Scalable and Robust Latches for Database Systems

Jan Böttcher Viktor Leis[★] Jana Giceva Thomas Neumann Alfons Kemper
Technische Universität München Friedrich-Schiller-Universität Jena[★]
{boettcher,giceva,neumann,kemper}@in.tum.de,viktor.leis@uni-jena.de

ABSTRACT

Multi-core scalability is one of the most important features for database systems running on today's hardware. Not surprisingly, the implementation of locks is paramount to achieving efficient and scalable synchronization. In this work, we identify the key database-specific requirements for lock implementations and evaluate them using both micro-benchmarks and full-fledged database workloads. The results indicate that optimistic locking has superior performance in most workloads due to its minimal overhead and latency. By complementing optimistic locking with a pessimistic shared mode lock we demonstrate that we can also process HTAP workloads efficiently. Finally, we show how lock contention can be handled gracefully without slowing down the uncontented fast path or increasing space requirements by using a lightweight *parking lot* infrastructure.

ACM Reference Format:
Jan Böttcher Viktor Leis[★] Jana Giceva Thomas Neumann Alfons Kemper . 2020. Scalable and Robust Latches for Database Systems. In *International Workshop on Data Management on New Hardware (DAMON'20), June 15, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3399666.3399908

1 INTRODUCTION

Efficient and scalable synchronization is one of the key requirements for systems that run on modern multi-core processors. Hence, there is also a variety of locking techniques to protect and synchronize data structure access, e.g., mutexes, optimistic locks, rw locks, etc. However, while it has been shown that every lock[1] type has its own area of application [4], to the best of our knowledge there has been no work that analyzes which one is best suited for high-performance database systems. This is a non-trivial problem since a DBMS must support a broad range of workloads: from write-heavy transactions to read-only analytics, and even hybrid workloads.

When designing our new database system Umbra [27], we started investigating different locking techniques in search for an optimal lock. We quickly discovered that it is not possible to find a lock that performs best across *all* workloads and on all machines. However, we noticed that there are some re-occurring best practices for locking and synchronization. Therefore, we first summarize the database specific demands on locking and then address them by analyzing and evaluating different locking techniques accordingly.

Locking Modes (§2)			**Space** (§4.3)
Optimistic	Pessimistic	Hybrid Locking	
Contention Handling Strategies (§3)			
Busy-Waiting		Kernel-Supported	
Spinning, Local, Ticket, Backoff		*Mutex, Futex, ParkingLot*	

Figure 1: Locking dimensions – *and sections in paper*

Which features and functionality should a "database-friendly" lock have? In general, most database workloads, even OLTP transactions, mostly read data, and thus reading should be fast and scalable. This includes table scans but also indexes like B-Trees or tries. For indexes, efficient synchronization is challenging as every lookup traverses the same root and upper levels have high traffic. Such read patterns or repetitively scanning small tables lead to hotspot areas in databases which should be lockable with **minimal overhead** as they are accessed so frequently.

Many modern in-memory database systems compile queries to efficient machine code to keep the **latency** as low as possible [26]. A lock should therefore integrate well with query compilation and avoid external function calls. This requirement makes pure OS-based locks unattractive for frequent usage during query execution.

To protect fine-granular data like index nodes, or hash table buckets, the lock itself should be **space efficient**. This does not necessarily mean minimal, but it should also not waste unreasonable amount of space. For instance, a `std::mutex` (40-80 bytes) would almost double the size required for an ART node [17].

Last but not least, another important aspect is efficient **contention handling**. While we assume that heavy contention is usually rare in a well-designed DBMS, some workloads make it unavoidable. The lock should, thus, handle contention gracefully without sacrificing its fast, uncondented path. While this is a goal for most production systems, during query execution we may have some additional demands. Imagine, for example, that the user wants to cancel a long-running query, but the working thread is currently sleeping while waiting for a lock. Waiting too long can lead to an unpleasant user experience. For this reason, it would be desirable if the lock's API would allow one to incorporate periodic cancellation checks while a thread is waiting.

In this paper, we show how we have addressed all the demands identified above, across the different dimensions of locking shown in Figure 1. More specifically, after discussing the advantages and shortcomings of optimistic and pessimistic locking modes, we present the design of a new hybrid lock that combines both modes to serve the various demands of versatile database workloads. Furthermore, after summarizing different contention handling strategies, we show how we avoid busy-waiting in Umbra by using the

[1]In this paper, we always use the term "lock" instead of "latch" since we focus on low-level data structure synchronization, not high-level concurrency control.

DaMoN'20, June 15, 2020, Portland, OR, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8024-9/20/06.
https://doi.org/10.1145/3399666.3399908

lightweight ParkingLot mechanism. Finally, we validate our proposed solution by comparing it to other standard locking techniques across a variety of factors and evaluating their performance with both micro-benchmarks and full-fledged database workloads.

2 LOCKING TECHNIQUES

For databases we need locks with minimal overhead and maximal scalability. Therefore, this section mostly focuses on optimistic locking as recent work shows that it has superior performance and advantages compared to pessimistic or lock-free designs [12, 20, 32]. Nevertheless, in write-heavy scenarios there is also a raison d'être for pessimistic locking. In Section 2.3, we show how both approaches can be combined into a single hybrid lock to handle all database workloads efficiently.

2.1 Optimistic Locking

The basic idea of optimistic locking is to validate that the data read in a critical section has not changed in the meantime, i.e., one has read consistent data. Therefore, the lock keeps a version that is incremented by every writer when releasing the lock. To validate that a reader has read consistent data, it must check that the version has not changed during its read. If the version has changed or if the lock bit (also encoded in the version field) is set, the reader must restart its read operation. Restarting can either be handled by the application, or transparently by the lock itself using a lambda-API as shown in Algorithm 1.

Optimistic locking avoids atomic writes and its cache line stays in shared mode. Pessimistic locks must always modify the cache line and thus their performance is bound by cache-coherency latencies [4].

Optimistic locking is particularly beneficial for frequently read data as it avoids the expensive atomic writes required by pessimistic lock acquisitions. Typical read hot-spots are certain shared tables, tuples, and index structures. In tree-like index structures, the topmost nodes are highly contended as every lookup or update must traverse them. Every index access would, thus, create unnecessary cache line bouncing on the nodes if they are locked pessimistically. With optimistic locking, cache invalidations are only needed when a node is updated, which in most cases will only happen on the lower, less frequented levels of the tree. Prior work shows how effective optimistic lock coupling is compared to traditional pessimistic locking, or even complex lock-free implementations [16, 32].

However, there are also some downsides and limitations to optimistic locking. First, optimistic locking can fail if there is a concurrent writer, and thus you can only use it, when it is safe to "fail" and to restart the read operation. This usually holds true for reading contiguous memory like tuples in tables, but can require some additional precautions when accessing index nodes or MVCC version chains, which might have been deleted or garbage collected [2]. For ART, we use an epoch guard to keep the memory of deleted nodes alive, until it is safe that no optimistic reader can access them anymore, i.e., every thread has advanced to the next epoch [20]. The deleted nodes are marked with a special obsolete bit to notify the reader of its deletion upon version validation.

Listing 1: Optimistic Locking

```
void readOptimistically(Lambda& readCallback)} {
  // Attempt to read optimistically
  for (i in [1 : MAX−ATTEMPTS]) {
    preVersion = getVersion();
    if (isLocked(preVersion))
      continue;
    readCallback();
    postVersion = getVersion();
    if (preVersion == postVersion)
      return;
  }
  // Fallback to pessimistic locking
  lockPessimistic();
  readCallback();
  unlock();
}
```

Another challenge of optimistic reading is that all operations must be restartable without any side effects. The user of the optimistic lock must be aware of this and implement some sort of restart logic. For instance, in a DBMS, this usually means that one must buffer the optimistically read tuples and only push them into query pipelines after a successful validation. Otherwise, the same tuples could be pushed again into the pipelines during a restart.

Further, when there is too much write contention, optimistic locking can also suffer from starvation. For this reason, one must include a fallback to pessimistic locking as shown in Algorithm **??**. If the lock does not support a shared mode, this means that the reader has to acquire the lock exclusively which limits its concurrency unnecessarily. For this reason, we propose the use of a hybrid lock which can fall back to shared locking in Section 2.3.

Another technique that guarantees fast, successful reads without any restarts is *Read-Optimized Write EXclusion (ROWEX)* [1, 20]. In contrast to optimistic locking, readers do not require any synchronization, not even version checking, while the writers must guarantee that all reads are consistent. In contrast to optimistic locking, ROWEX is a more involved synchronization technique, that can require major changes to the used algorithm or data structures as all writes now have to appear atomic to the readers [20].

2.2 Speculative Locking (HTM)

A special form of optimistic locking is Intel's hardware-supported speculative locking [18, 19, 21]. Speculative locking allows multiple threads to hold the same lock as long as their operations do not conflict [8]. In contrast to pure version-based optimistic locking, this also allows for non-conflicting concurrent writers within the same critical section. All conflicts are detected on L1-cache line granularity (usually 64 bytes) and the addresses of the joint read/write-set must fit into L1 cache. Additionally, the critical section should be short to avoid interrupts or context switches and must avoid certain system calls [9]. A major downside of hardware-based locking is the hardware itself. Only modern Intel and ARM processors support this or a similar feature [22]; other manufactures and older or low-end processors cannot use it at the moment. Thus, when using it, the system always needs a fallback to a traditional lock to

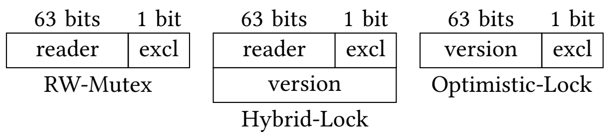


Figure 2: Hybrid-Lock – *Combining optimistic and pessimistic*

handle aborts (e.g., conflicts, read/write-set too big, etc.) or missing hardware support.

2.3 Hybrid Locking

While optimistic locking works best for read-only and low contention cases, it can easily suffer from frequent restarts or even starvation in mixed workloads. Alternatively, pessimistic modes like *exclusive* or *shared*, which guarantee that the execution of a critical section succeeds, can be used. However, unlike optimistic locking, they also add an overhead as every lock operation requires at least one atomic write. Table 1 summarizes their use cases based on their strengths and weaknesses and shows that shared-locking is a more robust solution in mixed workloads.

These insights are especially useful for database systems with diverse workloads. However, to use these findings, we must be able to lock the same data differently depending on the current context. For instance, when we access the pages (e.g., B-Tree nodes) in a buffer manager, we want to traverse the read-contended top-level nodes with minimal overhead, i.e., optimistically, but when we scan an entire leaf page, we prefer to do this pessimistically to avoid the risk of expensive restarts. So, the same node lock must support both optimistic and pessimistic locking.

For this reason, we have designed a Hybrid-Lock that extends a pessimistic RW-Mutex with support for optimistic locking. In theory, this would be possible by combining all fields of both locks into a single 64-bit word. However, for a more efficient and robust implementation, we decided to keep the version in a separate 64-bit field as shown in Figure 2.

Separating the lock from the version also allows one to reuse arbitrary, existing read-write lock implementations without changing their code as shown in Listing 2. Unlocking requires some precautions: We must increment the version before we release the lock to avoid races in the optimistic validation phase. On Intel platforms one could also use a `CMPXCHG16B` instruction to update the version and release the lock at the same time, but one must never release the lock before incrementing the version. Otherwise, the optimistic reader could miss an exclusive writer during its validation.

Reading optimistically still works like in Algorithm **??**, with the small but decisive difference that we can now fall back to shared instead of exclusive locking when the optimistic validation fails. This makes the lock very versatile and is the reason we use it throughout our database systems[2]. For graceful contention handling, we back it with a ParkingLot as described in Section 3.4. The additional bit required to indicate parking threads is encoded into the RW-Mutex and—in combination with the exclusive bits—also serves the purpose to indicate pending writers to the readers.

[2]We replaced the "Versioned Latches" described in earlier work [15, 27].

Listing 2: Hybrid Locking

```
class HybridLock {
  RWMutex rwLock;
  std::atomic<uint64_t> version;

  public:
  // Simply call rwLock
  void lockShared() { rwLock.lockShared(); }
  void unlockShared() { rwLock.unlockShared(); }
  void lockExclusive() { rwLock.lockExclusive(); }

  // Always increment the version before unlocking to avoid races!
  void unlockExclusive() { ++version; rwLock.unlockExclusive(); }

  bool tryReadOptimistically(Lambda& readCallback) {
    if (rwLock.isLockedExclusive())
      return false;
    auto preVersion = version.load();
    // Execute read callback
    readCallback();

    // Was locked meanwhile?
    if (rwLock.isLockedExclusive())
      return false;
   // Version still the same?
    return preVersion == version.load();
  }

  void readOptimisticIfPossible(Lambda& readCallback) {
    if (!tryReadOptimistically(readCallback)) {
      // Fall back to pessimistic locking
      lockShared();
      readCallback();
      unlockShared();
    }
  }
};
```

3 CONTENTION HANDLING

Actual lock contention must be rare in a database system designed for scalability. However, some workloads make it unavoidable and when it occurs, we want to handle it gracefully without slowing down the fast path. Here we present different contention handling strategies and discuss their advantages and pitfalls.

3.1 Busy-Waiting/Spinning

A common approach is to busy-wait, or "spin" until a lock is free again. While this approach itself sounds straightforward, there exist several variations of spinning and, especially without precautions, it has several pitfalls. For instance, spinning can lead to **priority inversion**, as spinning threads seem very busy to a scheduler they might receive higher priority than a thread that does useful work. Especially in the case of over-subscription, this can cause critical problems. Additionally, heavy spinning **wastes resources and energy** [6] and increases **cache pollution**, which is caused by additional bus traffic. Following the MESI-protocol, every atomic write needs to invalidate all existing copies in other cores. Ideally,

Table 1: Qualitative overview – *Which locking mode is* $\boxed{\textit{best}}$ *for a certain workload?*

Workload Type	Exclusive	Shared	Optimistic
Read-Only	Too restrictive	"Read-Read Contention"	$\boxed{\text{No Overhead}}$
Read-Mostly: cheap reads	Too restrictive	Still some contention	$\boxed{\text{Restarts unlikely and cheap}}$
Read-Mostly: big read set	Too restrictive	$\boxed{\text{Lock overhead diminishes}}$	Restarts can be expensive
Write-Heavy	Restrictive	$\boxed{\text{Good}}$	Many Aborts/Starvation
Write-Only	$\boxed{\text{Equally good (all writes are locked exclusively)}}$		

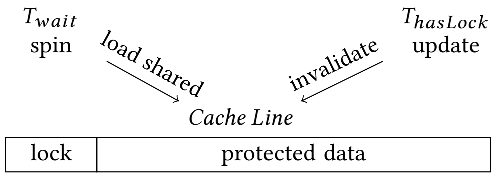


Figure 3: False-sharing – *Spinning can cause false-sharing with the writing thread and unnecessary bus traffic due to invalidations*

a core owns a cache line exclusively and does not need to send any invalidation messages. However, if other threads are spinning on the same lock, they constantly request this cache line, causing contention. The negative effects are worst when the waiting thread does write-for-ownership cycles, as those cause expensive invalidation messages [30]. For this reason, a waiting thread should use the `test-test-and-set` pattern and only do the write-for-ownership cycle when it sees that the lock is available. In other words, it only reads the lock state in the busy loop to keep the lock's cache line in shared mode.

However, even with the `test-test-and-set` pattern, spinning can still lead to cache pollution when the protected data is on the same cache line as the lock itself (cf. Figure 3). By spinning on the lock the waiting thread T_{wait} constantly requests the cache line in shared mode. Whenever the lock owning $T_{hasLock}$ updates the protected data, it must invalidate T_{wait}'s copy of the cache line. Having to send these invalidation messages, slows down $T_{hasLock}$ and increases the time spent in the critical section.

To limit the described problems, there exist several backoff strategies that add pause instructions to put the CPU into a lower power state, or call `sched_yield` to encourage the scheduler to switch to another thread. However, since the scheduler cannot guess when the thread wants to proceed, yielding is generally not recommended as its behavior is largely unpredictable [31].

3.2 Local Spinning using Queuing

The performance degradation of cache contention due to spinning becomes worse with an increasing number of cores or NUMA sockets [4, 30]. To overcome the problem of cache line bouncing, some spinlock implementations spin only on thread-local copies of the lock. Examples are the MCS-lock or a read-write mutex adaptation by Krieger et al. [13, 25]. When acquiring a lock, every thread creates a thread-local instance of the lock structure including its lock

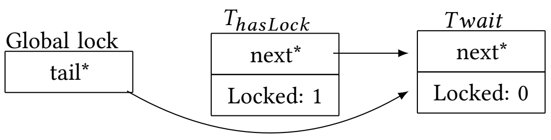


Figure 4: Queuing lock – *Threads spin on local copies only.*

state and a `next` pointer to build a linked list of waiting threads.[3] Then, it exchanges the next pointer of the global lock, making it point to its own local instance. If the previous `next` entry was `nil`, the lock acquisition was successful. Otherwise, if the entry already pointed to another instance, the thread enqueues itself in the waiting list by updating the `next`-pointer of the found instance (current `tail`) to itself. Figure 4 sketches the system's state when $T_{hasLock}$ is holding the lock and T_{wait} is waiting. While waiting, every thread spins on its own local `Locked` flag, until its predecessor releases the lock and updates the lock state. Besides reducing the amount of cache line bouncing, this queuing procedure also preserves the order of threads and, thus, guarantees fairness.

3.3 Ticket Spinlock

A ticket spinlock is another variant of spinlocks, which guarantees fairness without using queues. It does so by maintaining two counters: `next-ticket` and `now-serving`. A thread gets a ticket using an atomic `fetch_and_add` and waits until its ticket number matches that of now-serving. Besides giving fairness, this also enables more precise backoff in case of contention by estimating the wait time. The wait time can be estimated by multiplying the position in the queue and the expected time spent in the critical section. Mellor-Crummey and Scott argue that it is best to use the minimal possible time for the critical section, as overshooting in backoff will delay all other threads in line due to the FIFO nature [25].

3.4 Kernel-Supported ParkingLot

While some of the discussed busy-waiting strategies can reduce unnecessary cache contention or guarantee fairness, there is still no suitable solution for over-subscription or waste of energy. For this reason, many locks build on kernel-level locking, such as pthread mutexes, to suspend a thread until the lock becomes available again. As these system calls have a significant overhead, adaptive locks

[3]Some rw-mutex implementations also use a doubly-linked list, as readers should be able to release the lock in arbitrary order.

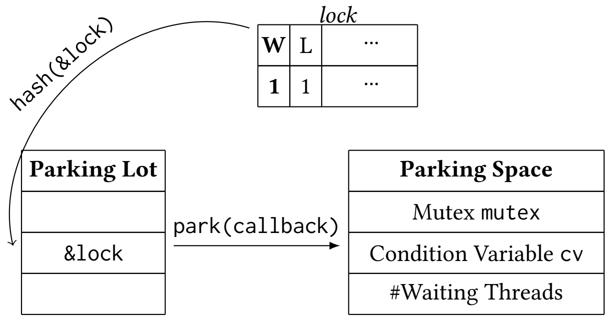


Figure 5: Parking Lot – *All waiting threads park themselves in the Parking Lot (global hash table) until the callback-condition is fulfilled. The suitable Parking Space is determined using the lock's address.*

like Linux' futexes (fast user-space mutexes) only block using the kernel when there is contention [23].

Building on the idea of futexes, WebKit proposed a more versatile form of adaptive locking called the Parking Lot [29]. A Parking Lot is a global hash table that maps arbitrary locks to wait queues using their addresses as keys. Unlike Linux' futexes, this design is portable and does not rely on non-standard, platform-specific system calls. It also allows additional functionality like passing a callback function that is invoked while "parking". In Umbra, we use this to integrate additional logic like checking for query cancellation, or in the buffer manager to ensure that the page we are currently waiting for has not been evicted in the meantime.

Figure 5 sketches our implementation of a ParkingLot. In the uncontented case, nothing changes and a thread acquires the lock as usual by setting the lock bit (L). However, when another thread tries to get the same lock, it will now wait in the parking lot. Therefore, it first brings the lock in a "someone-is-waiting" state by setting the wait bit (W). Then, it uses the lock's address to find a parking space in the global parking lot. If the user-defined waiting condition is still fulfilled, the thread starts waiting on the condition variable. When the first thread releases the lock, it sees that someone is waiting because the wait-bit was set. It looks up the parking space in the Parking Lot and wakes all parking thread(s). To avoid races during these parking operations, every parking space is guarded by a separate mutex.

The parking lot itself is implemented as a fixed-sized global hash table with 512 slots. More spaces are not necessary as the maximum number of contended locks is always smaller than the number of threads. For the unlikely case of hash collisions, we use chaining. When we park a thread that waits for a lock during query execution, we wake it up sporadically (every 10 ms) to check if the query was canceled meanwhile.

Listing 3 shows the pseudo code of our `park()` implementation. Note that while the implementation of the ParkingLot itself is fairly straightforward, the locks using it require a careful design. One must ensure that the information that a thread is parked is never lost; otherwise threads might remain in the parking state forever. So every operation that changes the state of the lock must respect the wait-bit, and wake waiting threads if necessary.

Listing 3: Parking Lot Implementation

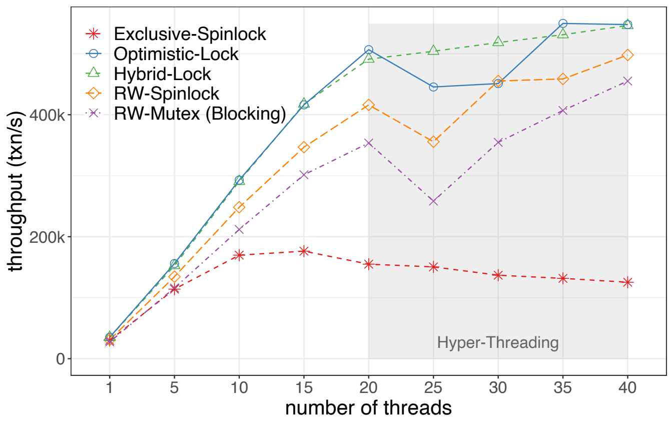
```
void park(void* lockAddr, Cb& callback, unsigned timeoutInMs)
// Park a thread until the callback's condition becomes true
{
 ParkingSpace& parkingSpace = getParkingSpace(lockAddr);
 // Lock the parking space
 parkingSpace.mutex.lock()
 ++parkingSpace.waiting;

 // Go to sleep after confirming that we still have to block (callback())
 if (!timeoutInMs) {
   while (!callback())
     parkingSpace.cv.wait(lock);
 } else {
   // Sporadically call the callback, e.g., to check for query cancellation
   while (!callback()) {
     parkingSpace.cv.waitWithTimeout(lock, timeoutInMs);
   }
 }

 // Leave the parking space
 −−parkingSpace.waiting;
 parkingSpace.mutex.unlock()
}
```

4 EVALUATION

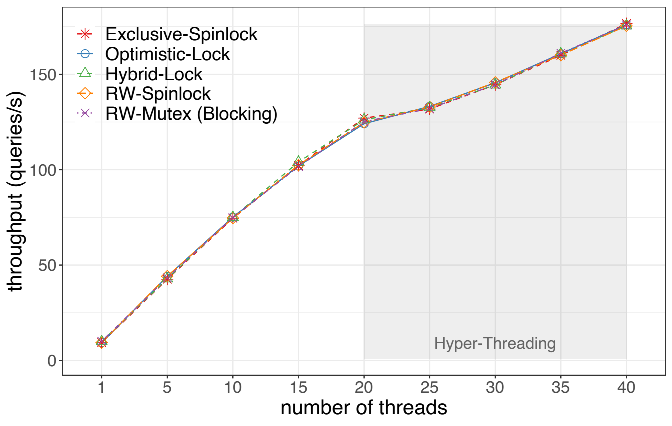
We evaluate the different locking approaches on an Ubuntu 18.04 machine with two Intel Xeon E5-2660 v2 CPUs running at 2.20 GHz with 10 physical cores (20 HT) each and a total of 256 GB DDR3. The sockets communicate using a high-speed QPI interconnect (16 GB/s). During all experiments we do not pin any threads to cores or NUMA sockets to allow the scheduler to distribute them freely. Table 2 lists all evaluated locking approaches and their concrete implementations.

4.1 TPC-C and TPC-H

We run TPC-C and TPC-H with lock representatives of the different locking modes (optimistic, shared, exclusive, and hybrid) and also a kernel-based RW-Mutex (`std::shared_mutex`). For the experiments, we replaced all locks in our DBMS HyPer that are relevant for query execution (table/tuple locks and the ART node locks) [11]. As HyPer is an in-memory DBMS, the results are not affected by any interrupts caused by IO. To see the effects of contention and cross-partition transactions, we do not pin any threads to warehouses in TPC-C. Figure 6 shows that the Optimistic and Hybrid-Locks dominate the throughput performance in TPC-C. This is mostly because their non-optimistic counterparts experience increasing "read-read contention" on the top-level nodes of the indexes. These read-read contention effects in the indexes were also very visible for the TPC-H benchmark. Only after disabling all index scans in TPC-H did the curves start to converge completely. This is because the lock acquisitions during table scans are more evenly distributed which reduces cache line contention. Also, scanning a chunk of tuples (1024 in our system [12]) amortizes the cost of acquiring a lock. Based on these findings, we always run the Hybrid-Lock in pessimistic mode when scanning bigger chunks of data as using



(a) TPC-C – *Increasing the number of threads (100 warehouses)*



(b) TPC-H – *Increasing the number of threads (sf-1, no indexes)*

Figure 6: Full database benchmarks – *Impact of lock choice*

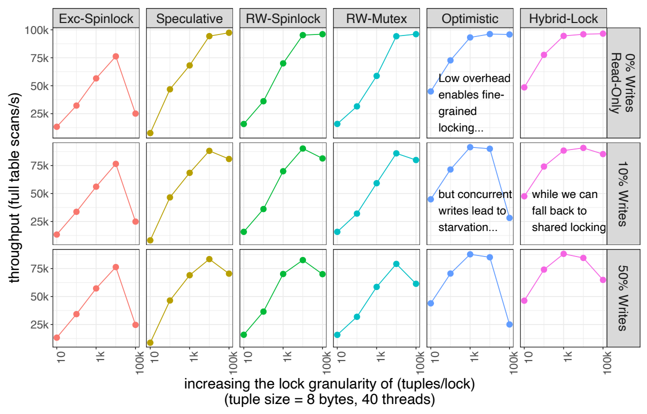


Figure 7: Granularity – *Exploring the sweet spot between lock granularity and overhead for mixed scans and updates*

Table 2: Space consumption – *in bytes, using Linux, C++17*

Lock	Implementation	Size
Optimistic-Lock	`version (Fig. 2)`	8
RW-Speculative	`tbb::speculative_spin_rw_mutex`	192
RW-Spinlock	`tbb::spin_rw_mutex`	8
RW-Local-Spinning	`tbb::queuing_rw_mutex`	8
RW-Mutex	`std::shared_mutex`	56
Exclusive-Spinlock	`atomic-flag`	1-8
TicketSpinLock	`next-ticket + now-serving`	8-128
OS-Mutex	`std::mutex`	40
Hybrid-Lock	`RW-Mutex + version (Fig. 2)`	16

optimistic mode hardly brings any benefit to justify the risk of an expensive restart.

4.2 Lock Granularity

The granularity, i.e., the number of tuples protected per lock, can have a big impact on the system's performance. For point accesses like updates, or key lookups, the granularity determines the maximum number of concurrent accesses. Thus, write-heavy workloads can benefit from fine-grained locking. However, during a full table scan, every additional lock increases the required number of lock acquisitions and reduces its effective memory bandwidth. In this experiment, we want to find the fine line between high concurrency and low overhead. Figure 7 shows that the Optimistic and Hybrid-Lock reach their sweet spots at a granularity of 1000 tuples, while the pessimistic locks need 10× more tuples to amortize the costs for lock acquisitions. While this does affect the peak performance in the read-only case, the more fine-grained locking starts to pay off when the number of writes increases. With 10% or more writes, the Optimistic and Hybrid-Locks outperform all other configurations while keeping their granularity at 1000 tuples.

4.3 Space Consumption

The space consumption of locks is important to support fine-grained concurrency. The smaller the area of protected data is, the more significant the lock's size is. Especially for cache line optimized index structures like ART [17], a lock should not extend the required space per node significantly. In general, user-space locks are more space efficient as they only use 1-2 atomic values. However, some techniques like speculative locking, or some TicketSpinLock implementations require additional padding to avoid false-sharing between cache lines. The locks relying on the OS generally also require more memory, as they are often implemented as a combination of condition variables and locks. Their sizes can also vary depending on the underlying OS and library. The exact sizes for the lock implementations used throughout this paper are listed in Table 2. For the Exclusive-Spinlock, we use a 64-bit atomic, as we saw 2-3× better throughput in micro-benchmarks compared to a single byte implementation.

Table 3: Performance Counters – *with and w/o contention*

Read-Only	1 thread			40 threads			
	cyc.	instr.	IPC	cyc.	instr.	IPC	L1-m
RW-Speculative	76	117	1.55	5135	119	0.02	1.3
RW-Local-Spinning	141	126	0.90	77,584	26,864	0.35	91.0
RW-Spinlock	57	57	1.00	4,531	59	0.01	1.2
RW-Mutex	81	108	1.34	9,583	118	0.01	3.0
Optimistic	8	30	3.77	12	30	2.58	0.0
Hybrid-Lock	11	35	3.05	17	35	1.85	0.0
+ParkingLot	11	35	3.07	17	35	2.06	0.0
Write-Only	**1 thread**			**40 threads**			
	cyc.	instr.	IPC	cyc.	instr.	IPC	L1-m
RW-Speculative	74	101	1.36	67,918	12,413	0.19	48.9
RW-Local-Spinning	70	86	1.23	79,928	28,058	0.35	84.0
RW-Spinlock	60	42	0.70	52,215	9,656	0.18	38.1
RW-Mutex	95	98	1.03	29,201	5,631	0.19	27.6
Optimistic	97	20	0.21	8,636	1,663	0.19	23.7
Hybrid-Lock	69	53	0.77	4,082	1,229	0.30	12.8
+ParkingLot	82	64	0.78	421	150	0.35	2.1

4.4 Efficiency of Lock Acquisition

In this experiment, we analyze the efficiency and micro-architectural properties of lock acquisitions. Table 3 shows the normalized cycles (cyc.), instructions (instr.), instructions per cycle (IPC), and L1-misses (L1-m) per lock acquisitions. An efficient lock keeps the number of instructions low, while maintaining a high throughput. When the code allows optimal branch predictions and caching, modern CPUs can issue up to four instructions per cycle [10]. Optimistic locking has such near-optimal IPC in the read-only case as it can keep the cache line with the version shared between all threads. In contrast, all pessimistic locks have a significant worse IPC, as their instructions are stalled by atomic writes at the start and end of every critical section. In the contended case, these operations become very expensive due to L1-cache contention and CAS-operations have to be repeated multiple times consuming many cycles.

For uncondented exclusive locking, all locks show about the same performance in terms of cycles although their number of instructions varies. For instance, the Local-Spinning lock uses more instructions as it creates a local copy for every lock acquisition, whereas normal Spinlocks only set a lock bit. When many threads are competing for the same lock, the consumed cycles and L1-misses go up. Whereas, the ParkingLot mechanism significantly helps to keep these effects minimal and the cache contention under control.

4.5 Contention Handling Strategies

Finally, although we believe that lock contention should be rare in a database system, it is sometimes unavoidable and requires a robust solution. Ideally, a contended exclusive lock gives the same throughput as serial execution. For this reason, we test and compare common contention handling strategies by "smashing" the same lock with increasing number of threads. We compare the performance of different spinlocks (traditional, ticket-based, and local

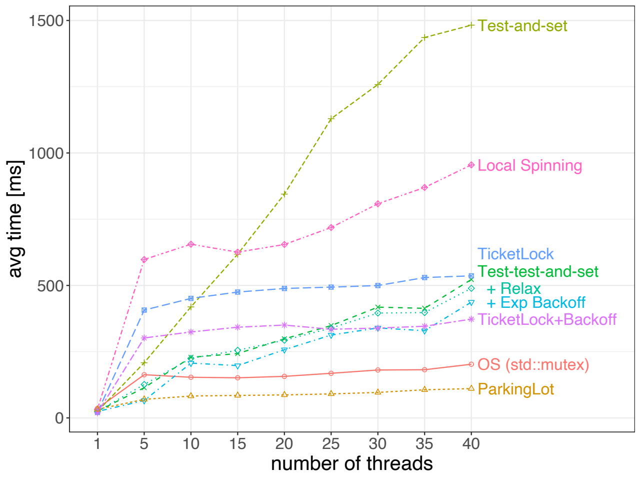


Figure 8: Contention Handling – *Measuring the latency when locking the same lock exclusively with an increasing number of threads*

spinning) to a kernel-based mutex and our hybrid ParkingLot implementation. The results, in Figure 8, show that OS-supported locks achieve the best performance. Both the `std::mutex` and our ParkingLot are hardly affected by increasing the number of threads. The lock acquisition itself is slightly more efficient in our ParkingLot implementation, which makes its baseline throughput superior to the full mutex. In contrast to the kernel-supported locks, the spinlocks suffer from increasing cache line contention. Thus, we tested several techniques to reduce this cache line contention. We achieved the biggest improvement by switching from a `test-and-set` to a `test-test-and-set` pattern. The cache pressure can be further reduced by adding additional pause/cpu_relax instructions. The best result was achieved when using them in combination with an exponential backoff based on the number of retries. For the ticket spinlock we can use a smarter backoff, as every thread can estimate its required waiting time from the its ticket number as described in Section 3.3. The bigger the difference between its ticket number and the currently active number is, the longer the wait time. Another cache contention avoidance strategy is local spinning. Here, every thread reduces the cache line contention by spinning only on its local copy of the lock. While this gives fairly stable performance, creating a local copy and inserting it into the queue adds a significant overhead to the approach.

5 RELATED WORK

There has been intensive work on locks and synchronization over the past decades. Most research was driven by the (operating) systems community and geared towards developing new or tuning existing locks [4, 30]. In the database community, different locks were mostly analyzed considering only index structures or high-level concurrency control [5, 7, 14, 20, 28, 32]. Optimistic (versioned) locks have been used to synchronize a growing number of data structures [3, 20, 24]. Falsafi et al. [6] show how the energy efficiency of software, in particular database systems, can be improved by interchanging locks. Surprisingly, no one has yet investigated the performance impacts of different locks in a holistic approach for entire database systems. In this paper, we perform database-focused experiments and combine those results with the findings of the systems community to infer best practices for locking in a DBMS.

6 CONCLUSION

In this paper, we analyzed locks based on the specific demands for databases. We showed how optimistic locking can be used to keep the overhead and latency of locking minimal. We also showed the implementation of a Hybrid-Lock, which can fall back to pessimistic locking when needed. This is particularly useful in general-purpose database systems that need to support a broad range of workloads. Through a series of experiments and evaluation criteria, we identified that ParkingLot-based contention handling works best for database systems supporting heterogeneous workloads. In the common, uncontented case they do not add any overhead and keep the size of the lock minimal. If there is contention, they handle contention gracefully by waiting in the kernel-space. Furthermore, its callback API allows one to integrate database specific logic like query cancellation checks.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286).

REFERENCES

[1] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD*. 521–534.

[2] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable Garbage Collection for In-Memory MVCC Systems. *VLDB* 13, 2 (2019). https://doi.org/10.14778/3364324.3364328

[3] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *VLDB*.

[4] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. https://doi.org/10.1145/2517349.2522714

[5] Jose M. Faleiro and Daniel J. Abadi. 2017. Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold?. In *CIDR*.

[6] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. 2016. Unlocking Energy. In *USENIX ATC 16*. USENIX Association, Denver, CO. https://www.usenix.org/conference/atc16/technical-sessions/presentation/falsafi

[7] Goetz Graefe. 2010. A survey of B-tree locking techniques. *ACM Trans. Database Syst.* 35, 3 (2010). https://doi.org/10.1145/1806907.1806908

[8] Intel. 2019. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.

[9] Intel. 2019. Speculative locking (Transactional Lock Elision). https://software.intel.com/en-us/node/506266.

[10] Intel. 2020. *Intel® VTune^{TM} Profiler User Guide*. Chapter CPU Metrics Reference.

[11] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System based on Virtual Memory Snapshots. In *ICDE*.

[12] Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. 2019. Scalable Analytics on Fast Data. *ACM TODS* 44, 1, Article 1 (Jan. 2019). https://doi.org/10.1145/3283811

[13] Orran Krieger, Michael Stumm, Ronald C. Unrau, and Jonathan Hanna. 1993. A Fair Fast Scalable Reader-Writer Lock. In *Proceedings of the 1993 International Conference on Parallel Processing, Syracuse University, NY, USA, August 16-20, 1993. Volume II: Software*. https://doi.org/10.1109/ICPP.1993.21

[14] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. 2011. High-performance concurrency control mechanisms for main-memory databases. *PVLDB* 5, 4 (2011).

[15] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. 185–196.

[16] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019). http://sites.computer.org/debull/A19mar/p73.pdf

[17] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *IEEE*. https://doi.org/10.1109/ICDE.2013.6544812

[18] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2014. Exploiting hardware transactional memory in main-memory databases. In *ICDE*. 580–591.

[19] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2016. Scaling HTM-Supported Database Transactions to Many Cores. *IEEE* 28, 2 (2016). https://doi.org/10.1109/TKDE.2015.2411272

[20] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*.

[21] Darko Makreshanski, Justin J. Levandoski, and Ryan Stutsman. 2015. To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-Free Indexing. *PVLDB* 8, 11 (2015).

[22] Berenice Mann. 2019. New Technologies for the Arm A-Profile Architecture. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/new-technologies-for-the-arm-a-profile-architecture.

[23] Linux Programmer's Manual. 2020. futex - fast user-space locking. http://man7.org/linux/man-pages/man2/futex.2.html.

[24] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage.. In *EuroSys*.

[25] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (1991), 21–65. https://doi.org/10.1145/103727.103729

[26] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011). https://doi.org/10.14778/2002938.2002940

[27] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[28] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM. https://doi.org/10.1145/2882903.2915251

[29] Filip Pizlo. 2016. Locking in WebKit. https://webkit.org/blog/6161/locking-in-webkit/.

[30] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the Cost of Atomic Operations on Modern Architectures. In *International Conference on Parallel Architectures and Compilation, PACT*. https://doi.org/10.1109/PACT.2015.24

[31] Linus Torvalds. 2020. No nuances, just buggy code. https://www.realworldtech.com/forum/?threadid=189711&curpostid=189723.

[32] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD*.