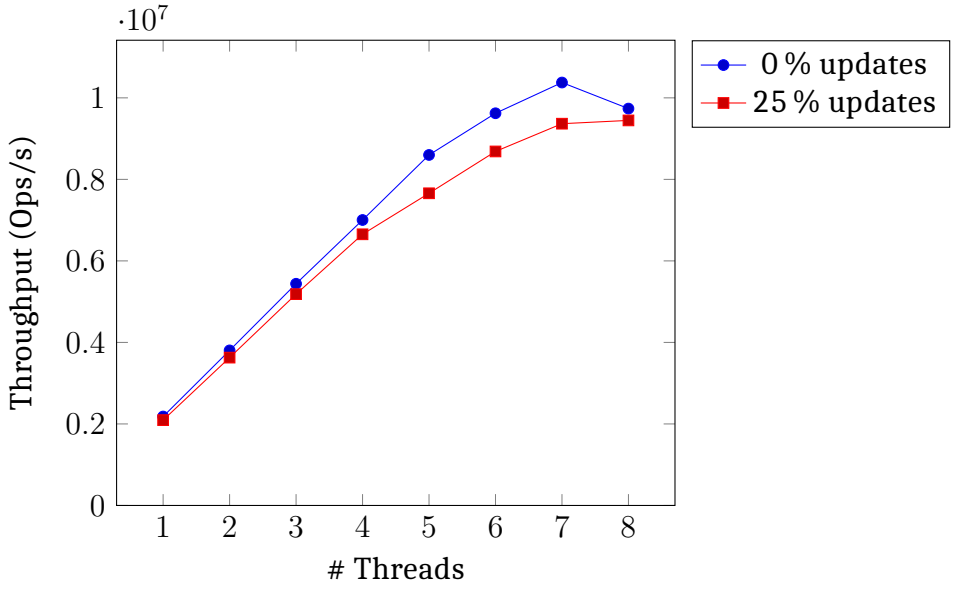


Entries	Operations	Update rate	Threads
4,000,000	20,000,000	25 %	*variable*

Figure 6.9: Using HLE

For this workload, the results deviate interestingly from the read-only workload. With the B-tree, the effect of adding HLE is reversed, leading to worse performance instead of better. This is likely because the high update rate causes more conflicts between transactions, thus leading to many aborts. In the CSB-tree, adding HLE makes very little difference, but the performance still goes up instead of down. We suspect that, while similar effects as in the B-tree should be at play, the more cache-friendly memory access patterns and space utilization might lead to fewer aborts.

Most interestingly, while the higher update rate causes diminishing returns for the B^{link}-tree with four threads, HLE completely negates that effect and the performance is almost on par with the read-only workload.



Entries	Operations	Update rate	Threads
4,000,000	20,000,000	*variable*	*variable*

Figure 6.10: Hyper-Threading in a B^{link}-tree (HLE)

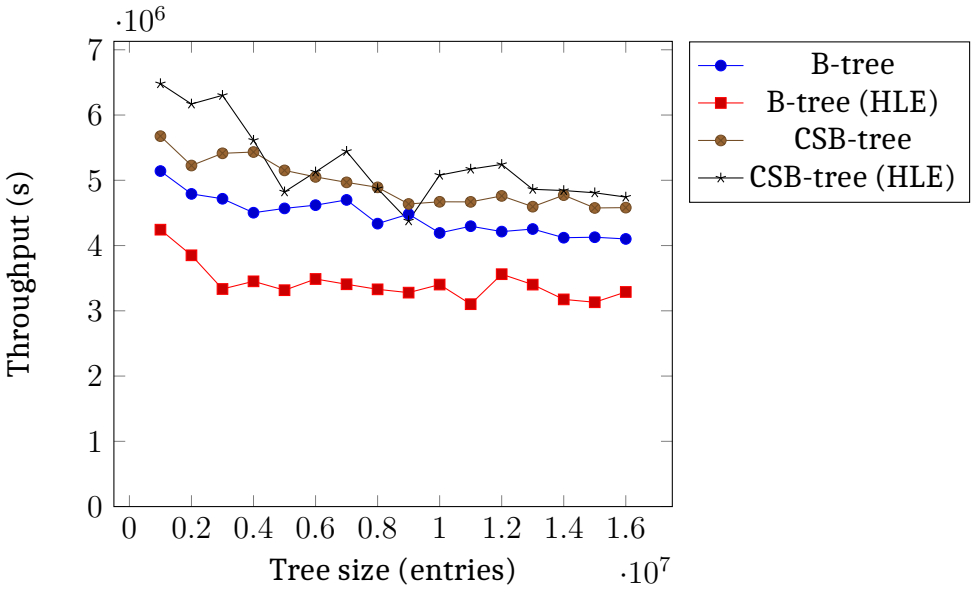
6.5.3 Effects of Hyper-Threading

Similar to Section 6.4, we looked at the effects of using two threads per core to measure the impact of Hyper-Threading. In this benchmark, 20,000,000 operations are performed in B^{link}-trees storing 4,000,000 entries: The results can be found in Figure 6.10. Again we have only run the benchmark with the B^{link}-tree because of its superior scalability. The results corroborate the earlier findings about scalability of the B^{link}-tree and show that even with a high update rate, multiple threads do not cause a major drop in performance. In addition, with Hyper-Threading the performance still goes up until seven threads are used and drops only with an eighth thread. This means that the performance drop with Hyper-Threading in Figure 6.7 is presumably *entirely* caused by locking, because otherwise the tree implementations and the workload is exactly the same.

6.5.4 HLE and Latch Coupling

In this benchmark we tried to ascertain whether the latch coupling protocol interacts negatively with HLE. At least in theory, because a new latch is acquired before the last one is released, a transaction eliding the locks can only be committed once the last latch is released. Thus, in a search, a transaction should span over the whole search. This in turn would mean that there exists a tree size where searches from root to

leaf cannot be handled transactionally and thus cause a visible drop in performance compared to when not using HLE, because every transaction is aborted. Also, there should exist a tree height at which the nested starting of transactions should exceed the nesting limit we determined in the microbenchmarks (Subsection 4.6.3). The results can be found in Figure 6.11.



Entries	Operations	Update rate	Threads
variable	5 x |entries|	25 %	4

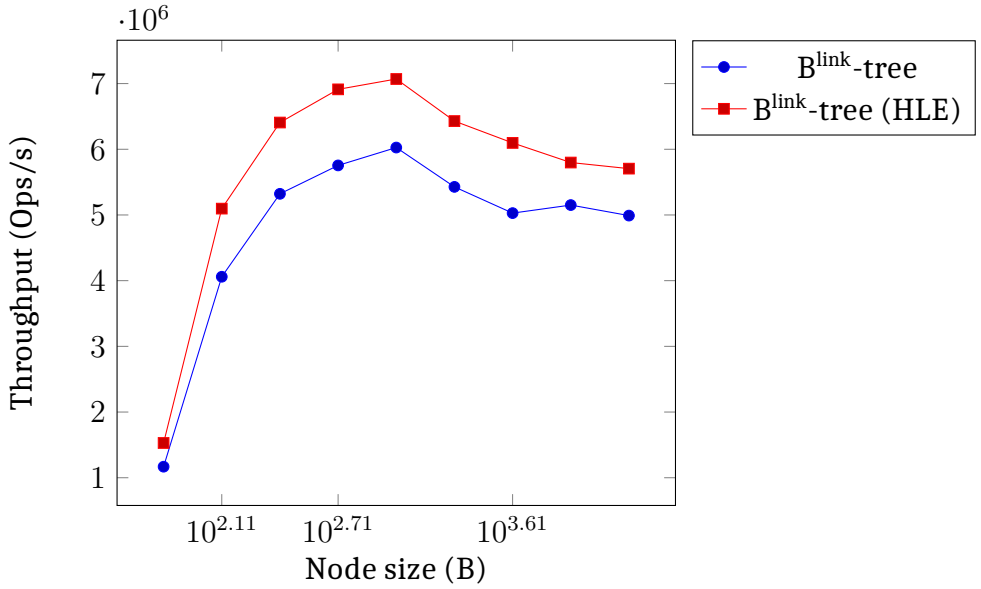
Figure 6.11: HLE with different tree sizes

Unfortunately we could find no such limit. The performance of CSB-trees with HLE was almost always above the performance without HLE, even in the largest trees (containing 16,000,000 entries). This means that in this range, no size or height exists that would guarantee transactions to be aborted. Data for larger trees could not be created due to hardware constraints.

With regular B-trees, the results are less sound, because HLE is always slower. This is surprising, because in some ways the node layout is very similar to CSB-trees:

- After determining the optimal node size, B-tree nodes are double the size of CSB-tree nodes (1024 B vs. 512 B), but the key arrays in which the searches are performed have roughly the same size because of the more compact layout of CSB-trees.

- With smaller tree sizes, if CSB-trees are faster with HLE, B-trees should be faster as well, due to having a similar height.



Entries	Operations	Update rate	Threads
4,000,000	20,000,000	25 %	4

Figure 6.12: Node sizes for Blink-trees with HLE

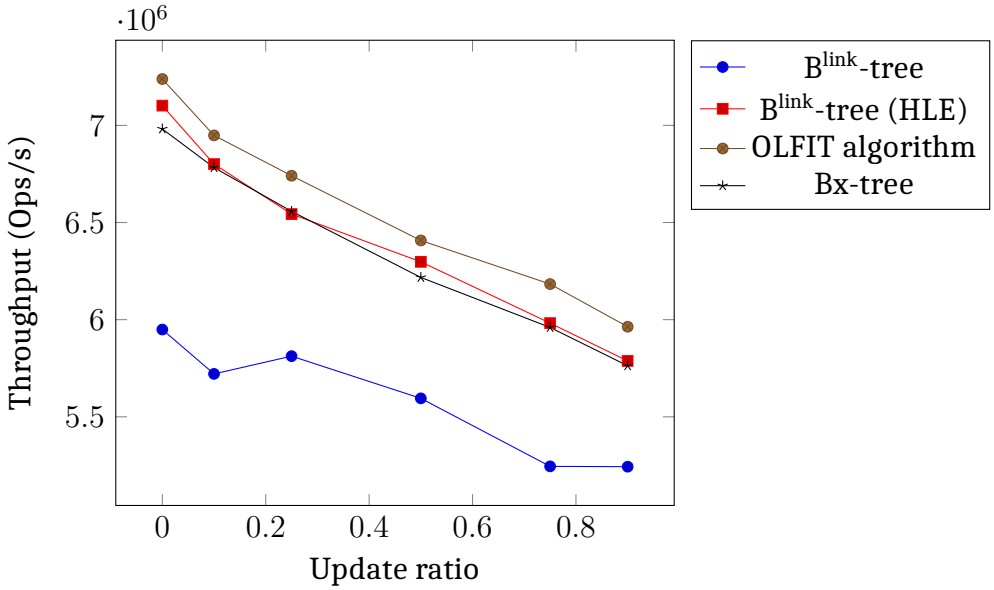
Therefore, because both use latch coupling, they both should display the performance loss we expected to see with HLE, or neither should.

6.5.5 HLE and Optimal Node Size

With this benchmark we wanted to check whether using HLE has an effect on the optimal node size. We ran the be benchmark with B^{link}-trees, because they don't display the inconsistencies discussed in the previous benchmark.

The results can be found in Figure 6.12. The tree with HLE was faster with every node size, which is not immediately obvious. With very small nodes, we expected the overhead of starting transactions in every node to cause a significant reduction in throughput, but this has not happened. The smallest tested size was 64 B, which is the size of a cache line. In this case, prefetching causes cache lines to be loaded that are not being used. However, even then HLE was slightly faster.

Also, unexpectedly, the global maximum is the same. On the one hand, this is in line with several previous benchmarks where we found the size of transactions not to have much impact on performance either way. Yet it is still counter-intuitive. As already mentioned, we would have expected latch contention to be a significant factor in where the optimal node size lies. Because with HLE that factor is largely gone, the optimal value could have changed.



Entries	Operations	Update rate	Threads
4,000,000	20,000,000	*variable*	4

Figure 6.13: Different update rates

6.6 Evaluation of the Bx-Tree

After having studied the possibilities of Hardware Lock Elision, we will now see how Restricted Transactional Memory in the Bx-tree fares. This section will also include the B^{link}-tree with the OLFIT algorithm (see Section 2.7) in the benchmarks. To evaluate these trees, we performed the following benchmarks:

1. Measuring throughput with four threads, dependent on update rate (Subsection 6.6.1).

2. Testing different retry counts for the Bx-tree (Subsection 6.6.2).

6.6.1 Varying Update Rates

In this first benchmark, we look at the performance of several trees when dealing with workloads having variable update rates. The implementations under scrutiny will be: the B^{link}-tree (with and without HLE), which outperformed the other trees in the previous benchmarks in almost all scenarios; the B^{link}-tree with the OLFIT algorithm, as an example of a state-of-the-art B-tree for OLTP workloads; and the Bx-tree with RTM. The results can be found in Figure 6.13. Unsurprisingly, the "traditional" B^{link}-tree is no match for the other implementations, which are geared towards these

workloads. The OLFIT algorithm dominates the other trees in every scenario, including the B^{link}-tree with HLE. As they mainly differ in the algorithms and not in the data structure, this contrast has to be attributed to these main differences:

- The OLFIT algorithm retries reads until they succeed without a version mismatch; the transactional B^{link}-tree retries reads only once and then locks. This means that a locked node or an abort cause accesses to that node to be serialized as in a non-transactional B^{link}-tree, whereas in the OLFIT algorithm, reads always happen in parallel.
- The transactional B^{link}-tree has the slightly higher overhead of starting and stopping transactions, compared to just reading a version number.

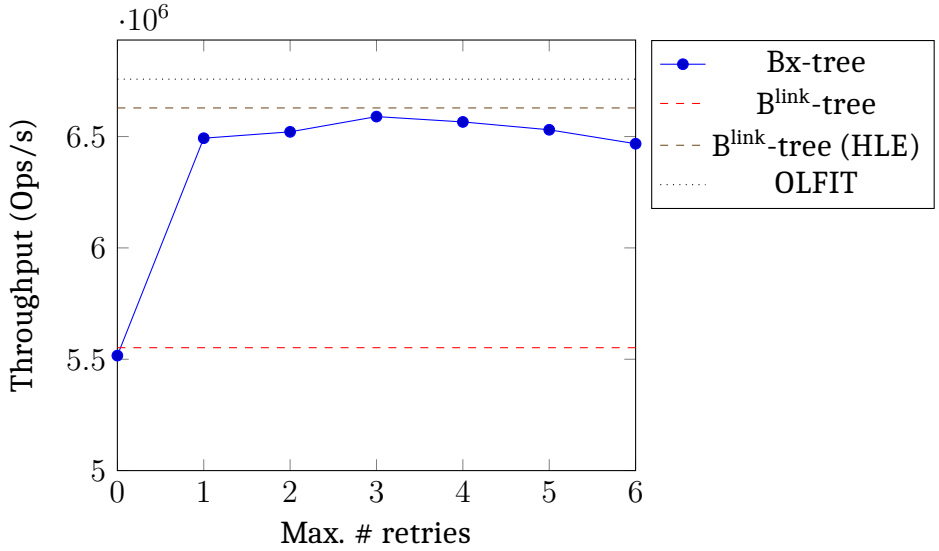
The Bx-tree performs better than the B^{link}-tree, showing that TSX in this case is a viable alternative to fine-grained latching. In contrast to the remaining two trees, however, a search has then to restart at the root instead of at the current node, in many cases.

There are also two general observations to be made:

- The transactional trees (we consider the OLFIT algorithm part of that group for the purpose of this comparison, because the reading of nodes based on a version number could be considered a form of software transactional memory) all outperform the B^{link}-tree, as they allow concurrent read access to nodes.
- This is also the reason why the B^{link}-tree is less impacted by the increasing write rate: the nodes are locked the same as before, only the pure cost of the operations change because of moving and copying of memory.

6.6.2 Varying Retry Counts

In this benchmark we look at the effect of varying the number of times a transaction is retried in the Bx-tree. The results can be found in Figure 6.14.



Entries	Operations	Update rate	Threads
4,000,000	20,000,000	25 %	4

Figure 6.14: Finding the optimal retry count

The results show the retry count not to be particularly important, as long as a transaction is attempted at all. For reference we have included the B-trees without a retry count, although in some sense

- the B^{link}-tree is a Bx-tree with a maximum retry count of 0,
- the B^{link}-tree is a Bx-tree with a maximum retry count of 1,
- and the OLFIT algorithm mimics a Bx-tree with infinite retries for reads.

Chapter 7

Summary and Outlook

In this final chapter we will briefly recapitulate our findings and then look at topics that warrant further research.

7.1 Summary

Optimizing in-memory OLTP databases for modern hardware is an important area of research. A current trend in CPU development, with frequency no longer being a primary target for improvement, is the inclusion of more and more cores on the one hand and extending the instruction set to cover many specialized use cases on the other hand. Scaling in-memory databases to these numbers of cores is a promising endeavor, as many tasks are easily parallelizable. With synchronization then becoming a bottleneck, hardware transactional memory seems like a prime opportunity to address that issue.

The B^{link}-tree then presents itself as a solid foundation on which to build optimized solutions for the new use cases. This is underlined by the fact that new designs like the Bw-tree and the OLFIT algorithm are based on the B^{link}-tree. Other ideas, such as the CSB-tree, concentrate on different areas for optimization.

Transactional memory, while having started out with a focus on simplifying the development of lock-free data structures, can now be considered a viable alternative for optimizing performance as well, when supported by the hardware.

Intel's Transactional Synchronization Extensions allow to simply apply transactional memory to existing data structures. In particular, we have demonstrated two advantages of HLE in B^{link}-trees: The throughput could be considerably increased, to the level of other modern concepts like the OLFIT algorithm; and we showed that HLE lets the B^{link}-tree scale much better with Hyper-Threading. RTM is also straight-

forward to use, but as our benchmarks showed, it may require considerably more analysis and fine-tuning if the goal is to achieve higher performance than with HLE or other competing B-tree designs.

7.2 Outlook

In this section we present open questions and issues that have arisen while working with TSX, as well as different paths for further development of B-trees using HTM and the Bx-tree in particular. We will begin by looking at more general issues with synchronization and concurrency. Then we will focus on theoretical aspects of evaluating TSX as a viable path forward and on practical ideas for further scrutinizing the Bx-tree.

7.2.1 Groundwork: Synchronization

One issue we have noticed while implementing the different B-trees is the innumerable, so to say, variety of latch types that can be considered for use in B-trees. This leads to a complex optimization problem, reinforced by the fact that many of the concepts encountered there are orthogonal to each other and can be combined into actual implementations:

- On a conceptual level, using either shared/exclusive or purely exclusive latches is a first differentiation.
- Then there is the question of which data types to use. While for the most basic latches boolean variables suffice, almost all more refined latches would require assembling the latch out of different parts, like combining an exclusive latch with a counter for counting threads in a shared latch, or implementing a fair latch by way of a ticket system or a linked list.
- Modifying these data types in the most effective way is another area that can be studied further, with metrics like CPU cycles for different atomic x86 instructions, or choosing the right memory model in more high-level code (for example where and when to insert memory barriers).
- Different strategies for latch acquisition also play a role, like the choice between test-and-set and test-and-test-and-set, or choosing a backoff strategy.

On top of that, there arises the question which of those implementations are possible to combine with Hardware Lock Elision.

7.2.2 Theoretical Side: Transactions in Hardware

One of the most important open questions after looking at the performance of the Bx-tree and the HLE-enabled B-trees is whether it would make sense to develop a theoretical cost model for hardware transactions. This entails at least two questions:

- What are the basic costs (in CPU cycles) of starting, committing and aborting transactions? Furthermore, do they differ between HLE and RTM? As our benchmarks of the B^{link}-tree showed, transactional access beats latching even in the smallest of nodes. As waiting for different latch implementations surely also requires more or less cycles, this leads to a complex comparison were different latches and their HLE-enabled counterparts can be analyzed in both theory and practice, factoring in variables like contention and number of threads.
- Can more general or reliable insights into success and abort rates and the causes for abort be obtained, leading to more tailored usage of transactions for different parts of the B-tree algorithms and different workloads?

7.2.3 Practical Side: Programming and Benchmarking

There are several enhancements to the implemented B-trees and the benchmark that would be useful in a broader analysis and comparison with other work in this area of research:

- Supporting range queries, which require more synchronization in B^{link}-trees than in B-trees.
- Making the Bx-tree (or the B^{link}-tree with HLE) viable for practical use, mainly by allowing different data types for keys and values.
- Benchmarking different access patterns, using different probability distributions for generating data, and adapting the trees to be used in recognized OLTP or in-memory database benchmarks.

When looking at the Bx-tree in particular, we have several ideas for enhancements that could be investigated:

- As the maximum size of a transaction is ultimately limited, so should the number of nodes be that the traversal algorithm attempts to read in a single transaction.

- The tree could be made more adaptive regarding the abort reasons. For example, when a transaction is aborted due to size, it could be retried with a more limited scope, accessing fewer nodes. When a transaction is aborted due to a node being locked, assuming the thread executing the operation is processing incoming requests in a row, the request could be deferred, in the hopes that a new request leads to a different part of the tree and the deferred transaction will succeed on a later attempt.

Bibliography

[Ail+99] Anastassia Ailamaki et al. “DBMSs on a Modern Processor: Where Does Time Go?” In: *Proceedings of the 25th International Conference on Very Large Data Bases*. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 266–277. URL: http://dl.acm.org/citation.cfm?id=645925.671662 (cit. on p. 1).

[BM72] R. Bayer and E.M. McCreight. “Organization and maintenance of large ordered indexes”. English. In: *Acta Informatica* 1.3 (1972), pp. 173–189. doi: 10.1007/BF00288683 (cit. on pp. 5, 6).

[Cha+01] Sang K. Cha et al. “Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems”. In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 181–190. URL: http://dl.acm.org/citation.cfm?id=645927.672375 (cit. on pp. 3, 17).

[Com79] Douglas Comer. “The Ubiquitous B-Tree”. In: *ACM Comput. Surv.* 11.2 (June 1979), pp. 121–137. doi: 10.1145/356770.356776 (cit. on p. 6).

[GNU13] GNU Project. *GCC 4.8 Release Series. Changes, New Features, and Fixes*. 2013. URL: https://gcc.gnu.org/gcc-4.8/changes.html (visited on 03/15/2016) (cit. on p. 23).

[Goe+14] B. Goel et al. “Performance and Energy Analysis of the Restricted Transactional Memory Implementation on Haswell”. In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. May 2014, pp. 615–624. doi: 10.1109/IPDPS.2014.70 (cit. on p. 35).

[Har+05] Tim Harris et al. “Composable Memory Transactions”. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. Chicago, IL, USA: ACM, 2005, pp. 48–60. doi: 10.1145/1065944.1065952 (cit. on pp. 19, 21).

[Her93] Maurice Herlihy. “A Methodology for Implementing Highly Concurrent Data Objects”. In: *ACM Trans. Program. Lang. Syst.* 15.5 (Nov. 1993), pp. 745–770. doi: 10.1145/161468.161469 (cit. on p. 21).

[HM93] Maurice Herlihy and J. Eliot B. Moss. “Transactional Memory: Architectural Support for Lock-free Data Structures”. In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ISCA '93. San Diego, California, USA: ACM, 1993, pp. 289–300. doi: 10.1145/165123.165164 (cit. on p. 19).

[INTC12a] Intel Corporation. *Coarse-grained locks and Transactional Synchronization explained*. 2012. URL: https://software.intel.com/en-us/blogs/2012/02/07/coarse-grained-locks-and-transactional-synchronization-explained (visited on 03/13/2016) (cit. on p. 23).

[INTC12b] Intel Corporation. *Transactional Synchronization in Haswell*. 2012. URL: https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell (visited on 03/13/2016) (cit. on p. 23).

[INTC12c] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*. 2012 (cit. on p. 23).

[INTC15a] Intel Corporation. *Intel® Core™ i7-6700T Processor (8M Cache, up to 3.60 GHz) Specifications*. 2015. URL: http://ark.intel.com/products/88200/Intel-Core-i7-6700T-Processor-8M-Cache-up-to-3_60-GHz (visited on 03/11/2016) (cit. on p. 32).

[INTC15b] Intel Corporation. *Intel Software Developers Manual Volume 2A: Instruction Set Reference, A-M*. 2015 (cit. on p. 24).

[INTC16] Intel Corporation. *Intel Xeon Processor E3-1200 v3 Product Family (Specification Update)*. 2016 (cit. on p. 23).

[Joh+10] Ryan Johnson et al. “Decoupling contention management from scheduling”. In: *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*. 2010, pp. 117–128. doi: 10.1145/1736020.1736035 (cit. on p. 11).

[Kan12] David Kanter. *Analysis of Haswell’s Transactional Memory*. 2012. URL: http://www.realworldtech.com/haswell-tm/ (visited on 03/11/2016) (cit. on pp. 25, 32).

[LKN13] Viktor Leis, Alfons Kemper, and Thomas Neumann. “The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases”. In: *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*. ICDE '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 38–49. doi: 10.1109/ICDE.2013.6544812 (cit. on p. 1).

[LLS13] J. J. Levandoski, D. B. Lomet, and S. Sengupta. “The Bw-Tree: A B-tree for new hardware platforms”. In: *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. Apr. 2013, pp. 302–313. doi: 10.1109/ICDE.2013.6544834 (cit. on p. 15).

[LY81] Philip L. Lehman and s. Bing Yao. “Efficient Locking for Concurrent Operations on B-trees”. In: *ACM Trans. Database Syst.* 6.4 (Dec. 1981), pp. 650–670. doi: 10.1145/319628.319663 (cit. on p. 3).

[Mil13] David Miller. *The GNU C Library version 2.18 is now available*. 2013. URL: https://sourceware.org/ml/libc-alpha/2013-08/msg00160.html (visited on 03/15/2016) (cit. on p. 23).

[MLS15] Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. “To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-free Indexing”. In: *Proc. VLDB Endow.* 8.11 (July 2015), pp. 1298–1309. doi: 10.14778/2809974.2809990 (cit. on pp. 37, 43).

[PSQL15] PostgreSQL Documentation. *Index Types*. 2015. URL: http://www.postgresql.org/docs/9.5/static/indexes-types.html (visited on 03/29/2016) (cit. on p. 5).

[RR00] 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. Dallas, Texas, USA: ACM, 2000, pp. 475–486. doi: 10.1145/342009.335449 (cit. on pp. 3, 16).

[ST95] Nir Shavit and Dan Touitou. “Software Transactional Memory”. In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '95. Ottowa, Ontario, Canada: ACM, 1995, pp. 204–213. doi: 10.1145/224964.224987 (cit. on p. 19).

[Wik15] WiredTiger Wiki. *Btree vs LSM*. 2015. URL: https://github.com/wiredtiger/wiredtiger/wiki/Btree-vs-LSM (visited on 03/29/2016) (cit. on p. 5).

[WV01] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. URL: http://dl.acm.org/citation.cfm?id=378243 (cit. on p. 12).