# Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method

Viktor Leis, Michael Haubenschild*, Thomas Neumann
Technische Universität München Tableau Software*
{leis,neumann}@in.tum.de mhaubenschild@tableau.com*

**Abstract**

*As the number of cores on commodity processors continues to increase, scalability becomes more and more crucial for overall performance. Scalable and efficient concurrent data structures are particularly important, as these are often the building blocks of parallel algorithms. Unfortunately, traditional synchronization techniques based on fine-grained locking have been shown to be unscalable on modern multi-core CPUs. Lock-free data structures, on the other hand, are extremely difficult to design and often incur significant overhead.*

*In this work, we make the case for Optimistic Lock Coupling as a practical alternative to both traditional locking and the lock-free approach. We show that Optimistic Lock Coupling is highly scalable and almost as simple to implement as traditional lock coupling. Another important advantage is that it is easily applicable to most tree-like data structures. We therefore argue that Optimistic Lock Coupling, rather than a complex and error-prone custom synchronization protocol, should be the default choice for performance-critical data structures.*

## 1 Introduction

Today, Intel's commodity server processors have up to 28 cores and its upcoming microarchitecture will have up to 48 cores per socket [6]. Similarly, AMD currently stands at 32 cores and this number is expected to double in the next generation [20]. Since both platforms support simultaneous multithreading (also known as hyperthreading), affordable commodity servers (with up to two sockets) will soon routinely have between 100 and 200 hardware threads.

With such a high degree of hardware parallelism, efficient data processing crucially depends on how well concurrent data structures scale. Internally, database systems use a plethora of data structures like table heaps, internal work queues, and, most importantly, index structures. Any of these can easily become a scalability (and therefore overall performance) bottleneck on many-core CPUs.

Traditionally, database systems synchronize internal data structures using fine-grained reader/writer locks[1]. Unfortunately, while fine-grained locking makes lock contention unlikely, it still results in bad scalability because


**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

[1]In this work, we focus on data structure synchronization rather than high-level transaction semantics and therefore use the term *lock* for what would typically be called *latch* in the database literature. We thus follow common computer science (rather than database) terminology.

lock acquisition and release require writing to shared memory. Due to the way cache coherency is implemented on modern multi-core CPUs, these writes cause additional cache misses[2] and the cache line containing the lock's internal data becomes a point of physical contention. As a result, any frequently-accessed lock (e.g., the lock of the root node of a B-tree) severely limits scalability.

Lock-free data structures like the Bw-tree [15] (a lock-free B-tree variant) or the Split-Ordered List [19] (a lock-free hash table) do not acquire any locks and therefore generally scale much better than locking-based approaches (in particular for read-mostly workloads). However, lock-free synchronization has other downsides: First, it is very difficult and results in extremely complex and error-prone code (when compared to locking). Second, because the functionality of atomic primitives provided by the hardware (e.g., atomically compare-and-swap 8 bytes) is limited, complex operations require additional indirections within the data structure. For example, the Bw-tree requires an indirection table and the Split-Ordered List requires "dummy nodes", resulting in overhead due to additional cache misses.

In this paper we make the case for *Optimistic Lock Coupling (OLC)*, a synchronization method that combines some of the best properties of lock-based and lock-free synchronization. OLC utilizes a special lock type that can be used in two modes: The first mode is similar to a traditional mutex and excludes other threads by physically acquiring the underlying lock. In the second mode, reads can proceed optimistically by validating a version counter that is embedded in the lock (similar to optimistic concurrency control). The first mode is typically used by writers and the second mode by readers. Besides this special lock type, OLC is based on the observation that optimistic lock validations can be interleaved/coupled—similar to the pair-wise interleaved lock acquisition of traditional lock coupling. Hence, the name Optimistic Lock Coupling.

OLC has a number of desirable features:

- By reducing the number of writes to shared memory locations and thereby avoiding cache invalidations, it **scales well** for most workloads.
- In comparison to unsynchronized code, it requires few additional CPU instructions making it **efficient**.
- OLC is **widely applicable** to different data structures. It has already been successfully used for synchronizing binary search trees [4], tries [14], trie/B-tree hybrids [17], and B-trees [22].
- In comparison to the lock-free paradigm, it is also **easy to use** and requires few modifications to existing, single-threaded data structures.

Despite these positive features and its simplicity, OLC is not yet widely known. The goal of this paper is therefore to popularize this simple idea and to make a case for it. We argue that OLC deserves to be widely known. It is a good default synchronization paradigm—more complex, data structure-specific protocols are seldom beneficial.

The rest of the paper is organized as follows. Section 2 discusses related work, tracing the history of OLC and its underlying ideas in the literature. The core of the paper is Section 3, which describes the ideas behind OLC and how it can be used to synchronize complex data structures. In Section 4 we experimentally show that OLC has low overhead and scales well when used to synchronize an in-memory B-tree. We summarize the paper in Section 5.

[2]The cache coherency protocol ensures that all copies of a cache line on other cores are invalidated before the write can proceed.

## 2 Related Work

Lock coupling has been proposed as a method for allowing concurrent operations on B-trees in 1977 [2]. This traditional and still widely-used method, described in detail in Graefe's B-tree survey [8], is also called "latch coupling", "hand-over-hand locking", and "crabbing". Because at most two locks are held at-a-time during tree traversal, this technique seemingly allows for a high degree of parallelism—in particular if read/write locks are used to enable inner nodes to be locked in shared mode. However, as we show in Section 4, on modern hardware lock acquisition (even in shared mode) results in suboptimal scalability.

An early alternative from 1981 is a B-tree variant called B-link tree [10], which only holds a single lock at a time. It is based on the observation that between the release of the parent lock and the acquisition of the child lock, the only "dangerous" thing that could have happened is the split of a child node (assuming one does not implement merge operations). Thus, when a split happens, the key being searched might end up on a neighboring node to the right of the current child node. A B-link tree traversal therefore detects this condition and, if needed, transparently proceeds to the neighboring node. Releasing the parent lock early is highly beneficial when the child node needs to be fetched from disk. For in-memory workloads, however, the B-link tree has the same scalability issues as lock coupling (it acquires just as many locks).

The next major advance, Optimistic Latch-Free Index Traversal (OLFIT) [5], was proposed in 2001. OLFIT introduced the idea of a combined lock/update counter, which we call *optimistic lock*. Based on these per-node optimistic locks and the synchronization protocol of the B-link tree, OLFIT finally achieves good scalability on parallel processors. The OLFIT protocol is fairly complex, as it requires both the non-trivial B-link protocol and optimistic locks. Furthermore, like the B-link tree protocol, it does not support merging nodes, and is specific to B-trees (cannot easily be applied to other data structures).

In the following two decades, the growth of main-memory capacity led to much research into other data structures besides the venerable B-tree. Particularly relevant for our discussion is Bronson et al.'s [4] concurrent binary search tree, which is based on optimistic version validation and has a sophisticated, data structure-specific synchronization protocol. To the best of our knowledge, this 2010 paper is the first that, as part of its protocol, interleaves version validation across nodes—rather than validating each node separately like OLFIT. In that paper, this idea is called "hand-over-hand, optimistic validation", while we prefer the term Optimistic Lock Coupling to highlight the close resemblance to traditional lock coupling. Similarly, Mao et al.'s [17] Masstree (a concurrent hybrid trie/B-tree) is also based on the same ideas, but again uses them as part of a more complex protocol.

The Adaptive Radix Tree (ART) [12] is another recent in-memory data structure, which we proposed in 2013. In contrast to the two data structures just mentioned, it was originally designed with single-threaded performance in mind without supporting concurrency. To add support for concurrency, we initially started designing a custom protocol called Read-Optimized Write Exclusion (ROWEX) [14], which turned out to be non-trivial and requires modifications of the underlying data structure[3]. However, fairly late in the project, we also realized, that OLC *alone* (rather than as part of a more complex protocol) is sufficient to synchronize ART. No other changes to the data structure were necessary. Both approaches were published and experimentally evaluated in a followup paper [14], which shows that, despite its simplicity, OLC is efficient, scalable, and generally outperforms ROWEX.

Similar results were recently published regarding B-trees [22]. In this experimental study a simple OLC-based synchronization outperformed the Bw-tree [15], a complex lock-free synchronization approach. Another recent paper shows that for write-intensive workloads, locking often performs better than lock-free synchronization [7]. These experiences indicate that OLC is a general-purpose synchronization paradigm and motivate the current paper.

[3]Note that ROWEX is already easier to apply to existing data structures than the lock-free approach. The difficulty depends on the data structure. Applying ROWEX is hard for B-trees with sorted keys and fairly easy for copy-on-write data structures like the Height Optimized Trie [3]—with ART being somewhere in the middle.

# 3 Optimistic Lock Coupling

The standard technique for inter-thread synchronization is mutual exclusion using fine-grained locks. In a B-tree, for example, every node usually has its own associated lock, which is acquired before accessing that node. The problem of locking on modern multi- and many-core processors is that lock acquisition and release require writing to the shared memory location that implements the lock. This write causes exclusive ownership of the underlying cache line and invalidates copies of it on all other processor cores. For hierarchical, tree-like data structures, the lock of the root node becomes a point of physical contention—even in read-only workloads and even when read/write locks are used. Depending on the specific data structure, number of cores, cache coherency protocol implementation, cache topology, whether Non-Uniform Memory Access (NUMA) is used, locking can even result in multi-threaded performance that is worse than single-threaded execution.

The inherent pessimism of locking is particularly unfortunate for B-trees: Despite the fact that logical modifications of the root node are very infrequent, every B-tree operation must lock the root node during tree traversal[4]. Even the vast majority of update operations (with the exception of splits and merges), only modify a single leaf node. These observations indicate that a more optimistic approach, which does not require locking inner nodes, would be very beneficial for B-trees.

## 3.1 Optimistic Locks

As the name indicates, optimistic locks try to solve the scalability issues of traditional locks using an optimistic approach. Instead of always physically acquiring locks, even for nodes that are unlikely to be modified simultaneously, after-the-fact validation is used to detect conflicts. This is done by augmenting each lock with a version/update counter that is incremented on every modification. Using this version counter, readers can optimistically proceed before validating that the version did not change to ensure that the read was safe. If validation fails, the operation is restarted.

Using optimistic locks, a read-only node access (i.e., the majority of all operations in a B-tree) does not acquire the lock and does not increment the version counter. Instead, it performs the following steps:

1. read lock version (restart if lock is not free)
2. access node
3. read the version again and validate that it has not changed in the meantime

If the last step (the validation) fails, the operation has to be restarted. Write operations, on the other hand, are more similar to traditional locking:

1. acquire lock (wait if necessary)
2. access/write to node
3. increment version and unlock node

Writes can therefore protect a node from other writes.

As we observed in an earlier paper [14], because of similar semantics, optimistic locks can be hidden behind an API very similar to traditional read/write locks. Both approaches have an exclusive lock mode, and acquiring a traditional lock in shared mode is analogous to optimistic version validation. Furthermore, like with some implementations of traditional read/write locks, optimistic locks allow upgrading a shared lock to an exclusive lock. Lock upgrades are, for example, used to avoid most B-tree update operations from having to lock inner nodes. In our experience, the close resemblance of optimistic and traditional locks simplifies the reasoning about optimistic locks; one can apply similar thinking as in traditional lock-based protocols.

[4] To a lesser extent this obviously applies to all inner nodes, not just the root.

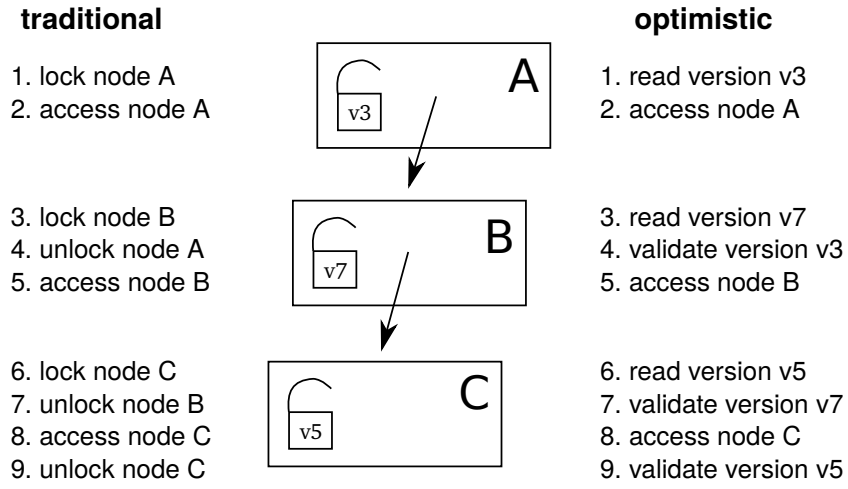


Figure 1: Comparison of a lookup operation in a 3-level tree using traditional lock coupling (left-hand side) vs. optimistic lock coupling (right-hand side).

### 3.2 Lock Coupling with Optimistic Locks

The traditional and most common lock-based synchronization protocol for B-trees is lock coupling, which interleaves lock acquisitions while holding at most two locks at a time. If, as we observed earlier, optimistic locks have similar semantics as traditional locks, it is natural to ask whether lock coupling can be combined with optimistic locks. And indeed the answer is yes: One can almost mechanically translate traditional lock coupling code to optimistic lock coupling code. This is illustrated in Figure 1, which compares the traversal in a tree of height 3 using traditional and optimistic locks. As the figure shows, the main difference is that locking is translated to reading the version and that unlocking becomes validation of the previously read version. This simple change provides efficient lock-free tree traversal without the need to design a complex synchronization protocol.

It is important to emphasize the conceptual simplicity of OLC in comparison to data structures that use custom protocols like the Bw-tree [15]. To implement lock-free access, the Bw-tree requires an indirection table, delta nodes, complex splitting and merging logic, retry logic, etc. OLC, on the other hand, can directly be applied to B-trees mostly by adding the appropriate optimistic locking code and without modifying the node layout itself. Therefore, OpenBw-Tree, an open source implementation of the Bw-tree, requires an order of magnitude more code than a B-tree based on OLC[5]. Given how difficult it is to develop, validate, and debug lock-free code, simplicity is obviously a major advantage.

### 3.3 Correctness Aspects

So far, we have introduced the high-level ideas behind OLC and have stressed its similarity to traditional lock coupling. Let us now discuss some cases where the close similarity between lock coupling and OLC breaks down. To make this more concrete, we show the B-tree lookup code in Figure 2. In the code, `readLockOrRestart` reads the lock version and `readUnlockOrRestart` validates that the read was correct.

One issue with OLC is that any pointer speculatively read from a node may point to invalid memory (if that node is modified concurrently). Dereferencing such a pointer (e.g., to read its optimistic lock), may cause a segmentation fault or undefined behavior. In the code shown in Figure 2, this problem is prevented by the extra check in line 25, which ensures that the read from the node containing the pointer was correct. Without this

[5]Both implementations are available on GitHub: `https://github.com/wangziqi2016/index-microbench`

```
1  std::atomic<BTreeNode*> root;
2
3  // search for key in B+tree, returns payload in resultOut
4  bool lookup(Key key, Value& resultOut) {
5     BTreeNode* node = root.load();
6     uint64_t nodeVersion = node->readLockOrRestart();
7     if (node != root.load()) // make sure the root is still the root
8        restart();
9
10    BTreeInner<Key>* parent = nullptr;
11    uint64_t parentVersion = 0;
12
13    while (node->isInner()) {
14       auto inner = (BTreeInner*)node;
15
16       // unlock parent and make current node the parent
17       if (parent)
18          parent->readUnlockOrRestart(parentVersion);
19       parent = inner;
20       parentVersion = nodeVersion;
21
22       // search for next node
23       node = inner->findChild(key);
24       // validate 'inner' to ensure that 'node' pointer is valid
25       inner->checkOrRestart(nodeVersion);
26       // now it safe to dereference 'node' pointer (read its version)
27       nodeVersion = node->readLockOrRestart();
28    }
29
30    // search in leaf and retrieve payload
31    auto leaf = (BTreeLeaf*)node;
32    bool success = leaf->findValue(key, resultOut);
33
34    // unlock everything
35    if (parent)
36       parent->readUnlockOrRestart(parentVersion);
37    node->readUnlockOrRestart(nodeVersion);
38
39    return success;
40 }
```

Figure 2: B-tree lookup code using OLC. For simplicity, the restart logic is not shown.

additional validation, the code would in line 27 dereference the pointer speculatively read in line 23. Note that the implementation of `checkOrRestart` is actually identical to `readUnlockOrRestart`. We chose to give it a different name to highlight the fact that this extra check would not be necessary with read/write locks.

Another potential issue with optimistic locks is code that does not terminate. Code that speculatively accesses a node, like an intra-node binary search, should be written in a way such that it always terminates—even in the presence of concurrent writes. Otherwise, the validation code that detects the concurrent write will never run. The binary search of a B-tree, for example, needs to be written in such a way that each comparison makes progress. For some data structures that do not require loops in the traversal code (like ART) termination is trivially true.

### 3.4 Implementation Details

To implement an optimistic lock, one can combine the lock and the version counter into a single 64-bit[6] word [14]. A typical read operation will therefore merely consist of reading this version counter atomically. In C++11 this can be implemented using the `std::atomic` type.

On x86, atomic reads are cheap because of x86's strong memory order guarantees. No memory fences are required for sequentially-consistent loads, which are translated (by both GCC and clang) into standard `MOV` instructions. Hence, the only effect of `std::atomic` for loads is preventing instruction re-ordering. This makes version access and validation cheap. Acquiring and releasing an optimistic lock in exclusive mode has comparable cost to a traditional lock: A fairly expensive sequentially-consistent store is needed for acquiring a lock, while a standard `MOV` suffices for releasing it. A simple sinlock-based implementation of optimistic locks can be found in the appendix of an earlier paper [14].

OLC code must be able to handle restarts since validation or lock upgrade can fail due to concurrent writers. Restarts can easily be implemented by wrapping the data structure operation in a loop (for simplicity not shown in Figure 2). Such a loop also enables limiting the number of optimistic retry operations and falling back to pessimistic locking in cases of very heavy contention. The ability to fall back to traditional locking is a major advantage of OLC in terms of robustness over lock-free approaches, which do not have this option.

In addition to the optimistic shared mode and the exclusive mode, optimistic locks also support a "shared pessimistic" mode, which physically acquires the lock in shared mode (allowing multiple concurrent readers but no writers). This mode is useful for table (or range) scans that touch many tuples on a leaf page (which would otherwise easily abort). Finally, let us mention that large range scans and table scans, should be broken up into several per-node traversals as is done in the LeanStore [11] system.

Like all lock-free data structures, but unlike traditional locking and Hardware Transactional Memory [9, 16, 13], OLC requires care when deleting (and reusing) nodes. The reason is that a deleting thread can never be sure that a node can be reclaimed because other threads might still be optimistically reading from that node. Therefore, standard solutions like epoch-based reclamation [21], hazard pointers [18], or optimized hazard pointers [1] need to be used. These memory reclamation techniques are, however, largely orthogonal to the synchronization protocol itself.

[6] Even after subtracting one bit for the lock status, a back-of-the-envelope calculation can show that 63 bits are large enough to never overflow in practice.

Table 4: Performance and CPU counters for lookup and insert operations in a B-tree with 100M keys. We perform 100M operations and normalize the CPU counters by that number.

| | threads | M op/s | cycles | instruc-tions | L1 misses | L3 misses | branch misses |
|---|---|---|---|---|---|---|---|
| lookup (no sync.) | 1 | 1.72 | 2028 | 283 | 39.1 | 14.9 | 16.1 |
| lookup (OLC) | 1 | 1.65 | 2107 | 370 | 43.9 | 15.1 | 16.7 |
| lookup (lock coup.) | 1 | 1.72 | 2078 | 365 | 42.3 | 16.9 | 15.7 |
| insert (no sync.) | 1 | 1.51 | 2286 | 530 | 59.8 | 31.1 | 17.3 |
| insert (OLC) | 1 | 1.50 | 2303 | 629 | 61.2 | 31.1 | 16.5 |
| insert (lock coup.) | 1 | 1.41 | 2473 | 644 | 61.0 | 31.0 | 17.2 |
| lookup (no sync.) | 10 | 15.48 | 2058 | 283 | 38.6 | 15.5 | 16.0 |
| lookup (OLC) | 10 | 14.60 | 2187 | 370 | 43.8 | 15.8 | 16.8 |
| lookup (lock coup.) | 10 | 5.71 | 5591 | 379 | 54.2 | 17.0 | 14.8 |
| insert (no sync.) | 10 | - | - | - | - | - | - |
| insert (OLC) | 10 | 10.46 | 2940 | 656 | 62.0 | 32.5 | 16.8 |
| insert (lock coup.) | 10 | 7.55 | 4161 | 667 | 75.0 | 28.6 | 16.2 |

# 4 Evaluation

Let us now experimentally evaluate the overhead and scalability of OLC. For the experiments, we use an in-memory B+tree implemented in C++11 using templates, which is configured to use nodes of 4096 bytes, random 8 byte keys, and 8 byte payloads. Based on this B-tree, we compare the following synchronization approaches:

- an OLC implementation[7]
- a variant based on traditional lock coupling and read/write locks
- the unsynchronized B-tree, which obviously is only correct for read-only workloads but allows measuring the overhead of synchronization

Note that earlier work has compared the OLC implementation with a Bw-tree implementation [22] and other state-of-the-art in-memory index structures.

We use a Haswell EP system with an Intel Xeon E5-2687W v3 CPU, which has 10 cores (20 "Hyper-Threads") and 25 MB of L3 cache. The system is running Ubuntu 18.10 and we use GCC 8.2.0 to compile our code. The CPU counters are obtained using the Linux perf API[8].

Table 4 compares the performance and CPU counters for lookup and insert operations in a B-tree with 100M keys. With *single-threaded* execution, we observe that all three approaches have very similar performance. Adding traditional or optimistic locks to unsynchronized B-tree code results in up to 30% of additional instructions without affecting single-threaded performance much.

As Figure 3 shows, the results change dramatically once we use multiple threads. For lookup, the scalability of OLC is near-linear up to 20 threads, even though the system has only 10 "real cores". The OLC scalability for insert is also respectable (though not quite as linear because multi-threaded insertion approaches the memory bandwidth of our processor). The figure also shows that the results of traditional lock coupling with read/write

[7] An almost identical OLC implementation is available on github: `https://github.com/wangziqi2016/index-microbench/tree/master/BTreeOLC`

[8] We use the following convenience wrapper: `https://github.com/viktorleis/perfevent`

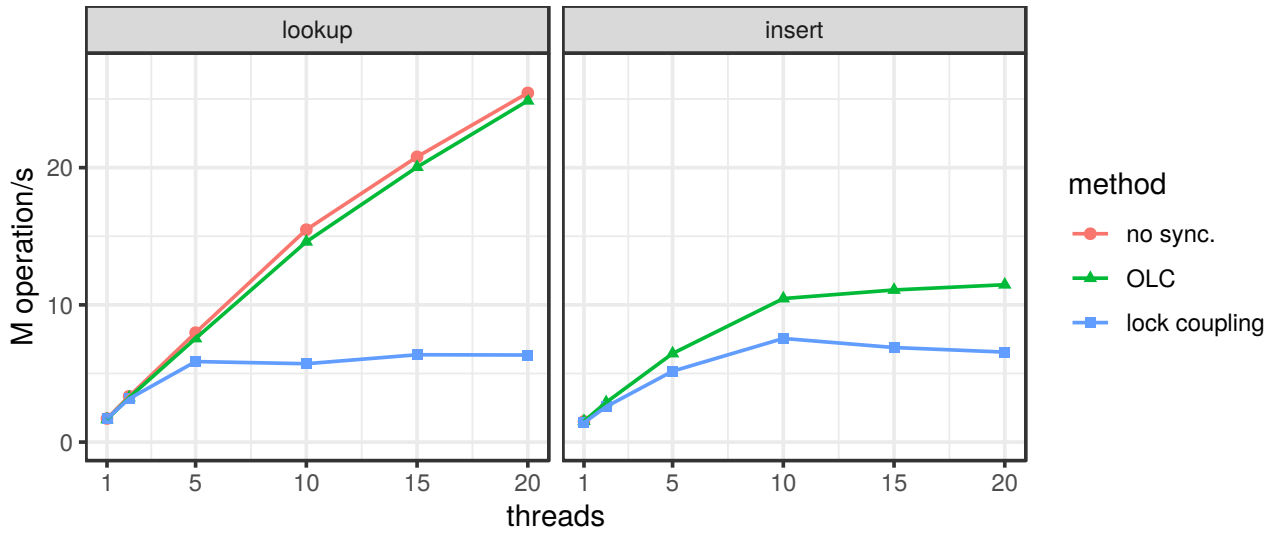


Figure 3: Scalability on 10-core system for B-tree operations (100M values).

locks are significantly worse than OLC. With 20 threads, lookup with OLC is 3.9× faster than traditional lock coupling.

## 5 Summary

Optimistic Lock Coupling (OLC) is an effective synchronization method that combines the simplicity of traditional lock coupling with the superior scalability of lock-free approaches. OLC is widely applicable and has already been successfully used to synchronize several data structures, including B-trees, binary search trees, and different trie variants. These features make it highly attractive for modern database systems as well as performance-critical systems software in general.

## References

[1] O. Balmau, R. Guerraoui, M. Herlihy, and I. Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *SPAA*, 2016.

[2] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9, 1977.

[3] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis. HOT: A height optimized trie index for main-memory database systems. In *SIGMOD*, 2018.

[4] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPOPP*, 2010.

[5] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, 2001.

[6] I. Cutress. Intel goes for 48-cores: Cascade-AP with multi-chip package coming soon. https://www.anandtech.com/show/13535/intel-goes-for-48cores-cascade-ap, 2018 (accessed January, 2019).

[7] J. M. Faleiro and D. J. Abadi. Latch-free synchronization in database systems: Silver bullet or fool's gold? In *CIDR*, 2017.

[8] G. Graefe. Modern B-tree techniques. *Foundations and Trends in Databases*, 3(4), 2011.

[9] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving in-memory database index performance with Intel® transactional synchronization extensions. In *HPCA*, 2014.

[10] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, 6(4), 1981.

[11] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. Leanstore: In-memory data management beyond main memory. In *ICDE*, 2018.

[12] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*, 2013.

[13] V. Leis, A. Kemper, and T. Neumann. Scaling HTM-supported database transactions to many cores. *IEEE Trans. Knowl. Data Eng.*, 28(2), 2016.

[14] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of practical synchronization. In *DaMoN*, 2016.

[15] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-tree: A B-tree for new hardware platforms. In *ICDE*, 2013.

[16] D. Makreshanski, J. J. Levandoski, and R. Stutsman. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *PVLDB*, 8(11), 2015.

[17] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[18] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6), 2004.

[19] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3), 2006.

[20] A. Shilov. AMD previews EPYC 'Rome' processor: Up to 64 Zen 2 cores. https://www.anandtech.com/show/13561/amd-previews-epyc-rome-processor-up-to-64-zen-2-cores, 2018 (accessed January, 2019).

[21] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[22] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. Andersen. Building a Bw-tree takes more than just buzz words. In *SIGMOD*, 2018.