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:

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

\(^1\)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.

database specific demands on locking and then address them by analyzing and evaluating different locking techniques accordingly.

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, uncontented 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

<table>
<thead>
<tr>
<th>Locking Modes (§2)</th>
<th>Contention Handling Strategies (§3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimistic</td>
<td>Busy-Waiting</td>
</tr>
<tr>
<td>Pessimistic</td>
<td>Kernel-Supported Spinning, Local, Ticket, Backoff</td>
</tr>
<tr>
<td>Hybrid Locking</td>
<td>Mutex, Futex, ParkingLot</td>
</tr>
</tbody>
</table>

Figure 1: Locking dimensions – and sections in paper
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 top-most 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

```java
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 manufacturers 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
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 RWMutex and—in combination with the exclusive bits—also serves the purpose to indicate pending writers to the readers.

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 best for a certain workload?

<table>
<thead>
<tr>
<th>Workload Type</th>
<th>Exclusive</th>
<th>Shared</th>
<th>Optimistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Only</td>
<td>Too restrictive</td>
<td>“Read-Read Contention”</td>
<td>No Overhead</td>
</tr>
<tr>
<td>Read-Mostly: cheap reads</td>
<td>Too restrictive</td>
<td>Still some contention</td>
<td>Restarts unlikely and cheap</td>
</tr>
<tr>
<td>Read-Mostly: big read set</td>
<td>Too restrictive</td>
<td>Lock overhead diminishes</td>
<td>Restarts can be expensive</td>
</tr>
<tr>
<td>Write-Heavy</td>
<td>Restrictive</td>
<td>Good</td>
<td>Many Aborts/Starvation</td>
</tr>
<tr>
<td>Write-Only</td>
<td>Equally good (all writes are locked exclusively)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.2 Local Spinning using Queuing

The performance degradation of cache contention due to spinning becomes worse with an increasing number of cores on 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 3: False-sharing – Spinning can cause false-sharing with the writing thread and unnecessary bus traffic due to invalidations](image)

state and a next pointer to build a linked list of waiting threads. 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

---

Some rw-mutex implementations also use a doubly-linked list, as readers should be able to release the lock in arbitrary order.
When we park a thread that waits for a lock during query execution, waiting because the wait-bit was set. It looks up the parking space with its 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 fulfilled, the thread starts waiting on the condition variable. 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 uncontended 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.

```
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();
  } else {
    // Sporadically call the callback, e.g., to check for query cancellation
    while (!callback()) {
      parkingSpace.cv.waitWithTimeout(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...
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 2: Space consumption — in bytes, using Linux, C++17

<table>
<thead>
<tr>
<th>Lock</th>
<th>Implementation</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimistic-Lock</td>
<td>version (Fig. 2)</td>
<td>8</td>
</tr>
<tr>
<td>RW-Speculative</td>
<td>tbb::speculative_spin_rwlock</td>
<td>192</td>
</tr>
<tr>
<td>RW-Spinlock</td>
<td>tbb::spin_rwlock</td>
<td>8</td>
</tr>
<tr>
<td>RW-Local-Spinning</td>
<td>tbb::queuing_rwlock</td>
<td>8</td>
</tr>
<tr>
<td>RW-Mutex</td>
<td>std::shared_mutex</td>
<td>56</td>
</tr>
<tr>
<td>Exclusive-Spinlock</td>
<td>atomic-flag</td>
<td>1-8</td>
</tr>
<tr>
<td>TicketSpinLock</td>
<td>next-ticket + now-serving</td>
<td>8-128</td>
</tr>
<tr>
<td>OS-Mutex</td>
<td>std::mutex</td>
<td>40</td>
</tr>
<tr>
<td>Hybrid-Lock</td>
<td>RW-Mutex + version (Fig. 2)</td>
<td>16</td>
</tr>
</tbody>
</table>
When the code allows optimal branch predictions and caching, whereas normal Spinlocks only set a lock bit. When many threads are competing for the same lock exclusively with an increasing number of threads, the consumed cycles and L1-misses go up. Whereas, the ParkingLot mechanism significantly helps to 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.

### 4.4 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 by 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 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 fail 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, uncontended 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