



TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM School of Computation, Information and Technology

Scalable Concurrency Control Methods for Modern Database Systems

Jan Böttcher



TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM School of Computation, Information and Technology

Scalable Concurrency Control Methods for Modern Database Systems

Jan D. Böttcher

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz: Prof. Dr.-Ing. Pramod Bhatotia

Prüfer*innen der Dissertation: 1. Prof. Alfons Kemper, Ph.D.
2. Prof. Dr. Thomas Neumann
3. Prof. Dr.-Ing. Wolfgang Lehner

Die Dissertation wurde am 09.12.2022 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 28.04.2023 angenommen.

Abstract

In the constant strive for optimal performance, database systems must be as scalable as possible to run efficiently on modern many-core machines.

This thesis addresses the most fundamental scalability choke points: The locking and synchronization primitives. First, we analyze different types of locking techniques and search for an optimal lock for database systems. While many locks are specialized for specific cases, the challenge is to find a lock that is as versatile as database workloads while providing (near) optimal performance in all scenarios. We propose a hybrid lock with pessimistic and optimistic modes to enable high read scalability. The lock sends waiting threads to a global *parking lot* that implements reasonable fairness, cache topology awareness, and robust contention handling without sacrificing fast-path performance or requiring additional in-place space.

The second half of the thesis demonstrates the integration of such locks into a modern Multi-Version Concurrency Control (MVCC) database system and focuses on the scalability of MVCC transactions in general. We discuss how to tackle logical contention that leads to serialization aborts and show how to deal with hybrid transactional/analytical processing (HTAP) in an MVCC system. Handling HTAP workloads with a mix of fast writes and long-running read transactions is an inherent challenge for MVCC. Traditional implementations struggle to keep the number of versions down as they are too coarse-grained to handle long-running queries. We propose a scalable garbage collection approach that prunes obsolete versions eagerly to enable high HTAP performance even in the presence of high update rates.

Zusammenfassung



Die Skalierbarkeit eines Datenbanksystems wird im Zeitalter von immer größer werdenden Maschinen mit vielen CPU Kernen immer ausschlaggebender für eine optimale Leistung. Diese Dissertation beschäftigt sich deswegen zunächst mit dem skalierbaren Einsatz und der Implementierung von Locks, den grundlegenden Synchronisierungsbausteinen in einem System. Dazu analysieren wir unterschiedliche Locking-Techniken auf der Suche nach dem optimalen Lock für ein Datenbanksystem. Die Herausforderung ist aus der breiten Masse an spezialisierten Locks, ein Lock zu finden, das genauso vielseitig einsetzbar ist wie ein Datenbanksystem und trotzdem in möglichst allen Fällen optimal abschneidet. Hierfür entwickeln wir ein hybrides Lock bestehend aus pessimistischen und optimistischen Sperrmodi für eine hohe Skalierbarkeit. Wenn mehrere Threads gleichzeitig auf ein Lock warten, nutzen wir einen globalen *Parkplatz*, um das Warten cache-freundlich und fair zu organisieren. Der zweite Teil der Dissertation zeigt wie derartige Locks in ein modernes Datenbanksystem integriert werden können. Des Weiteren diskutieren wir, wie logische Konflikte zwischen Schreibtransaktionen und gemischte transaktional-analytische Workloads effizient in einem Multi-Version Concurrency Control (MVCC) System bearbeitet werden können. Gerade langlaufende Anfragen stellen in der Gegenwart von Schreibtransaktionen eine Herausforderung für MVCC Systeme dar. Traditionelle Ansätze können mit der Menge an erzeugten Tupelversionen nicht effizient umgehen, da sie nicht feingranular genug arbeiten. Wir entwickeln deswegen einen skalierbaren Ansatz, der nicht mehr benötigte Versionen aggressiv direkt beim Ändern eines Tupels entfernt.

ACKNOWLEDGMENTS

I want to thank everyone who supported and accompanied me during my years at TUM, making it a fun and special time. In particular, I want to point out certain people:

- Prof. Viktor Leis: For advising me throughout all of my papers and work, and providing me with valuable tips and feedback when structuring and presenting a paper.
- Prof. Jana Giceva: Similar to Viktor, she co-authored most of my papers, gave super valuable feedback, and added completely new aspects from the system's community.
- Prof. Alfons Kemper: For supervising my thesis, letting me be part of this amazing group, and supporting me throughout the journey. He deserves special thanks for keeping my future CV clean by encouraging me to drop *garbage collection* from my thesis title.
- Prof. Thomas Neumann: For providing super valuable feedback, insights, and guidance whenever needed. And for co-supervising my thesis.
- Prof. Wolfgang Lehner: For reviewing this thesis and his support during my early Ph.D. years at the SFB project.
- All of my colleagues: For great discussions, feedback, and making the last years a fun time.
- Alice Rey, Jana Giceva, and Moritz Kroiss: For proof reading my thesis and their valuable feedback.
- My family and partner: For their ubiquitous support during those years and beyond.

Funding.

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).  

PREFACE

Excerpts of this thesis were published in advance.

Chapter 2 has previously been published in:

Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. “Scalable and robust latches for database systems”. In: *DaMoN*. ed. by Danica Porobic and Thomas Neumann. ACM, 2020

Parts of Chapter 4 have previously been published in:

Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. “Scalable Analytics on Fast Data”. In: *ACM TODS* (Jan. 2019)

Which is an extended version of

Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. “Analytics on Fast Data: Main-Memory Database Systems versus Modern Streaming Systems”. In: *EDBT*. 2017

Chapter 5 has previously been published in:

Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. “Scalable Garbage Collection for In-Memory MVCC Systems”. In: *VLDB* (2019)

CONTENTS

Acknowledgments	i
Preface	iii
1 Introduction	1
1.1 Thesis Contributions	2
1.2 Prior Publications and Authorship	3
2 Scalable and Robust Latches	5
2.1 Motivation	5
2.2 Contention Handling	7
2.2.1 Busy-Waiting/Spinning	7
2.2.2 Local Spinning using Queuing	9
2.2.3 Ticket Spinlock	9
2.2.4 Kernel-Supported Parking Lot	10
2.2.5 Hybrid Locking	11
2.3 Locking Techniques	16
2.3.1 Optimistic Locking	16
2.3.2 Speculative Locking (HTM)	17
2.4 Evaluation	19
2.4.1 TPC-C and TPC-H	19
2.4.2 Lock Granularity	20
2.4.3 Space Consumption	21
2.4.4 Efficiency of Lock Acquisition	22
2.4.5 Contention Handling Strategies	24
2.5 Related Work	24
2.6 Summary	26
3 Deeper Dive into Latches: Adding Fairness and Cache Topology Awareness	27
3.1 Motivation	27
3.2 Dimensions and Goals of Locks	28
3.2.1 Fast Path Performance	28
3.2.2 Reader Friendliness/Read Scalability	29
3.2.3 Contention Handling	29
3.2.4 Locking Algorithms	30

3.2.5	Space Consumption	31
3.2.6	Fairness	32
3.2.7	NUMA and Cache Topology Awareness	33
3.2.8	Overview	33
3.3	Hardware Analysis	34
3.3.1	Intel Xeon Gold 6212U	34
3.3.2	AMD EPYC 7713	34
3.3.3	AWS Graviton3 (Arm Neoverse v3)	37
3.4	Extending the parking lot	38
3.4.1	Adding Fairness	39
3.4.2	Adding Read-Write Lock Support	42
3.4.3	Adding Cache Topology Awareness	45
3.5	Evaluation	48
3.5.1	Fast path	48
3.5.2	Contention Handling	49
3.5.3	Comparison with ARM	50
3.5.4	Fairness	50
3.5.5	Tradeoff: Throughput vs Fairness	55
3.5.6	Topology Awareness	55
3.5.7	Summary	58
3.6	Related Work	59
3.7	Conclusion	61
4	Applying Optimistic Locking in an MVCC Database System	63
4.1	Motivation	63
4.2	Optimistic Synchronization of Tables Scans and Tuple Accesses	63
4.3	Optimistic Synchronization of Index Structures	66
4.4	Scalable Transaction Management	67
4.5	Handling Logical Contention and Data Skew	71
4.6	Performance Evaluation	72
4.6.1	Read Scalability of Optimistic Locking	72
4.6.2	Handling of Skew and Serialization Conflicts	73
4.7	Summary	75
5	Scalable Garbage Collection for In-Memory MVCC Systems	77
5.1	Motivation	77
5.2	Versioning in MVCC	80
5.2.1	Identifying Obsolete Versions	81
5.2.2	Practical Impacts of GC	82
5.3	Steam Garbage Collection	83

5.3.1	Basic Design	83
5.3.2	Scalable Synchronization	84
5.3.3	Eager Pruning of Obsolete Versions	85
5.3.4	Layout of Version Records	89
5.4	Garbage Collection Survey	91
5.5	Evaluation	97
5.5.1	Garbage Collection Over Time	99
5.5.2	TPC-C	101
5.5.3	Scalability in Mixed Workloads	103
5.5.4	Garbage Collection Frequency	103
5.5.5	Skew	105
5.5.6	Varying Read-Write Ratios	106
5.5.7	Eager Pruning of Obsolete Versions	107
5.6	Related Work	109
5.7	Summary	110
6	Conclusion	111
	Bibliography	113

LIST OF FIGURES

Figure 1	Locking dimensions – and sections	6
Figure 2	False-sharing – <i>Spinning can cause false-sharing with the writing thread and unnecessary bus traffic due to invalidations</i>	8
Figure 3	Queuing lock – <i>Threads spin on local copies only.</i>	10
Figure 4	Parking Lot – <i>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.</i>	11
Figure 5	Hybrid-Lock – <i>Combining optimistic and pessimistic locks</i>	14
Figure 6	TPC-C – <i>Increasing the number of threads (100 warehouses)</i>	19
Figure 7	TPC-H – <i>Increasing the number of threads (sf-1, no indexes)</i>	20
Figure 8	Granularity – <i>Exploring the sweet spot between lock granularity and overhead for mixed scans and updates</i>	21
Figure 9	Contention Handling – <i>Measuring the latency when locking the same lock exclusively with an increasing number of threads</i>	25
Figure 10	Intel Xeon Gold 6212U, 24 Cores – <i>Core to core latencies (with Hyper-Threads) Min=7.6ns Median=46.9ns Max=61.2ns</i>	36
Figure 11	Topology of Intel Xeon Gold 6212U – <i>All cores share the same L3 cache</i>	37
Figure 12	AMD EPYC 7713, 2x64 Cores – <i>Core to core latencies: On the first socket: Min=20.1ns Median=105.1ns Max=340.5ns Across all cores (sockets): Min=20.1ns Median=285.9ns Max=340.5ns</i>	38
Figure 13	Topology of AMD EPYC 7713 – <i>Every core complex (CCX) has its own L3</i>	39
Figure 14	AWS Graviton3, 64 Cores – <i>Core to core latencies: Min=30.4ns Median=47.2ns Max=58.4ns</i>	40

Figure 15	Eventual Fair Parking Lot – <i>When parking, every thread adds itself to the queue in the Parking Space belonging to the lock’s address. In contrast to the basic parking lot, we now manage the waiting threads in a queue and store the time when we want to become fair.</i>	44
Figure 16	EventualFairParkingLot: Read-Write – <i>We now distinguish between readers and writers. Readers can wait on the same queue element for better read concurrency. Using an additional sharedWaiter pointer, readers can directly join the shared RW-QueueElement without iterating over the list. . . .</i>	47
Figure 17	Cache Topology Awareness – <i>Instead of maintaining a single global waiting list, we can also make our lock cache topology aware by grouping threads from the same L3 cache or NUMA node. The cache topology of a system (i.e., the number of required nodes) can be determined dynamically when the parking lot is constructed.</i>	48
Figure 18	Micro Contention – <i>On Intel Xeon 6212U</i>	50
Figure 19	Micro Contention – <i>On AWS Graviton3 (ARM)</i>	51
Figure 20	Fairness of locks in different scenarios – <i>Some locks become very unfair with increasing numbers of threads or longer critical sections (Intel Xeon 6212U)</i>	52
Figure 21	Lock Acquisitions per thread – <i>Using different numbers of threads and increasingly long critical sections. With unfair and blocking locks more and more threads suffer from starvation (Intel Xeon 6212U)</i>	53
Figure 22	Finding an optimal fairness threshold – <i>When setting the fairness threshold to zero or a short time, our lock will always do a fair handover. When increasing the threshold, we allow more bargaining. Up to a certain point, this improves the throughput at the cost of fairness (score is printed next to every point). (Intel Xeon 6212U)</i>	54
Figure 23	Fairness vs. Throughput – <i>We compare different fairness settings of our lock with different fairness scores and throughputs of others. Critical section length = 100 microseconds (Intel Xeon 6212U, 48 Threads)</i>	56
Figure 24	Topology Awareness – <i>Measuring the effects of topology awareness when processing critical sections on a dual-socket machine with 16 CCXs (separate L3 Caches) (AMD Epyc 7713).</i>	57

Figure 25	Adding optimistic locking to MVCC in HyPer	64
Figure 26	Optimistic Range Scans – <i>When the validation of a node fails, the scan is restarted. To avoid frequent restarts, the maximum number of retrieved results is limited to 1024. If there are more matching tuples, the scan has to be resumed with the last matching key as the new startKey.</i>	69
Figure 27	Using thread local lists to manage and garbage collect transactions	70
Figure 28	TPC-H SF10 – <i>The scalability of the pessimistic latches suffers from read-read contention in the used index structures</i> . .	73
Figure 29	Skew handling in HyPer – <i>Throughput of skew handling techniques using 10 threads with increasing skew</i>	74
Figure 30	MVCC’s vicious cycle of garbage – <i>Old versions cannot be garbage collected as long as there are long-running transactions that have to retrieve them</i>	78
Figure 31	Practical Impacts – <i>The system’s performance drops within minutes in a mixed workload using a standard garbage collection strategy</i>	79
Figure 32	Long version chain – <i>Containing many unnecessary versions that are not GC’ed by traditional approaches</i>	81
Figure 33	Transaction lists – <i>Ordered for fast GC</i>	84
Figure 34	Thread-local design – <i>Each thread manages a subset of the transactions</i>	85
Figure 35	Prunable version chain – <i>Example for an active transaction with id 20</i>	87
Figure 36	Performance over time – <i>CH benchmark with 1 OLAP and 1 OLTP thread. (Mean values shown in italics)</i>	99
Figure 37	TPC-C – <i>Performance for increasing number of OLTP threads (100 warehouses)</i>	101
Figure 38	CH benchmark – <i>Performance for increasing number of OLAP threads using 1 OLTP thread</i>	102
Figure 39	GC Frequency – <i>Varying a) the period when the GC thread is triggered or b) the count of committed transactions before an epoch might be advanced (TPC-C, 20 OLTP threads)</i>	104
Figure 40	Cheap key-value updates – <i>Increasing the skew in key-value updates (using 20 OLTP threads)</i>	105
Figure 41	Varying read-write ratios – <i>Mixing table scans and key-value update transactions (20 threads)</i>	106

LIST OF TABLES

Table 1	Qualitative overview – Which locking mode is best for a certain workload?	13
Table 2	Space consumption – in bytes, using Linux, C++17	22
Table 3	Performance Counters – with and w/o contention	23
Table 4	Conceptual overview of locks	35
Table 5	Performance counters normalized per lock acquisition – On Intel Xeon 6212U	49
Table 6	Summary of locks and their performance	60
Table 7	Comparison with HANA’s Interval GC	88
Table 8	Data Layout of Version Records	89
Table 9	Garbage Collection Overview – Categorizing different GC implementations of main-memory database systems	92
Table 10	Configuration and Setup	98
Table 11	Effect of using EPO – CH benchmark with 1 read thread and 1 write thread running 300k transactions in total	108

1 | INTRODUCTION

Back in 2011, the in-memory database system HyPer [42] was first published and delivered unprecedented high transaction rates [43]. Its highly efficient code compilation delivered excellent query performance and transaction rates starting a new era of database systems. HyPer’s design was recently awarded ICDE’s *ten-year influential paper award*¹ and VLDB’s *test of time award*².

Now, roughly a decade after HyPer’s first publication, its compilation techniques are still highly competitive and were refined to deliver even better performance [44, 45]. However, current CPU architectures with an increasing number of cores create new challenges for database engineers. To utilize modern many-core systems effectively, a system must be as scalable as possible [13]. The first step towards scaling up HyPer was the introduction of morsel-driven parallelism, making the query processing scalable and NUMA-aware [53].

However, the transaction processing itself was still inherently single-threaded. Even with the addition of multi-version concurrency control, HyPer still relied on a global lock to ensure the strict exclusiveness of write transactions [78]. Since the single-threaded OLTP throughput was reasonably high, one did not want to add the overhead and complexity of synchronization to the underlying physical data structures. Nowadays, with recent advances in lightweight synchronization, like optimistic locking, this basic assumption has changed [59].

This thesis analyzes recent trends in efficient synchronization techniques and shows how to integrate and apply them to database systems. After synchronizing the physical data structures, we also look at other concurrency choke points and show how to eliminate them. Here, we focus on Multi-Version Concurrency Control (MVCC), the central concurrency control mechanism of modern database systems. An inherent problem of MVCC is the handling of long-running queries in the presence of write transactions. Traditional garbage collection approaches are too coarse-grained and cannot keep the number of tuple versions under control in such cases. Delayed or imprecise garbage collection leads to very long version chains if the same tuple is updated multiple

¹ http://tab.computer.org/tcde/icde_inf_paper.html

² <https://vlldb.org/2021/?vlldb-endowment-awards>

times. We investigate how the number of versions can be kept controllable by pruning versions immediately as soon as they become obsolete.

1.1 THESIS CONTRIBUTIONS

SCALABLE LATCHES FOR DATABASE SYSTEMS (CHAPTERS 2 AND 3) These chapters intend to design a lock that is as versatile as a database system. The lock should be scalable and robust when running on many cores and also reflect the current hardware trends, such as multiple L3-caches or NUMA sockets and the rise of other architectures like ARM. Therefore, we build on the success of optimistic locking and lock coupling and combine it with a *parking lot* for waiting threads. A parking lot is a small global hash table that provides robust contention handling for every lock without adding any in-place storage to the lock or slowing down its fast path in the absence of contention. We further extend the parking lot approach to implement reasonable fairness and cache topology awareness for optimal performance.

INTEGRATION INTO A MVCC DBMS (CHAPTER 4) This chapter shows the integration of optimistic locking into an MVCC DBMS. We first show how physical data structures (tables and indexes) can be synchronized optimistically. Then, we revise the central transaction management of MVCC for scalable transaction processing and show how to deal with conflicting concurrent transactions.

SCALABLE GARBAGE COLLECTION IN MVCC (CHAPTER 5) This chapter is devoted to the inherent problem of MVCC with long-running transactions in the presence of frequent updates. Traditional version garbage collectors are too coarse-grained which leads to an increasingly high number of versions and long version chains in the system. In this chapter, we propose a novel garbage collection approach that prunes obsolete versions eagerly. Its seamless integration into the transaction processing keeps the garbage collection overhead minimal and ensures good scalability.

1.2 PRIOR PUBLICATIONS AND AUTHORSHIP

Although I am the principal author of the research in this dissertation, I did all of the work in collaboration with my advisors Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. I use the first person plural in this thesis to reflect my collaborators' contributions.

2 | SCALABLE AND ROBUST LATCHES

Excerpts of this chapter have been published in [6].

2.1 MOTIVATION

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, and rw-locks. However, while it has been shown that every lock¹ type has its own area of application [13], 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 [77], 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.

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

¹ In this thesis, we use the term “lock” instead of “latch” since we focus on low-level data structure synchronization, not high-level concurrency control.

Locking Modes (§2.3)			Space (§2.4.3)
Optimistic	Pessimistic	Hybrid Locking	
Contention Handling Strategies (§2.2)			
Busy-Waiting		Kernel-Supported	
<i>Spinning, Local, Ticket, Backoff</i>		<i>Mutex, Futex, ParkingLot</i>	

Figure 1: Locking dimensions – and sections

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 [76]. 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 [58].

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 chapter, 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 work-

loads. Furthermore, after summarizing different contention handling strategies, we show how we avoid busy-waiting in Umbra by using the lightweight parking lot 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.

2.2 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.

2.2.1 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** [24] 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, 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 [93]. 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.

Algorithm 1: Spinning patterns

```

1 bool tryLock(lock) { return lock.CAS(exp_unlocked, locked); }
2 bool isLocked(lock) { return lock.load() == locked; }
3
4 // Test-and-set pattern
5 lockTAS() {
6     while (!tryLock(lock)) { cpu_relax() }
7 }
8
9 // Test-test-and-set pattern
10 lockTTAS {
11     while (isLocked(lock)) { cpu_relax() }
12     tryLock(lock) // can still fail!
13 }

```

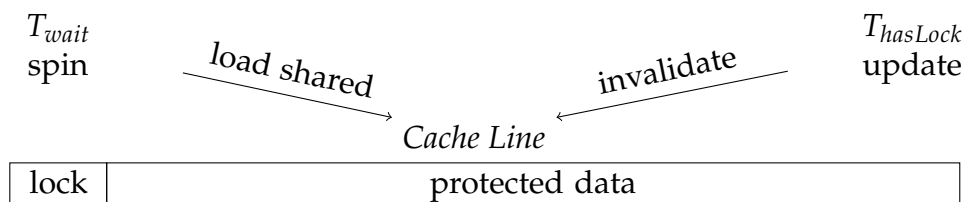


Figure 2: False-sharing – Spinning can cause false-sharing with the writing thread and unnecessary bus traffic due to invalidations

Listing 1 shows possible implementations of the *test-test-and-set pattern* and the *test-and-set pattern*.

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 2). By spinning on the lock the waiting thread T_{wait} constantly loads 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 [99].

2.2.2 Local Spinning using Queuing

The performance degradation of cache contention due to spinning becomes worse with an increasing number of cores or NUMA sockets [93, 13]. 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. [71, 49]. When acquiring a lock, every thread creates a thread-local instance of the lock structure including its lock state and a next pointer to build a linked list of waiting threads.² 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 3 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.

2.2.3 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 [71].

² Some rw-mutex implementations also use a doubly-linked list, as readers should be able to release the lock in arbitrary order.

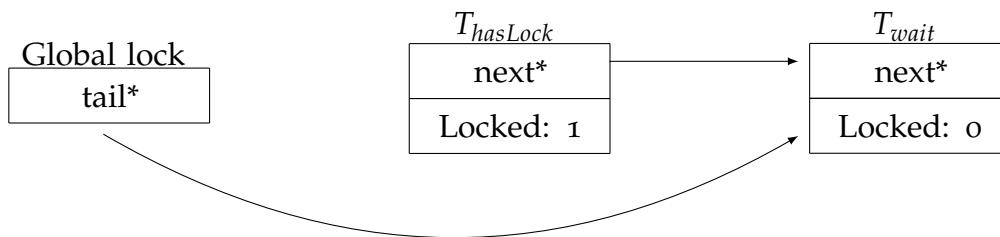


Figure 3: **Queuing lock** – *Threads spin on local copies only.*

2.2.4 Kernel-Supported Parking Lot

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 like Linux’ futexes (fast user-space mutexes) only block using the kernel when there is contention [68].

Building on the idea of futexes, WebKit proposed a more versatile form of adaptive locking that uses a parking lot for waiting threads [88]. 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 4 sketches our implementation of a parking lot. 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.

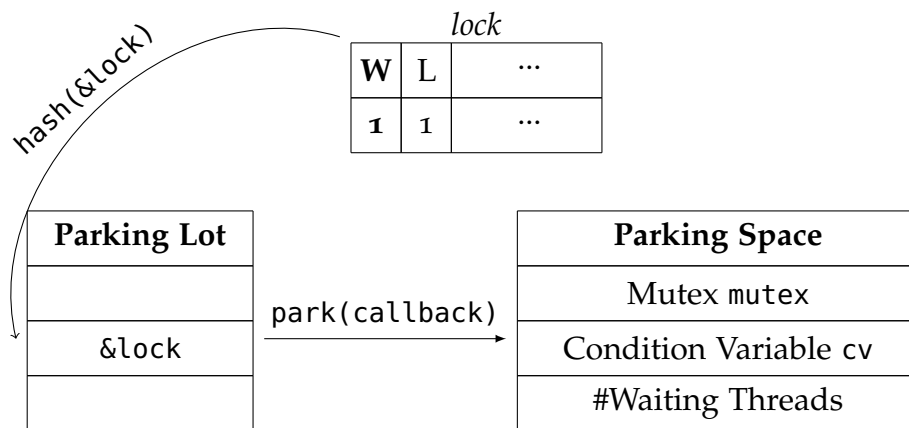


Figure 4: Parking Lot – All waiting threads park themselves in the Parking Lot (global hash table) until the callback-condition is fulfilled. The suitable Parking Space is determined using the lock’s address.

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 2 shows the pseudo code of our `park()` implementation. Note that while the implementation of the parking lot 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.

2.2.5 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.

Algorithm 2: Parking Lot Implementation

```

1 void park(void* lockAddr, Cb& callback, unsigned timeoutInMs)
2 // Park a thread until the callback's condition becomes true
3 {
4   ParkingSpace& parkingSpace = getParkingSpace(lockAddr);
5   // Lock the parking space
6   parkingSpace.mutex.lock()
7   ++parkingSpace.waiting;
8
9   // Go to sleep after confirming that we still have to block (callback())
10  if (!timeoutInMs) {
11    while (!callback())
12      parkingSpace.cv.wait();
13  } else {
14    // Sporadically call the callback, e.g., to check for query cancellation
15    while (!callback()) {
16      parkingSpace.cv.waitWithTimeout(timeoutInMs);
17    }
18  }
19
20  // Leave the parking space
21  --parkingSpace.waiting;
22  parkingSpace.mutex.unlock()
23 }

```

Table 1: Qualitative overview – Which locking mode is **best** for a certain workload?

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

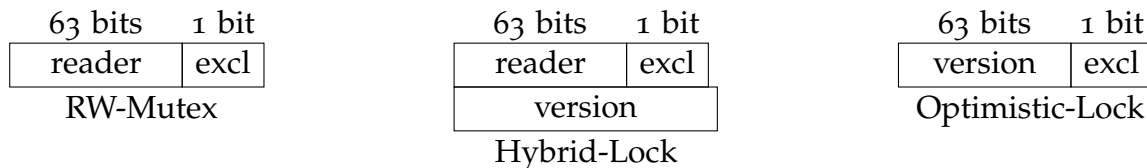


Figure 5: **Hybrid-Lock** – Combining optimistic and pessimistic locks

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-contented 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 5.

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 3. 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 Listing 4, 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 systems³. For graceful contention handling, we back it with a `ParkingLot` as described in Section 2.2.4. 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.

³ We replaced the “Versioned Latches” described in earlier work [77, 54].

Algorithm 3: Hybrid Locking

```

1 class HybridLock {
2     RWMutex rwLock;
3     std::atomic<uint64_t> version;
4
5     public:
6     // Simply call rwLock
7     void lockShared() { rwLock.lockShared(); }
8     void unlockShared() { rwLock.unlockShared(); }
9     void lockExclusive() { rwLock.lockExclusive(); }
10
11     // Always increment the version before unlocking to avoid races!
12     void unlockExclusive() { ++version; rwLock.unlockExclusive(); }
13
14     bool tryReadOptimistically(Lambda& readCallback) {
15         if (rwLock.isLockedExclusive())
16             return false;
17         auto preVersion = version.load();
18         // Execute read callback
19         readCallback();
20
21         // Was locked meanwhile?
22         if (rwLock.isLockedExclusive())
23             return false;
24         // Version still the same?
25         return preVersion == version.load();
26     }
27
28     void readOptimisticIfPossible(Lambda& readCallback) {
29         if (!tryReadOptimistically(readCallback)) {
30             // Fall back to pessimistic locking
31             lockShared();
32             readCallback();
33             unlockShared();
34         }
35     }
36 };

```

2.3 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 [103, 59, 48]. Nevertheless, in write-heavy scenarios there is also a *raison d'être* for pessimistic locking. In Section 2.2.5, we show how both approaches can be combined into a single hybrid lock to handle all database workloads efficiently.

2.3.1 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 Listing 4.

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 [13].

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 [55, 103].

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 [7]. 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 [59]. The deleted nodes are marked with a special obsolete bit to notify the reader of its deletion upon version validation.

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 Listing 4. 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 2.2.5.

Another technique that guarantees fast, successful reads without any restarts is *Read-Optimized Write EXclusion (ROWEX)* [59, 5]. 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 [59].

2.3.2 Speculative Locking (HTM)

A special form of optimistic locking is Intel’s hardware-supported speculative locking [56, 66, 57]. Speculative locking allows multiple threads to hold the same lock as long as their operations do not conflict [36]. 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

Algorithm 4: Optimistic Locking

```

1 void readOptimistically(Lambda& readCallback) {
2   // Attempt to read optimistically
3   for (i in [1 : MAX-ATTEMPTS]) {
4     preVersion = getVersion();
5     if (isLocked(preVersion))
6       continue;
7     readCallback();
8     postVersion = getVersion();
9     if (preVersion == postVersion)
10      return;
11   }
12   // Fallback to pessimistic locking
13   lockPessimistic();
14   readCallback();
15   unlock();
16 }

```

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 [38]. A major downside of hardware-based locking is the hardware itself. Only modern Intel and ARM processors support this or a similar feature [67]; 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 handle aborts (e.g., conflicts, read/write-set too big, etc.) or missing hardware support.

Intel's Threading Building Blocks (TBB) library offers implementation of speculative mutexes that already include suitable fallback mechanisms and hide the complexity of using hardware transactions. To avoid false sharing with any other data, Intel's speculative mutexes add padding to their locks which increases their sizes to 2 or in the case of a read-write mutex 3 cache lines⁴.

⁴ <https://software.intel.com/en-us/node/506270>

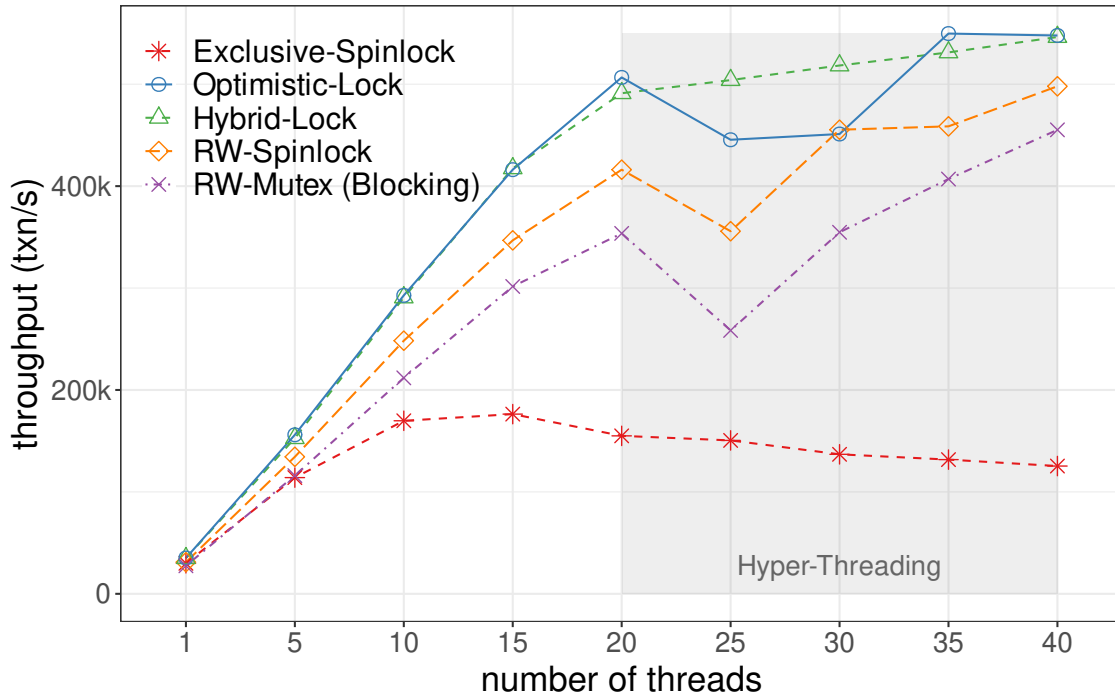


Figure 6: TPC-C – Increasing the number of threads (100 warehouses)

2.4 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.

2.4.1 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) [42]. 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 through-

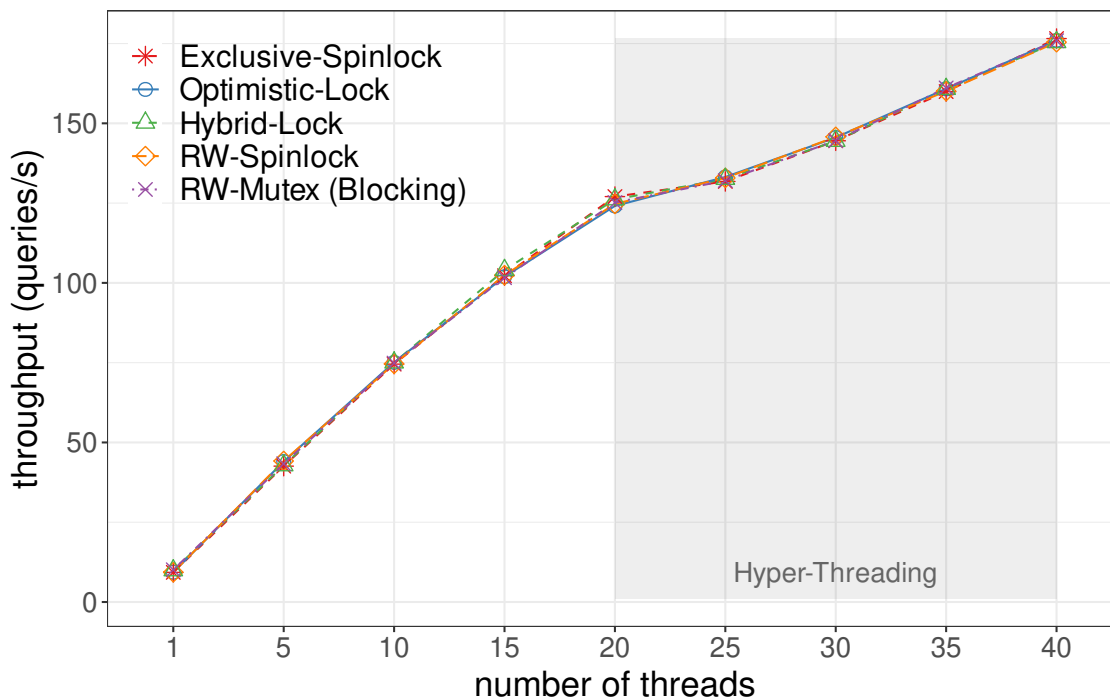


Figure 7: TPC-H – Increasing the number of threads (*sf-1*, no indexes)

put 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 [48]) 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.

2.4.2 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

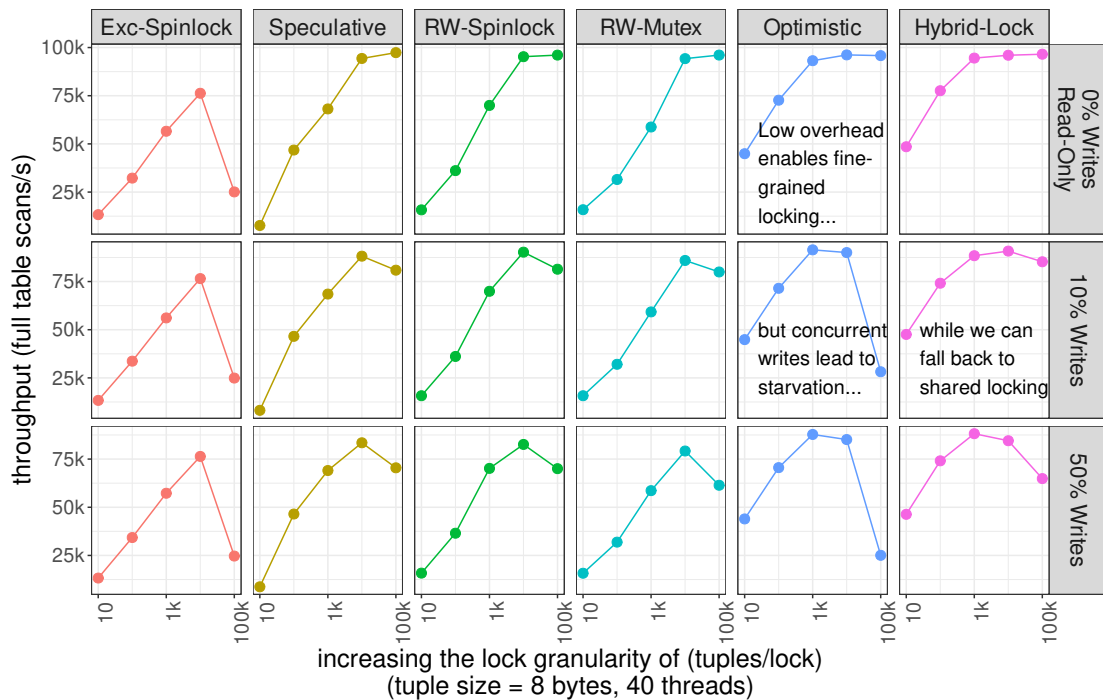


Figure 8: Granularity – Exploring the sweet spot between lock granularity and overhead for mixed scans and updates

this experiment, we want to find the fine line between high concurrency and low overhead. Figure 8 shows that the Optimistic and Hybrid-Lock reach their sweet spots at a granularity of 1000 tuples, while the pessimistic locks need $10\times$ 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.

2.4.3 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 [58], 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,

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

Lock	Implementation	Size
Optimistic-Lock	version (Fig. 5)	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. 5)	16

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 thesis are listed in Table 2. For the Exclusive-Spinlock, we use a 64-bit atomic, as we saw $2\text{-}3\times$ better throughput in micro-benchmarks compared to a single byte implementation.

2.4.4 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 [37]. 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

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

and L1-misses go up. Whereas, the ParkingLot mechanism significantly helps to keep these effects minimal and the cache contention under control.

2.4.5 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 parking lot implementation. The results, in Figure 9, 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 its ticket number as described in Section 2.2.3. 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.

2.5 RELATED WORK

There has been intensive work on locks and synchronization over the past decades. Most research was driven by the (operating) systems community

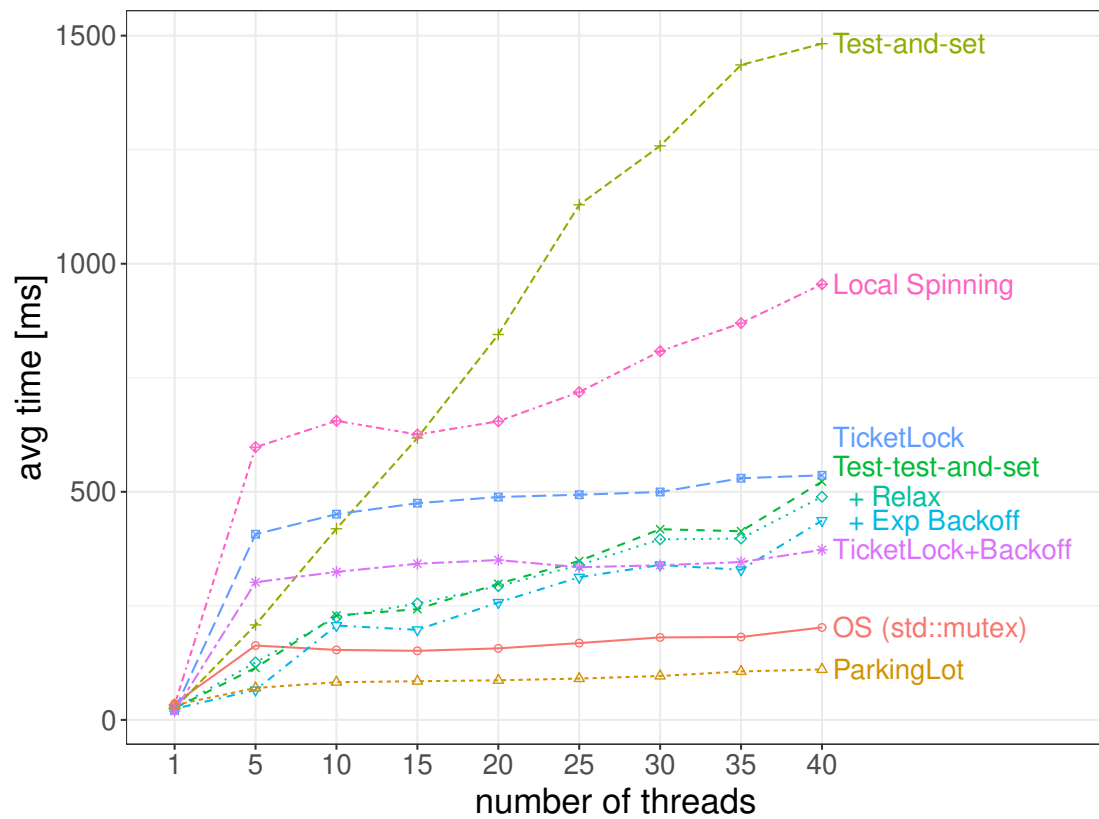


Figure 9: **Contention Handling** – Measuring the latency when locking the same lock exclusively with an increasing number of threads

and geared towards developing new or tuning existing locks [13, 93]. In the database community, different locks were mostly analyzed considering only index structures or high-level concurrency control [32, 59, 103, 22, 50, 82]. Optimistic (versioned) locks have been used to synchronize a growing number of data structures [10, 69, 59]. Falsafi et al. [24] 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.

2.6 SUMMARY

In this chapter, 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 parking lot-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.

3 | DEEPER DIVE INTO LATCHES: ADDING FAIRNESS AND CACHE TOPOLOGY AWARENESS

3.1 MOTIVATION

In the previous chapter, we have already covered the most important aspects when choosing a lock for a database system. We introduced a hybrid read-write lock that is scalable for read-read contention using optimistic locking and can still fall back to pessimistic shared locking if necessary. Further, we showed how every lock could rely on a parking lot to handle contention robustly without spinning.

However, when asking database engineers for their preferred lock, most will probably ask for more features: besides being fast and scalable, a lock should be reasonably fair and work on modern many-core machines with complex cache topologies.

Unfortunately, these properties can be somewhat conflicting in practice, making it highly non-trivial to build a general-purpose lock like this. The previous chapter already describes a reasonable solution in the form of hybrid locks that handle contention using a concept called *parking lot* for waiting threads [6]. However, in the last chapter, we did not address the aspects of fairness or cache awareness. In particular, fairness is increasingly important in production and many-core systems to guarantee reasonable tail latencies and fair progress across the system.

This chapter identifies and discusses previously untouched aspects of locks and shows the individual requirements or costs of getting a specific property, like fairness. Ideally, we end up finding an approach that gets us all the individual benefits of specialized locks without sacrificing performance in the general case. Therefore, we do not reinvent the wheel nor develop anything completely new in this chapter. We simply test and combine different locking techniques to find the optimal lock for a modern DBMS. Our goal is to make the lock decision easy by designing a lock that is on par with the best solution for (almost) every scenario. Therefore, we first recap the critical aspects of locks and show how to implement them by extending the parking lot approach

of the previous chapter. In Section 3.5, we evaluate the different designs based on the identified dimensions and goals.

This chapter discusses the following demands on locks:

- Fairness/Tail Latencies: What does it cost me to be fair? And how do I guarantee reasonable tail latencies?
- Many-Core/Socket scalability: What do I need for big machines with complex cache topologies?
- ARM vs. x86: Can I use the same lock on different architectures?
- Contention: How should I handle contention?
- Space Consumption: How much space should I use?

3.2 DIMENSIONS AND GOALS OF LOCKS

In this section, we introduce different aspects and classes of locks. We focus on the parts that are relevant when synchronizing a database system.

3.2.1 Fast Path Performance

In an ideal world, when there is no physical contention, there should hardly be an overhead when acquiring or releasing a lock.

Hardware-supported speculative locking does not require lock acquisition in the non-contending case at all [57, 38]. The hardware monitors the changes made during the critical section, and as long as they do not conflict with other concurrent changes, the lock succeeds. This only works if the hardware supports it, the critical section is short enough and unlikely to conflict, and the working set fits into the cache [70]. If speculative locking fails for one of these reasons, the system must fall back to “traditional” locking.

In traditional locking, the CPU must introduce memory barriers and at least do an atomic write when acquiring or releasing the lock. A simple implementation would be setting and resetting a lock bit with a compare-exchange instruction. More complex lock acquisitions could involve more instructions, such as inserting itself into a list or storing some additional bookkeeping information.

Nevertheless, the goal should be to keep the overhead of acquiring a lock as low as possible to encourage fine-grained locking and good scalability.

3.2.2 Reader Friendliness/Read Scalability

Previous papers show that the scalability of reads, especially in hierarchical data structures like trees, is crucial for optimal performance [59, 55, 6]. In such data structures, every insert or lookup operation traverses the root and likely the same top layers nodes. For this reason, those locks should be as scalable as possible. While in theory, read-write mutexes should not conflict when the lock is only acquired shared, the read performance does not scale linearly in practice due to cache line ping-ponging. Every time we acquire or release a node's lock we also invalidate its cache line. This also holds for shared acquisitions, as we need an atomic write to increment the reader's count, followed by another atomic write to decrement it again. Every time the lock is modified, its underlying cache line gets invalidated for the other threads. Consequently, multiple read lock acquisitions lead to cache line ping-ponging between the threads and severely hinder the performance and scalability [6]. On multi-socket machines, the CPU latency for a cache miss can easily exceed 300 ns¹.

Optimistic locking avoids this kind of read-lock cache contention. Every read-write mutex can be extended by a version field to support optimistic locking [6]. During reading the node, it is now enough to validate that the version of the lock is still unchanged after processing the critical section. The lock version is only changed when someone updates the protected data. The top layers of a tree, traversed by almost everyone and hardly changed, can be kept in the caches.

3.2.3 Contention Handling

Lock contention happens whenever multiple threads fight for the same lock. There are two main waiting strategies: *blocking* and *spinning/busy-waiting*. Blocking locks typically use the kernel infrastructure to wait for the lock. The scheduler can put the waiting thread to sleep until the lock is available again. While sleeping makes the resources of a CPU core available for other threads, this approach can also cause expensive context switches. Especially if the wait

¹ <https://github.com/nviennot/core-to-core-latency>

time is short, the cost of a context switch adds significant overhead to the lock acquisition.

For this reason, many lock implementations first or entirely rely on spinning instead of directly going to sleep. Spinning can help the throughput and the latency of lock acquisitions. However, excessive spinning on a contended resource also creates various problems, such as cache line contention, priority inversion in the scheduler, and wastes CPU resources.

For this reason, lock implementations only try to spin for a short time before blocking in kernel space or they try to limit the damage of spinning with different techniques. Those techniques involve a backoff mechanism to reduce the cache pressure, or they reduce the cache contention by spinning on a local state. Local spinning can be implemented using queue-based approaches like the MCS lock [71]: the lock itself is only a pointer to the head of a waiting list, and every thread links itself into the list and spins on its local list element only.

While all of those approaches can reduce the cache contention, spinning does still block CPU resources for other threads that could do “productive” work instead. For this reason, excessive spinning in user space can be very harmful.

3.2.4 Locking Algorithms

Regardless of the choice of blocking or spinning for contention handling, every lock implements an algorithm for deciding which thread will get the lock next.

THUNDER LOCK In a thunder lock, all threads wait in the same pool for the lock. When the lock is released, the entire pool is woken up at once and all threads race for the lock like a *thundering herd*. The threads that lose the race must enter the wait pool again.

The implementation does not need a queue: The waiting pool can be implemented using a single condition variable per lock.

BARGING Barging locks introduce a queue to order the waiting threads. In contrast to the thunder lock, the unlocker notifies only the first thread in the queue. However, there are still no guarantees that this thread will get the lock, as it must compete with newly arriving threads that have yet to enter the queue. Those threads could be quicker and steal the lock from the woken thread. Also, the thread that released the lock could reacquire it right away. Indeed, this is

likely to happen since the lock is still hot in the caches and waking up is a costly operation [26].

While *barging* is unfair and can lead to starvation, it significantly improves the throughput of a lock.

HANDOFF A handoff lock also uses a queue for the waiting threads, but in contrast to *barging*, it also ensures that the first thread in the queue gets the lock by directly handing it to the waiting thread. The lock is never physically released during this process, so no other thread can jump in and steal the lock. This mechanism makes a lock fair. The downside is that the throughput of such a lock is limited by the speed of the notified thread. Especially if the thread was sleeping and not spinning, the entire lock throughput stalls due to the cost of waking it up. Other threads that would be ready to take the lock right away are not allowed to jump in like in the previous locks.

ADAPTIVE By combining the two strategies, one can mitigate the individual problems of the approaches and build a very robust lock. The default behavior of a lock would be *barging* to enable high throughput, and we only switch temporarily to a direct *handoff* when a specific threshold time is reached. This approach would ensure eventual fairness for the lock and prevents the starvation of threads.

3.2.5 Space Consumption

The required space of a lock is essential when implementing fine-grained locking or protecting smaller parts of a data structure like tree nodes that can hardly amortize the cost of a big-sized lock. Threads relying on spinning can usually be tiny, requiring the bare minimum of a byte (or even a bit within a tagged pointer). Typical spinlocks use 8 bytes for their implementation. Using 8 bytes is enough to implement entire read-write locks and can also yield better performance as CPU instructions operating on word level are often more efficient than those on byte level.

Blocking locks often require more space for their implementations. For instance, the lock implementations of the C++ standard library require 40 bytes for a `std::mutex` or 56 bytes for a `std::shared_mutex` on Linux. They store more housekeeping information, like which thread is holding the lock.

This is significantly more memory than a spinlock and can be too much when protecting a small data structure or a small part of it. When designing

a more powerful or robust lock, a typical resort is to rely on ad-hoc allocated space. For instance, queue-based spin locks only require 1 word of lock space because the queue elements can be put onto a thread's stack. For more robust features, one can also implement something like a parking lot [6]. A parking lot is a global hash table that uses the memory address of a lock to identify its hash bucket. The bucket can have arbitrary size and functionality without adding to the lock's space consumption within the data structure. The buckets are allocated on demand when there is contention on the lock. This approach combines a spinlock's low in-place space consumption with the robust contention handling of a blocking mutex. Additionally, it can support timeouts or callbacks.

3.2.6 Fairness

Fairness of locks is a vital feature to guarantee reasonable tail latencies and to avoid the starvation of locks. Strictly speaking, fairness means that waiting threads are served in FIFO order. The threads should get the lock in the order of their arrival. In practice, there exist more relaxed definitions of fairness: A lock can also be considered as "fair" if all threads can acquire a thread equally often over a certain time [84, 2]. However, when implementing strict FIFO fairness, there exist two general approaches: Queuing and ticket locks.

QUEUING LOCK An intuitive approach to enforce FIFO execution is using a FIFO queue, e.g., a linked list. Every arriving thread either starts a new list if there is none or adds itself to the end of the list of waiting threads. The thread waits on its own (often local) instance of the list element until its predecessor hands the lock over. Figure 3 shows a basic example of a queuing lock implementation.

In general, there are different ways to implement a queue-based lock. A thread can wait on its queue element until its predecessor releases the lock. This approach requires additional operations during the unlock, as the unlocking thread must visit its successor to notify him. Thus, the list must also be explicit using next pointers [71].

Alternatively, some locks implement implicit lists by spinning on their predecessors [63]. *CLH* lock and *Hemlock* are representatives for this kind of approach [12, 18, 94]. The *Hemlock* maintains one word per thread to avoid the cost and lifetime issues with creating and destructing separate queue nodes per lock acquisitions [18].

TICKET LOCK The alternative to queuing is using a ticket lock consisting of two counters. Like a waiting room, a thread pulls a number from the first counter and waits until its number is “announced” by the second counter. Therefore, it must constantly poll the second counter creating unpleasant cache contention. Upon unlocking, a thread increments the second counter to pass the lock to the next waiting thread.

3.2.7 NUMA and Cache Topology Awareness

NUMA systems or modern CPUs create new challenges for the design of locks as the core-to-core latencies increase significantly if the cores do not use the same shared L3 cache. In Section 3.3, we see that the latencies can increase from roughly 40 ns to over 300 ns in such cases.

For this reason, many NUMA-aware locks were created over the last years [63, 84, 40, 17]. The basic idea is to avoid expensive inter-socket communication by re-ordering the waiting threads accordingly.

A technique for NUMA-aware locking is *lock conhorting* [19]. This technique allows to transform regular locks into NUMA-aware locks. Waiting threads of the same socket are grouped to avoid expensive internode coherence traffic. The cohort implementation uses a global lock and socket-local helper locks. When a thread owns the global lock, it does not release it if there are more waiting threads on the same socket. Instead, it hands it over to the next waiter of the same socket. Thereby, the global lock is untouched, and there is no inter-node communication. All threads are only waiting on the socket-local locks, while the number of back-to-back transfers on the same socket is bounded to avoid the starvation of other sockets.

The disadvantage of the cohort approach is that it requires up to 1600 bytes of in-place memory to lay out the socket local locks with enough padding to avoid false-sharing [41]. There also exist compact NUMA-aware locks that allocate the required space ad-hoc to save in-place space [17]. Kashyap et al. developed an adaptive NUMA-aware implementation that gracefully falls back to blocking to avoid excessive spinning [41].

3.2.8 Overview

To summarize this section, we now look at commonly used locks and categorize them according to the previously identified dimensions in Table 4. At this

stage, we only focus on a lock’s high-level properties. Later during the evaluation in Section 3.5, we will also investigate and add their concrete performance characteristics and implications.

We analyze two types of fair locks: a spinning lock represented by the *Queuing-RW-Mutex* and *Parking Lot*-based variations that implement strict or eventual fairness. We describe their differences and implementation in detail in Section 3.4.1.

For the *locking algorithm*, we represent *thundering*, *barging*, and *adaptive* locks. Section 3.4.1 describes the implementation of the adaptive parking lot. Although there is no explicit waiting pool in the *RW-Spinlock*, we still categorize it as a *thunder* lock, as all waiting threads are constantly racing for the lock.

3.3 HARDWARE ANALYSIS

Understanding the underlying hardware characteristics, in particular, the memory models and hierarchies, are of great importance when designing or analyzing scalable locks. For this reason, we briefly look at different architectures and their properties. We measure and plot the core-to-core latencies of the Intel, AMD, and an AWS (ARM) machine that we use for our evaluation in this chapter using the publicly available core-to-core-latency tool².

3.3.1 Intel Xeon Gold 6212U

Our first evaluation machine is a single-socket Intel Xeon Gold 6212U with 24 cores (48 Hyper-Threads). In Figure 11³, we see that all cores share the same L3 cache. The shared cache leads to stable core-to-core latencies, as shown in Figure 10. Despite the latencies within the same physical core (i.e., the Hyper-Thread latency), the measured latencies are all very close to the median of 46.9 ns.

3.3.2 AMD EPYC 7713

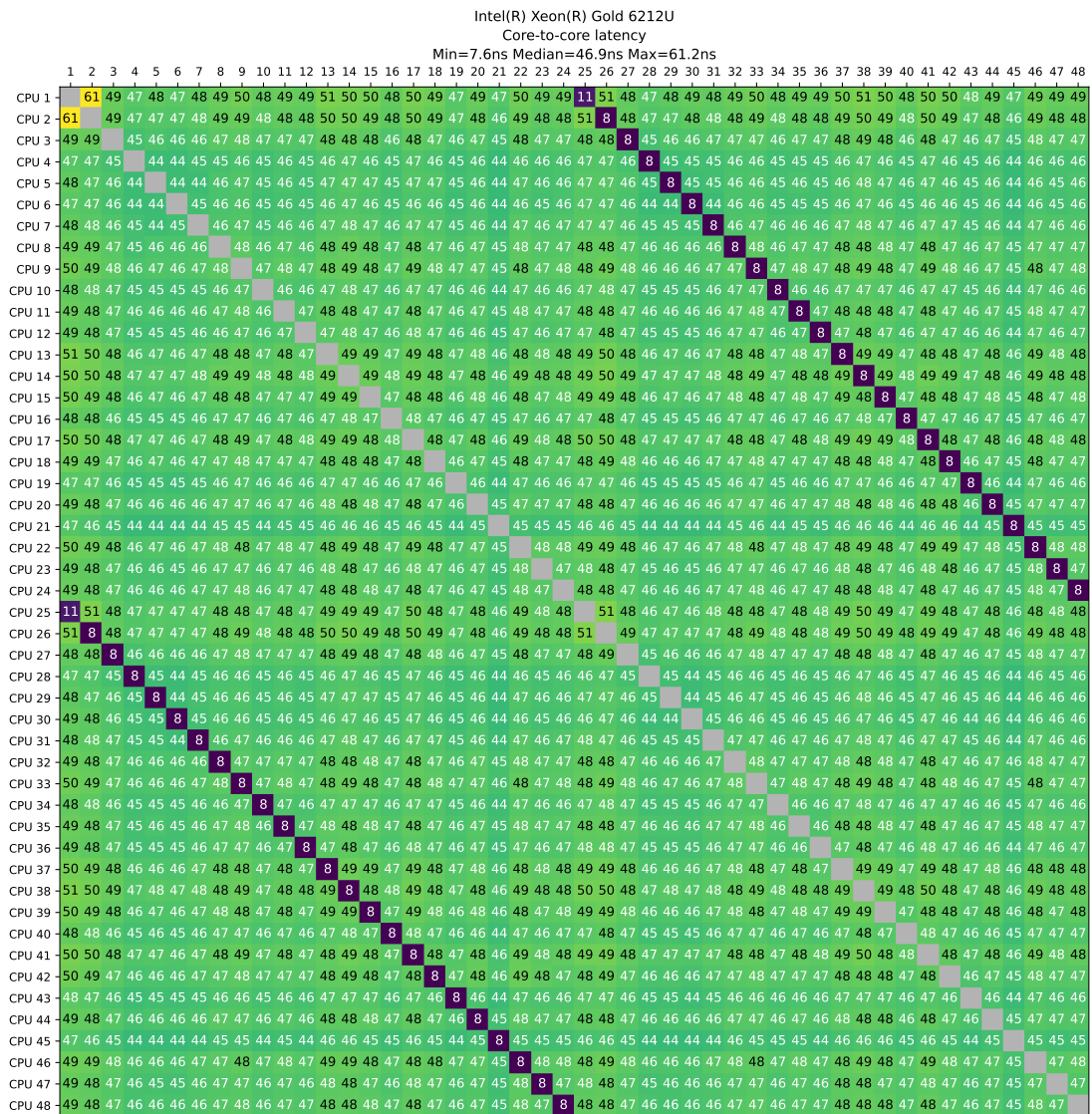
Figure 13 shows the cache topology of a dual-socket AMD EPYC 7713 retrieved by the Linux tool `lstopo`. In total, the processor has two sockets with

² <https://github.com/nviennot/core-to-core-latency>

³ Created using `lstopo`

Table 4: Conceptual overview of locks

	RW-Spinlock	RW-Mutex	Queuing-RW-Mutex	Unfair-ParkingLot	Fair-ParkingLot	EventualFair-ParkingLot
	spin_rw_mutex	shared_mutex	queuing_rw_mutex	rw_mutex + PL	rw_mutex + F-PL	rw_mutex + EF-PL
	tbb	std	tbb [49]	Sec. 2.2.4 and [6]	Sec. 3.4.1	Sec. 3.4.1
Fairness	no	no	yes	no	yes	yes (bounded)
Contention Handling	spinning	blocking (OS)	local spinning	blocking (PL)	blocking (PL)	blocking (PL)
Locking Algorithm	Thunder	Barging	Handoff	Thunder	Handoff	Adaptive
rithm						



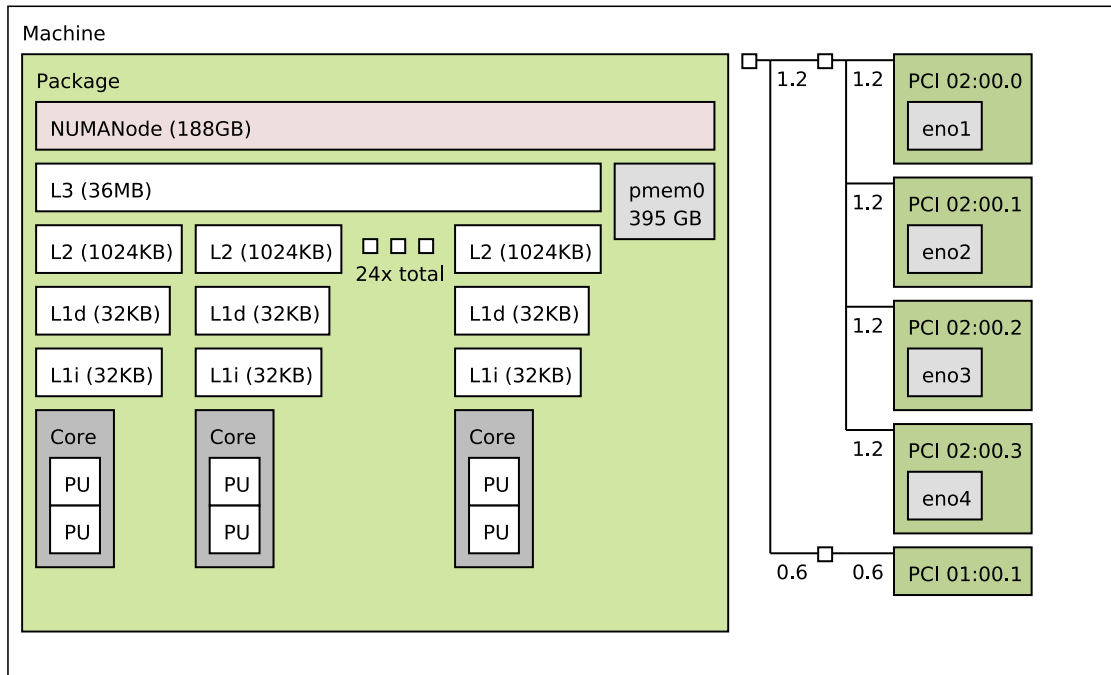


Figure 11: Topology of Intel Xeon Gold 6212U – All cores share the same L3 cache

median if we remain on the same socket. Leaving the socket increases the latencies further to over 300 ns. We also see an abnormality on the second socket. One-quarter of the cores have significantly higher latencies, although lying on the same socket. In summary, we see that communicating across different CCX is roughly an order of magnitude more expensive than keeping the data local.

3.3.3 AWS Graviton3 (Arm Neoverse v3)

To conclude our analysis, we now look at an ARM machine, namely the AWS Graviton3 with 64 cores: the biggest and newest currently available Graviton instance on Amazon AWS. Figure 14 shows that the median latency of 47,2 ns is roughly in the same ballpark as the Intel Xeon. However, there is also slightly more variance across the cores. Note that every depicted core is an actual physical core, as Graviton does not use the concept of hyper-threads⁴.

⁴ <https://docs.aws.amazon.com/whitepapers/latest/aws-graviton-performance-testing/what-is-aws-graviton.html>

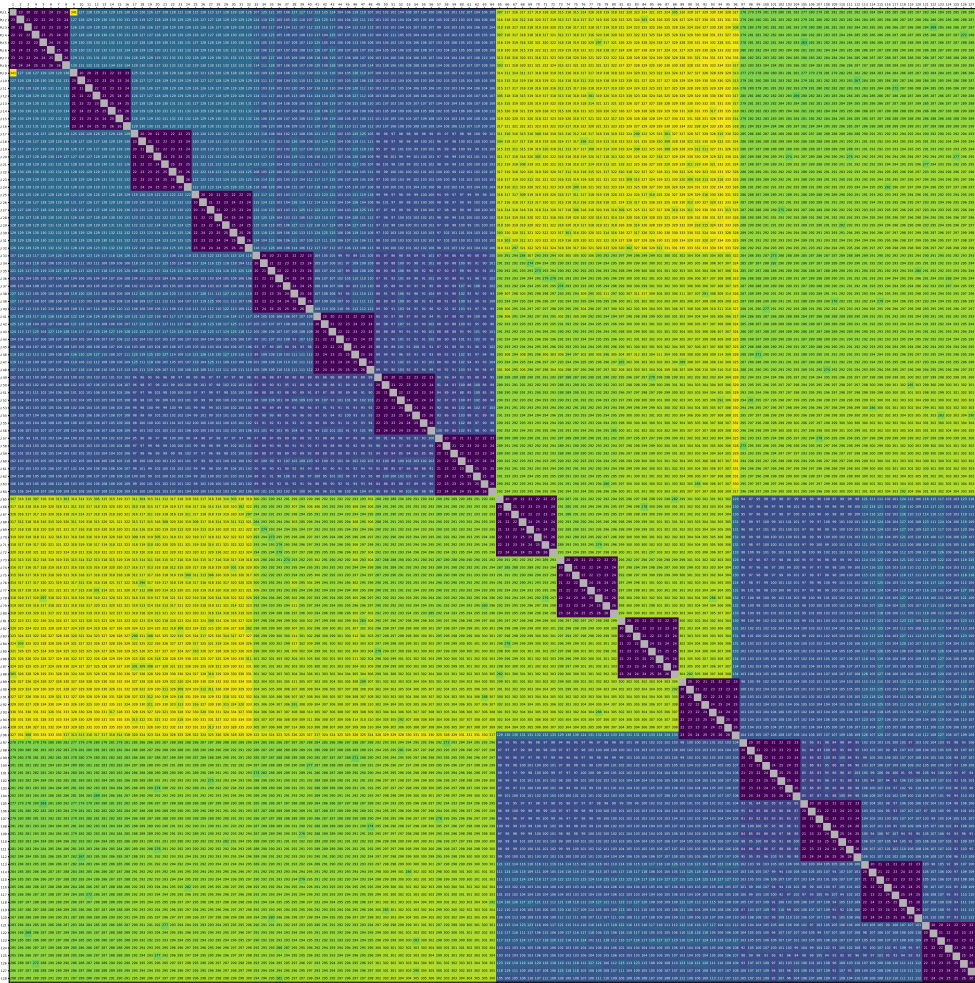


Figure 12: AMD EPYC 7713, 2x64 Cores – Core to core latencies:
On the first socket: Min=20.1ns Median=105.1ns Max=340.5ns
Across all cores (sockets): Min=20.1ns Median=285.9ns Max=340.5ns

3.4 EXTENDING THE PARKING LOT

The goal of a parking lot is to add robust contention handling to a lock without sacrificing its fast-path performance or adding any in-place space. Our basic parking lot implementation (cf. Section 2.2.4) already ensures robust contention handling. However, it lacks several features described in the previous section. In particular, its unfair nature can cause trouble in specific scenarios. In this section, we improve the parking lot incrementally by adding reasonable fairness, better read-write handling, and cache topology awareness.

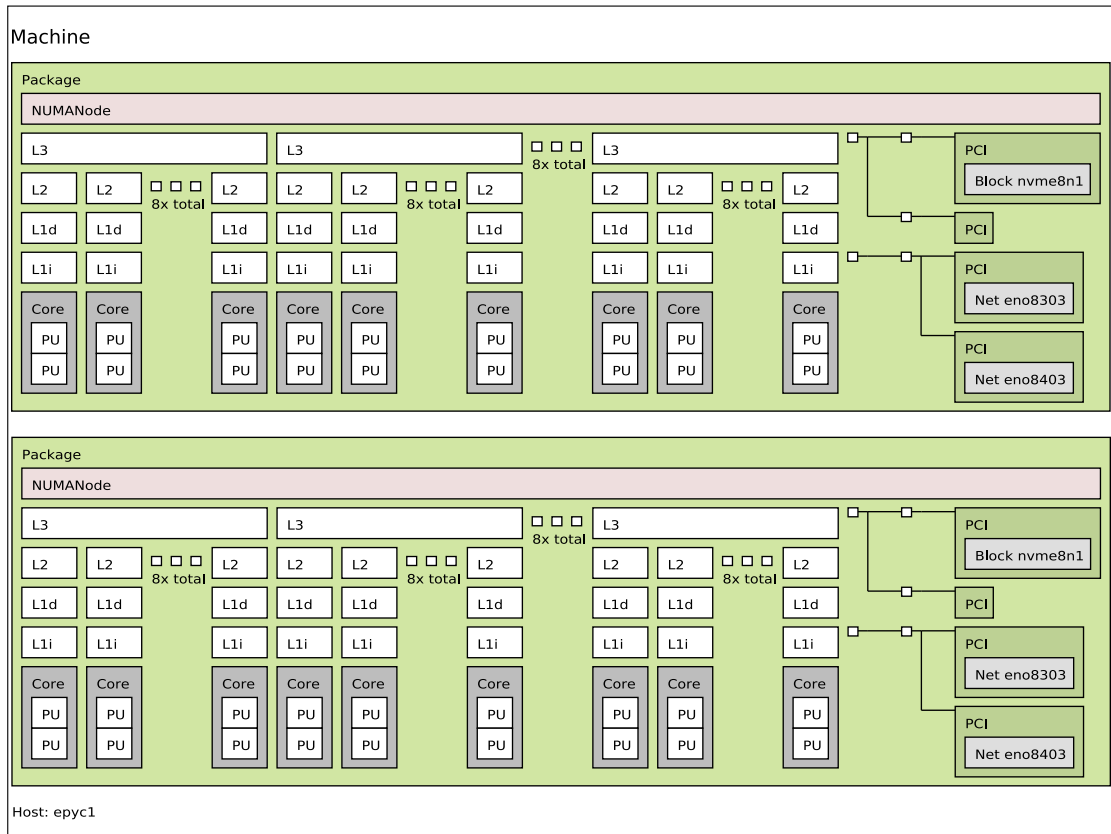


Figure 13: Topology of AMD EPYC 7713 – Every core complex (CCX) has its own L3

3.4.1 Adding Fairness

The basic parking lot is a robust solution for contention handling without sacrificing any performance on the *fast-path* (no contention) and without adding additional space to the lock itself. However, although its *wait-notify* pattern appears reasonably fair, in practice, it is not. This unfairness has two reasons: first, condition variables and the internally used futexes do not make any fairness guarantees [110]. It is up to the scheduler to decide which thread it wakes up. And second, just calling `notify_one()` does not ensure that the notified thread gets the lock. It only awakens and must race for the now unlocked/available lock. Although the unpark thread has just unlocked it, nothing stops it from acquiring the lock itself again. Retaking the same lock after releasing it is called *barging* and is often very desirable when it comes to throughput. The caches are still hot, and the thread can proceed with its work without being intercepted by others.

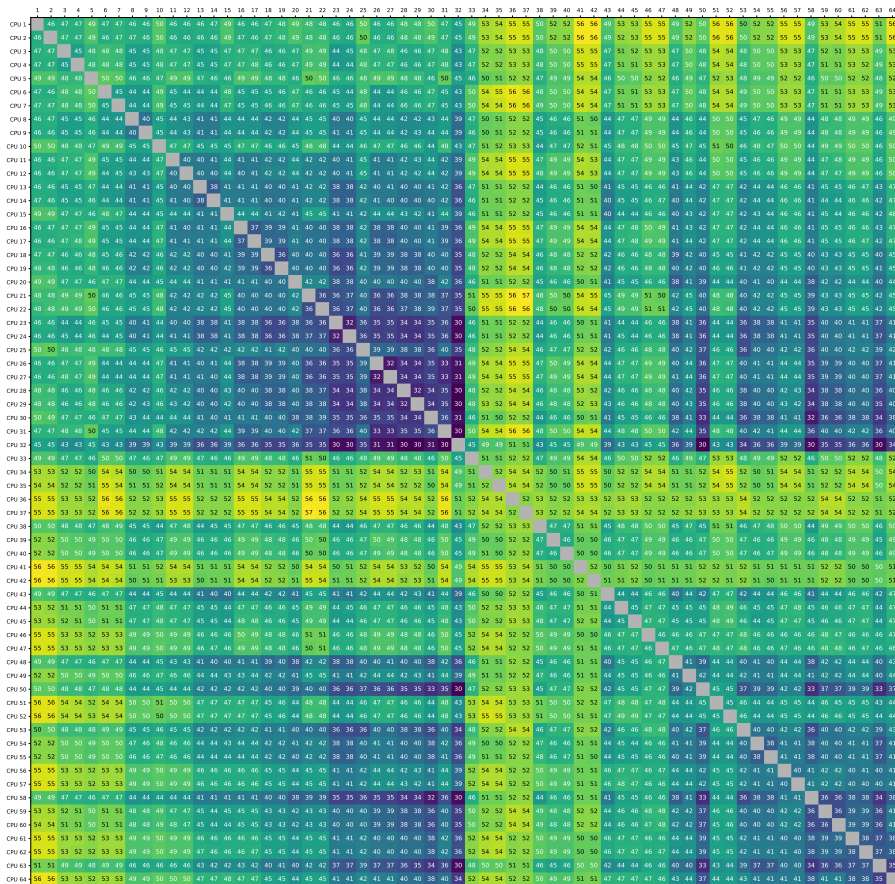


Figure 14: AWS Graviton3, 64 Cores – Core to core latencies:
 Min=30.4ns Median=47.2ns Max=58.4ns

The downsides of (un-)fairness

When a thread wakes another thread, it has the highest probability to reacquire the hot lock directly again, which can lead to starvation [26]. To test the practical implications of this behavior, we run simple micro-benchmarks where multiple threads try to acquire a lock repeatedly for a specific time. Besides the overall throughput, we measure the time the same thread gets the lock again and the tail latencies of the different locks. Our experiments (cf. Section 3.5.4) show that high throughput numbers of some unfair locks can come with extreme unfairness. Some workloads are almost executed serially, leading to high tail latencies or even the starvation of threads.

The intuitive fix would be to implement a fair parking lot that uses an internal queue for waiting threads and directly hands the lock to the next waiter. While such a *FairParkingLot* would guarantee fair lock handovers, the throughput would also stagnate dramatically. Every lock handover could come with an expensive wake operation and context switch. Especially if the critical sec-

tion is short, the throughput collapses due to the overhead of context switches during the handovers (cf. Section 3.5.5).

The developers of the Webkit ParkingLot came to the same results in their benchmarks and concluded that strict fairness is too expensive [88]. However, they and the developers of Go also noticed the issue of starvation caused by unfairness in the presence of high contention [31]. To tackle this, both Go and Webkit now use an appealing adaptation of the original parking lot idea that guarantees eventual fairness and hardly sacrifices the throughput in the case of micro contention⁵.

Implementing eventual fairness

Instead of implementing strict fairness, Webkit now uses a modified version of the ParkingLot that guarantees *eventual fairness*. *Eventual* means that the maximum time a thread must wait for a lock is bound by a certain threshold. For Webkit, that is 1 ms times the number of other waiting threads or respectively longer if the critical section takes longer.

The language Go uses a similar technique since version 1.9 [31]. From this version, every mutex has two modes: normal and starvation. Like the parking lot approach, the waiting threads enter a FIFO queue, and the longest waiting thread is woken up during the unlock. In the *normal* mode, a thread is only woken up without a lock handover to encourage *barging* for higher throughput. If a thread fails to get the lock for more than 1 ms, it sets the lock to *starvation* mode, enforcing a fair handover during the next unlock⁶.

We implemented an eventual fairness lock that builds on the Webkit implementation [104]. The general principle of the parking mechanism is shown in Figure 15. When a thread must wait on a lock, it goes to the parking lot (a global static hash table) and identifies its parking space using the lock's memory address. Within a parking space, the threads wait in a queue. Every parking space tracks the `nextFairTime` that indicates when the next fair handover should happen. This timestamp is increased by a fairness threshold (e.g., 1 ms) after every fair handover. Unparking operations that do simple unlockings (i.e., encourage unfair “barging”) instead of fair handovers, do not change the `nextFairTime`. Apart from those changes, the parking lot works similarly to the basic parking lot described in Section 2.2.4.

⁵ <https://github.com/WebKit/WebKit/commit/24e899259cf1724af10cb1bf1b8bb740d5b69c4b>

⁶ <https://go.dev/src/sync/mutex.go?s=5772:5796>

The locks must also be slightly adapted to work with an eventual fair parking lot. Listing 5 shows an exemplary implementation based on the original Webkit code [104]. While the fast (uncontended) path remains untouched, we now have to check how we were unparked: We could either own the lock after a *direct handover*, or have to race for the lock if we are in *barging* mode.

The unlock operation must also be adapted as shown in Listing 6. In the unfair parking lot, we could release the lock before notifying waiting threads in the parking lot. This is no longer possible in a fair parking lot, as we must wait to unlock the lock before we unpark someone to enable fair handovers. For this reason, we hold the lock while entering the parking lot and pass an unpark callback to the parking lot that holds the lock-specific unlock or handover logic (Listing 7).

3.4.2 Adding Read-Write Lock Support

Our parking lot should support waiting for exclusive and shared locks. The current approach needs to distinguish between shared and exclusive waiters. While this is correct, it does limit the parallelism if multiple readers are waiting. In the worst case, all waiting readers would get the lock sequentially without any read concurrency.

For this reason, we adapt the QueueElements so that they can store a shared or exclusive number of waiters as displayed in Figure 16. When a new reader arrives, it can join a shared Queue-Element by incrementing the counter and waiting on the same condition variable. When the waiting readers are unparked, the lock state can be set atomically to the exact number of waiting readers. This reduces the number of atomic writes, as the lock can be handed over to all readers using a single store instruction.

Regarding the scheduling or rather enqueueing of new readers in the parking lot, there are different options. The fairest option would be that a reader is only allowed to join a shared queue element when it is at the end of the queue, i.e., a reader is not allowed to skip the line and join earlier readers.

The other option would be to allow skipping the line and group all waiting readers in a single element. As shown in Figure 16, a new reader can find this element efficiently in the queue as we directly point to it in an additional sharedWaiter pointer.

Previous work showed that the latter approach yields better performance and scalability [9]. It also makes memory management more straightforward,

Algorithm 5: Lock() implementation of eventual fair lock

```

1 // Based on Webkit's implementation [104]
2 void lock()
3 {
4     unsigned spinCount = 0;
5     for (;;) {
6         auto value = data.load(memory_order::relaxed);
7         if (!isLocked(value)) {
8             // Try to get the lock
9             if (data.compareExchange(value, value | exclusiveMask)) {
10                return;
11            }
12        }
13
14        // If no one is waiting: we spin some time first
15        if (!isMarkedAsWaiting(value) && ++spinCount < 40) {
16            yield();
17            continue;
18        }
19
20        if (!isLocked(value)) {
21            continue;
22        }
23
24        // Make sure that the parking bit is set
25        if (!isMarkedAsWaiting(value)) {
26            if (!data.compareExchange(value, value | waitMask)) {
27                continue;
28            }
29            value |= waitMask;
30        }
31
32        // As long as the lock is locked and marked as parking, we park
33        assert(isLocked(value) && isMarkedAsWaiting(value));
34        auto parkResult = ParkingLot::compareAndPark(&data, value);
35        if (parkResult.wasUnparked) {
36            switch (parkResult.token) {
37                case DirectHandoff:
38                    // The lock never released and directly handed to us
39                    assert(isLockedExclusive(data.load()));
40                    return;
41                case Barging:
42                    // Someone unparked us without handing us the lock.
43                    // We now have to race for the lock
44                    // but may lose to barging threads.
45                    break;
46            }
47        }
48    }
49 }

```

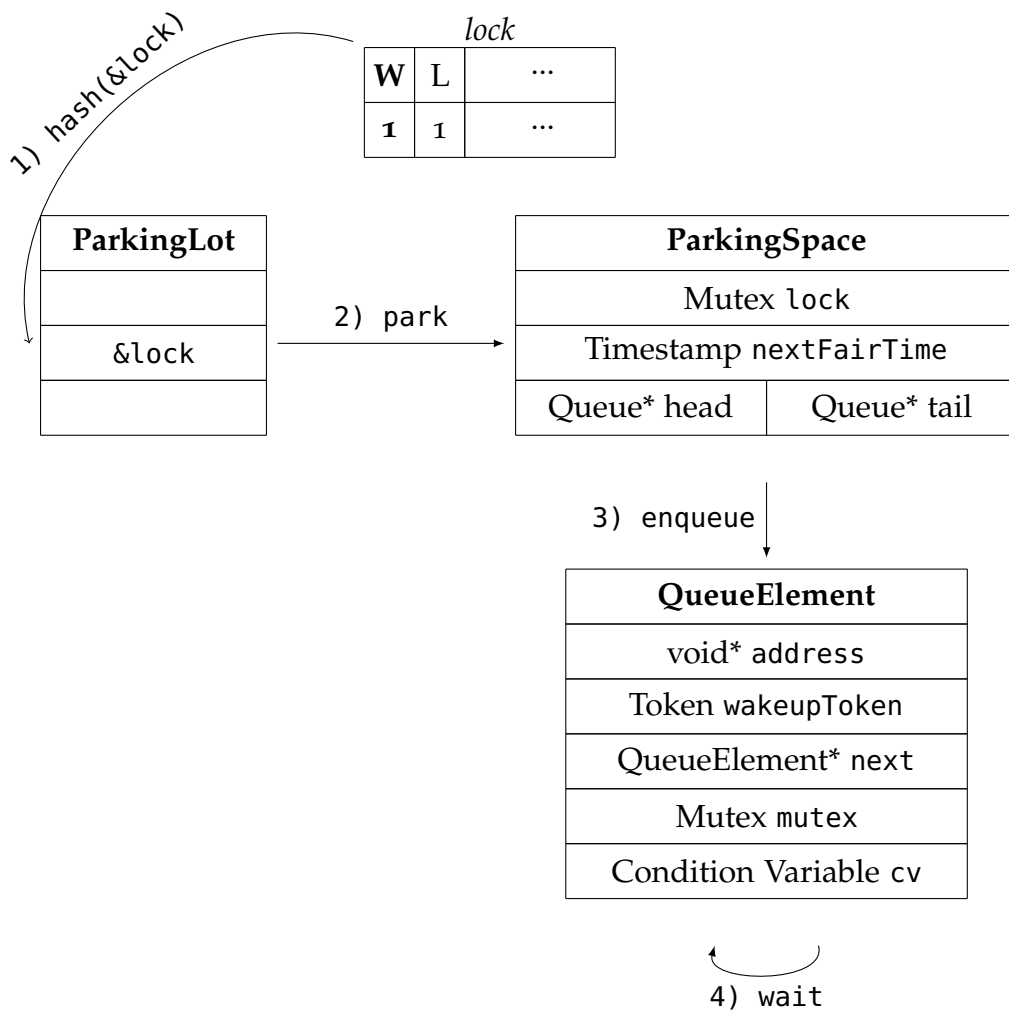


Figure 15: Eventual Fair Parking Lot – When parking, every thread adds itself to the queue in the Parking Space belonging to the lock’s address. In contrast to the basic parking lot, we now manage the waiting threads in a queue and store the time when we want to become fair.

Algorithm 6: Unlock() implementation of eventual fair lock

```

1  bool unlock() {
2      auto value = data.load();
3      assert(isLockedExclusive(value));
4
5      // Fast path
6      if (!isMarkedAsWaiting(value)) {
7          if (data.compareExchange(value, 0, std::memory_order::release)) {
8              return;
9          }
10     }
11     // Slow path: We must unpark someone
12     // We pass the lock specific unparkCallback (Listing 7) to the parking lot
13     ParkingLot::unparkOne(
14         &data,
15         [&](UnparkResult result) { return unparkCallback(result); });
16 }

```

as we need to keep at most two instances of shared queue elements: One for newly arriving and one for departing readers. Thus, we can keep them pre-allocated within the Parking Space.

3.4.3 Adding Cache Topology Awareness

The topology of modern CPUs brings new challenges when designing scalable systems. To further increase the number of cores, chip manufacturers spread them across multiple NUMA sockets or CPU core complexes (CCX). In Section 3.3, we already saw how much the cache latency increases when the communication crosses a socket boundary or leaves a CCX in AMD's Epyc. The difference in latency can be over 300 ns in the architectures we analyzed.

Thus, when designing locks, it is critical for optimal performance to keep the underlying CPU architecture and cache topology in mind to minimize cache traffic [40].

Cache traffic does not only arise when we access a lock but also when we process the data protected by the lock. For instance, in a B Tree, when getting the lock for a node, we are also likely to access or change the content of the node. Ideally, the lock and the node's content are still in the cache of the thread that gets the lock. By making a lock topology-aware, we can increase the likelihood of that happening [40].

Algorithm 7: Unlock or handover callback

```

1 // Extending Webkit's implementation to support read–write locks [104]
2 Token unparkCallback(UnparkResult& result)
3 {
4     // This callback is only called when we hold the lock
5     // and decide whether to unlock (allow barging) or handover the lock
6     // Thus, we own the lock at the moment, and the parking bit should be set
7     auto curValue = data.load();
8     assert(isLocked(curValue) && isMarkedAsWaiting(curValue));
9
10    if (result.didUnparkThread && result.timeToBeFair) {
11        // It's time to be fair: We hand the lock over to the waiter(s)
12        uint64_t newValue = 0;
13        if (result.exclusive) {
14            // Prepare exclusive handover
15            newValue = exclusiveMask;
16        } else {
17            // Prepare shared handover to numWaiters readers
18            newValue = result.numWaiters;
19        }
20        // Should we keep the parking bit?
21        if (result.mayHaveMoreThreads) {
22            newValue |= waitMask;
23        }
24        // Prevent pending writers from starvation
25        if (result.moreExclusiveWaiters) {
26            assert(result.mayHaveMoreThreads);
27            newValue |= exclusiveMask;
28        }
29
30        if (curValue == newValue) {
31            // NO-OP: lock does not have to be changed for this handover
32            return DirectHandoff;
33        }
34        data.store(newValue, memory_order::release);
35        return DirectHandoff;
36    }
37
38    // Unlock to allow barging
39    uint64_t newValue = result.mayHaveMoreThreads ? waitMask : 0;
40    data.store(newValue, memory_order::release);
41    return Barging;
42 }

```

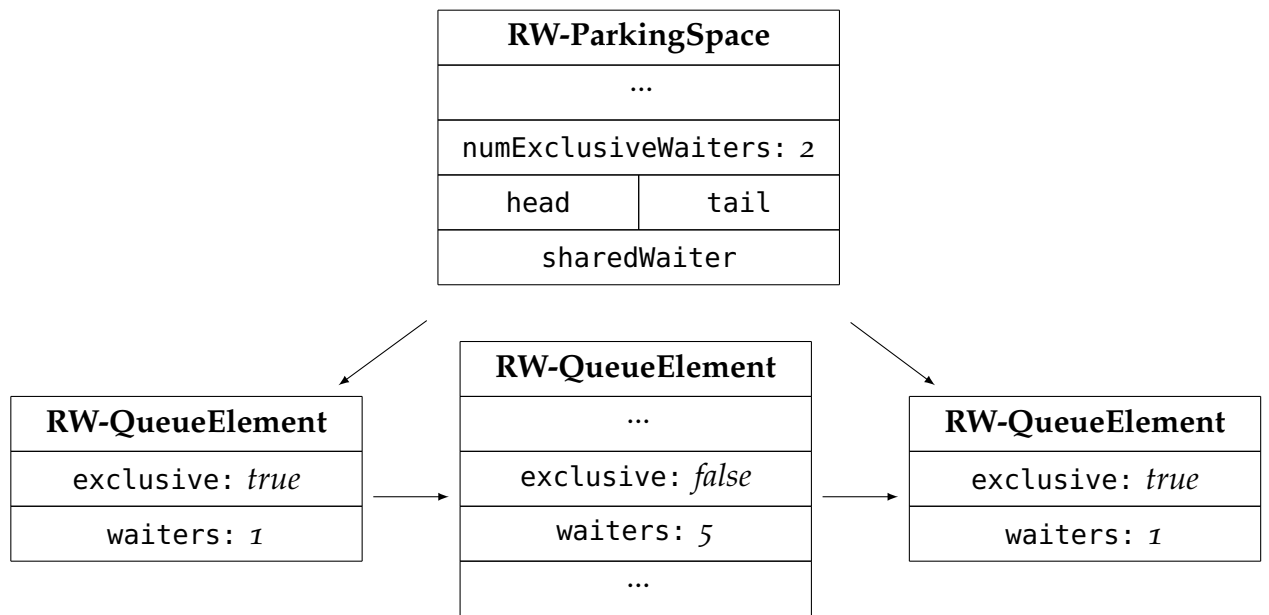


Figure 16: EventualFairParkingLot: Read-Write – We now distinguish between readers and writers. Readers can wait on the same queue element for better read concurrency. Using an additional `sharedWaiter` pointer, readers can directly join the shared `RW-QueueElement` without iterating over the list.

When we recap the topologies we saw in Section 3.3, we remember that NUMA nodes and AMD’s CCX use separate L3 caches. As soon as we leave a shared L3 cache, the latencies increase drastically. For this reason, we extend our Parking lot to use a queue per L3 node instead of having a single global queue per lock as shown in Figure 17. When we unlock or hand over a lock, we prefer to keep it within our current L3 node. If no other thread is waiting on our node, we move on to the next one. To avoid starvation of sockets, we apply the same fairness pattern as for the normal queue: When our fairness threshold time is reached, we also move on to the next socket.

The advantage of the parking lot approach over existing NUMA-aware solutions is that we can include the topology awareness dynamically into every system [63, 17, 19, 41]. We do not have to make our in-place lock any bigger or know the system’s architecture at compile time. We can identify the system’s architecture once at startup time when we construct the global static parking lot. In Linux, the cores that share the same L3 cache can be identified by reading the contents of `/sys/devices/system/cpu/cpu[X]/cache/index3/shared_cpu_list`.

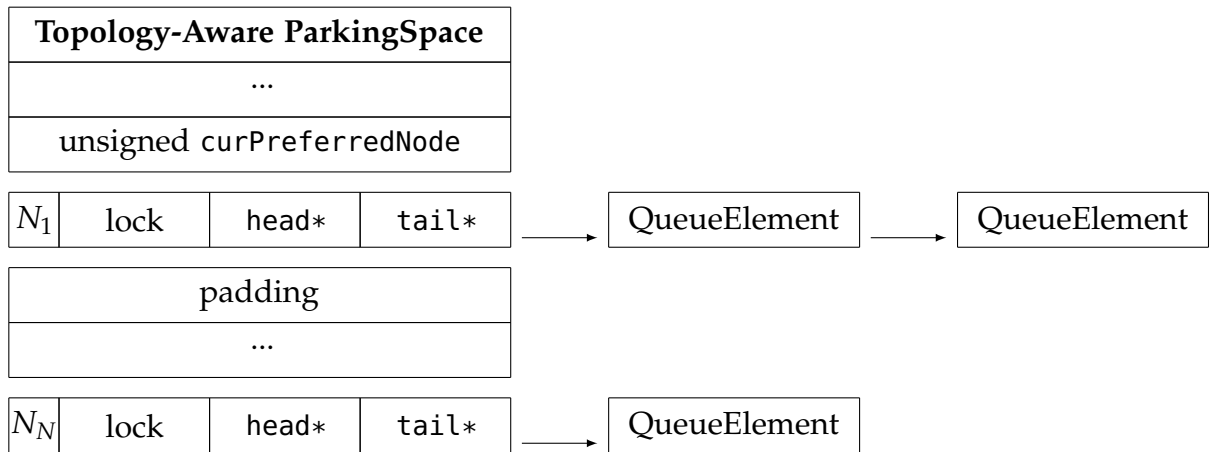


Figure 17: Cache Topology Awareness – *Instead of maintaining a single global waiting list, we can also make our lock cache topology aware by grouping threads from the same L3 cache or NUMA node. The cache topology of a system (i.e., the number of required nodes) can be determined dynamically when the parking lot is constructed.*

3.5 EVALUATION

In this section, we evaluate the performance of our extended parking lot implementation (Section 3.4). We compare it to blocking locks from the C++ standard library (`std::mutex` and `std::shared_mutex`), a fair (`tbb::queuing_rw_mutex`) and an unfair spinlock (`tbb::spin_rw_mutex`) from the Intel TBB library, and our basic parking lot implementation as described in Section 2.2.4.

We run all experiments on Ubuntu 22.04 using GCC 12. We use three different processor architectures as described in Section 3.3: two x86 machines: a single-socket Intel Xeon and a dual-socket AMD Epyc, and an AWS Graviton 3 representing ARM v8.

3.5.1 Fast path

First, we look at the raw performance counters of the locks to prove our philosophy of extending the capabilities of a lock with a parking lot without hurting its fast path. Table 5 shows that our parking lot uses the fewest instructions if there is no contention. Under contention, when 48 threads repeatedly acquire the same lock, using a parking lot still uses the least number of in-

Table 5: Performance counters normalized per lock acquisition – On Intel Xeon 6212U

	Fast path (1 thread)		
	cycles	instructions	IPC
EventualFairParkingLot	63.25	33	0.52
UnfairParkLock	64.25	36	0.56
tbb::spin_rw_mutex	57.78	48	0.83
tbb::queuing_rw_mutex	67.03	99	1.48
std::mutex	55.67	73	1.31
std::shared_mutex	95.47	107	1.12
	Contention (48 threads)		
	cycles	instructions	IPC
EventualFairParkingLot	5164.58	1343.62	0.26
UnfairParkLock	10786.66	1526.04	0.14
tbb::spin_rw_mutex	178751.99	21220.55	0.12
tbb::queuing_rw_mutex	150231.02	51179.31	0.34
std::mutex	19982.61	2156.43	0.11
std::shared_mutex	43695.15	4495.22	0.10

structions. While the spinning locks “burn” an increasingly high number of cycles.

3.5.2 Contention Handling

In this experiment, we want to focus on micro contention and how the lock deals with it. We let an increasing number of threads compete for the same lock. The critical section itself is super short: increment a global variable. Ideally, the performance stays as close to the single-threaded performance as possible.

In Figure 18, we see that the locks built on top of blocking futexes dominate this experiment, while the spin locks are highly affected by cache contention. At a certain point, all locks roughly plateau. Only the *UnfairParkLock* gets overtaken by the *EventualFairParkingLot* at a high level of contention. We assume that the *thundering* nature of the unfair lock causes this behavior: All threads wait on the same condition variable and are released as a “thundering herd”. In contrast, the eventual fair parking lot uses a *barging* queue, where only one thread at a time is notified. For more details on the different locking algorithms, see Section 3.2.4. While the thundering might increase the throughput under low contention, the barging locks are seemingly more robust.

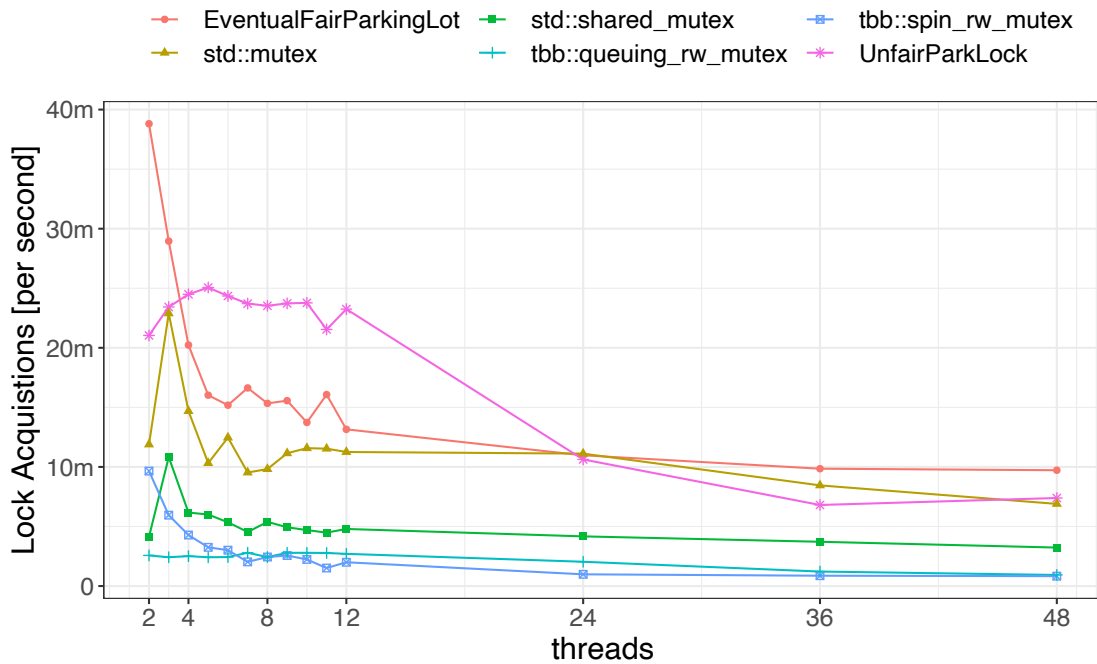


Figure 18: Micro Contention – On Intel Xeon 6212U

3.5.3 Comparison with ARM

In Figure 19, we repeat the contention experiment of the previous Section 3.5.2 on an ARM machine. We use an AWS-hosted Graviton 3 with 64 cores as described in Section 3.3.3. Despite the different memory models, the locks behave similarly under contention. One difference is that the throughput on the Graviton 3 already stabilizes at four threads. On the Xeon machine, some locks performed better under light contention (less than 12 threads) before they reached a plateau. Also, the `std::mutex` that was almost on par with the parking lots on x86 now falls behind in ARM. In summary, a parking lot approach is a robust approach for our ARM representative.

3.5.4 Fairness

In this section, we analyze the fairness of the locks. We use the relaxed definition of fairness, which claims that a lock is fair if the number of lock acquisitions per thread is uniform over a given time.

To quantify the fairness of the different locks, we use Jain’s fairness index [39]:

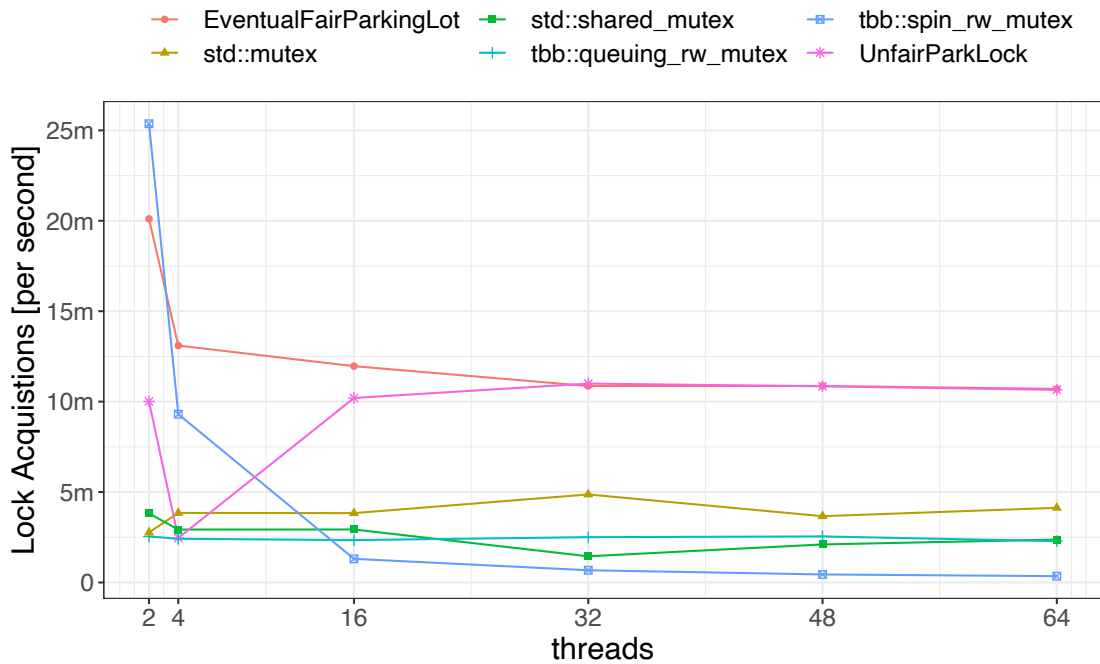


Figure 19: Micro Contention – On AWS Graviton3 (ARM)

$$\frac{(\sum_{i=1}^n x_i)^2}{n \cdot \sum_{i=1}^n x_i^2} \quad (1)$$

Whereas n is the number of threads and x_i is the number of times thread t_i got the lock. A lock can score 1.0 in the best-case (fair) or $1/n$ in the worst-case (maximal unfair).

In Figure 20, we show how the fairness of locks varies when increasing the number of threads or the length of a critical section. We ran and measured every setup for five seconds. We see that the fair and our eventual fair lock stay perfectly fair in all cases. The unfair locks, however, and in particular the blocking mutexes, become extremely unfair when the length of the critical section increases. When looking at the distribution of lock acquisitions in Figure 21, one sees that most threads do not get the lock at all, i.e., they starve. In fact, only one or two threads make up for the entire throughput of lock acquisitions. The fair locks do not suffer starvation which is reflected by the compact boxes in the plot. The starvation issue of the blocking mutexes is an artifact of their barging nature (cf. Section 3.2.4).

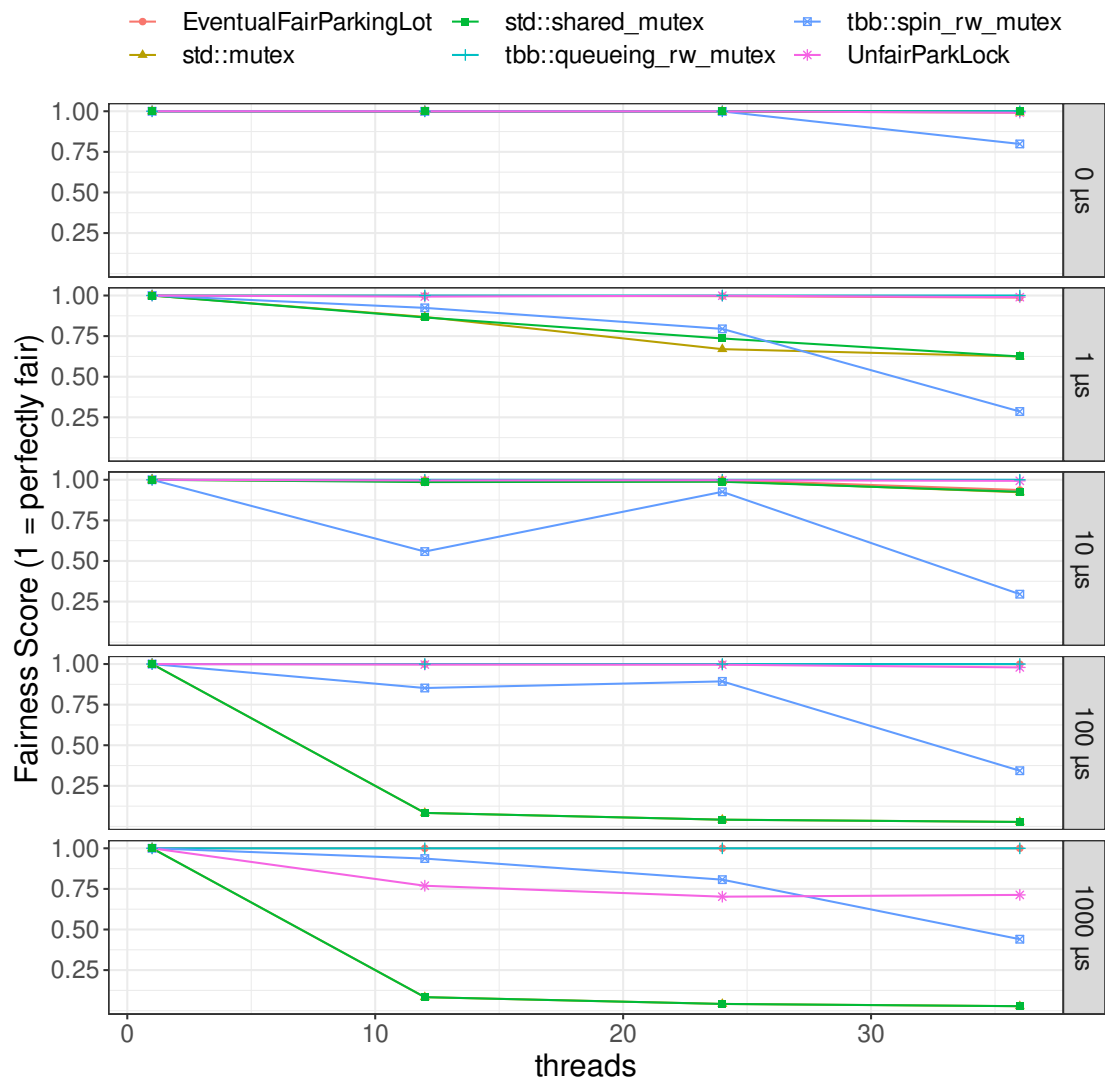


Figure 20: Fairness of locks in different scenarios – Some locks become very unfair with increasing numbers of threads or longer critical sections (Intel Xeon 6212U)

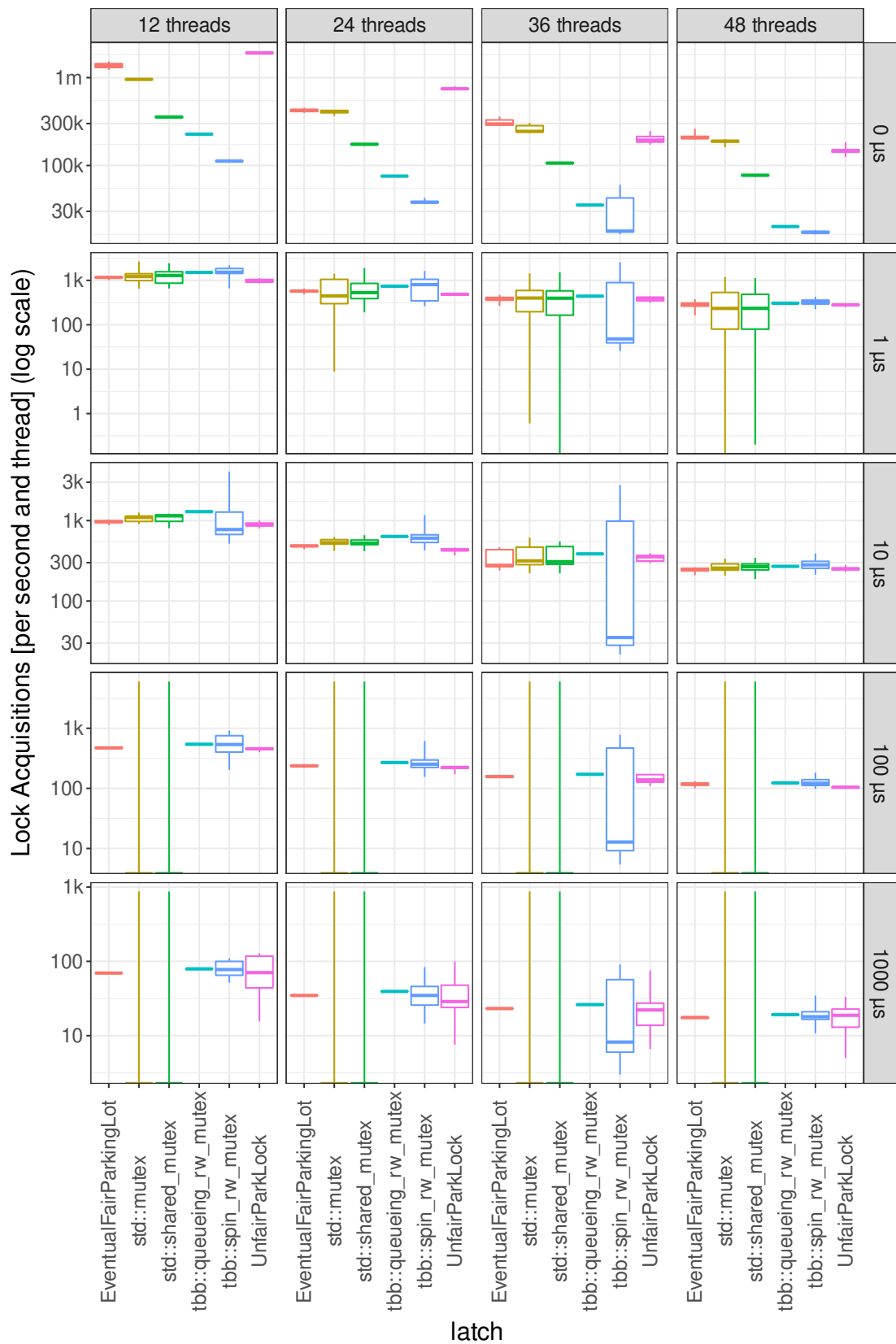


Figure 21: Lock Acquisitions per thread – Using different numbers of threads and increasingly long critical sections. With unfair and blocking locks more and more threads suffer from starvation (Intel Xeon 6212U)

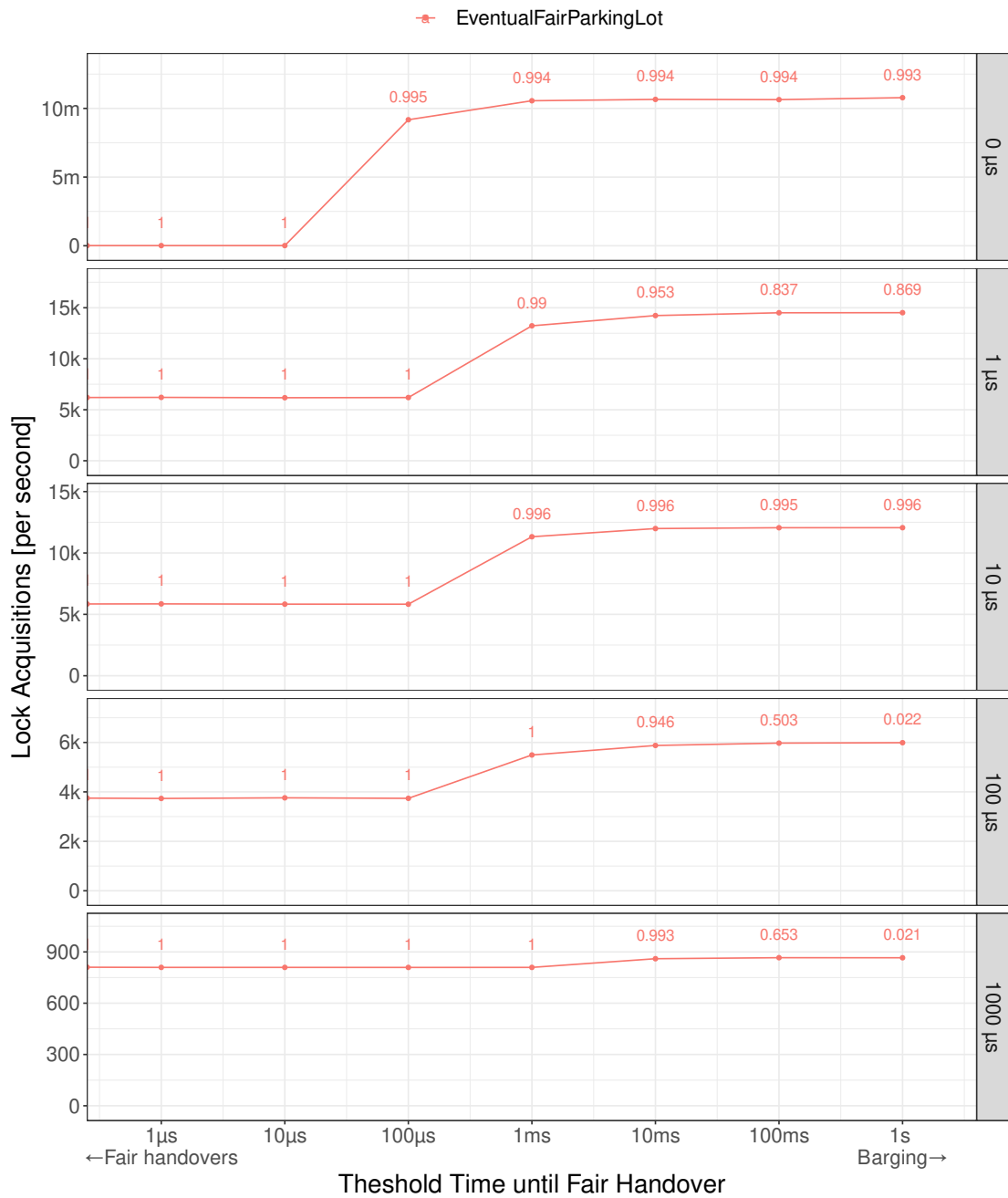


Figure 22: Finding an optimal fairness threshold – When setting the fairness threshold to zero or a short time, our lock will always do a fair handover. When increasing the threshold, we allow more barging. Up to a certain point, this improves the throughput at the cost of fairness (score is printed next to every point). (Intel Xeon 6212U)

3.5.5 Tradeoff: Throughput vs Fairness

In this section, we want to find the optimal setting for the *fairness threshold time*. When a thread is parked for longer than this time, the unlocking thread hands the lock directly over to the waiting thread. When the threshold time still needs to be reached, it notifies the waiting thread but does not hand over the lock to the waiter. The thread unlocks the lock and makes it available to grab for the thread that reaches the lock first. This so-called “barging” allows higher throughput as another thread that is more “ready” could take the lock first. A potential context switch of the notified thread does not stall the progress of the lock. Setting the threshold time allows trading fairness for throughput to a certain degree.

To figure out the optimal setting, we vary the length of the critical section and track both the resulting throughput and the fairness. We run the experiment with different fairness threshold times ranging from 0 (i.e., every unlocking is a fair handover) to a very high value that never does handovers (simulating a purely barging lock). In Figure 22, we see that the throughput is very poor when we are (too) fair, and the critical section is short. Also, in the case of short critical sections, even unfair locks are reasonably fair. Vice versa, with longer critical sections, the overhead of fair handovers diminishes, and the barging locks become noticeably unfair.

Based on this experiment, we find an optimal threshold setting of 1 ms. At this point, the throughput begins to plateau, and the fairness starts to drop. This result confirms the 1 ms setting used by WebKit and Go [88].

We plot the throughput versus the achieved fairness score in Figure 23 to put the different fairness settings into perspective. Almost all locks achieve the same throughput, but the fairness score on the y-axis varies a lot. The blocking mutexes from the standard library are highly unfair in this setup. The `EventualFairLock` is—using the optimal setting as identified above—perfectly fair and almost on par with the throughput of the slightly less fair spin locks without creating the typical spinning problems like CPU burning or priority inversion in the scheduler.

3.5.6 Topology Awareness

In this experiment, we want to analyze the performance benefits of cache topology awareness as described in Section 3.4.3. We run those experiments on the dual-socket AMD Epyc machine with 16 CCX, i.e., 16 separate L3 caches.

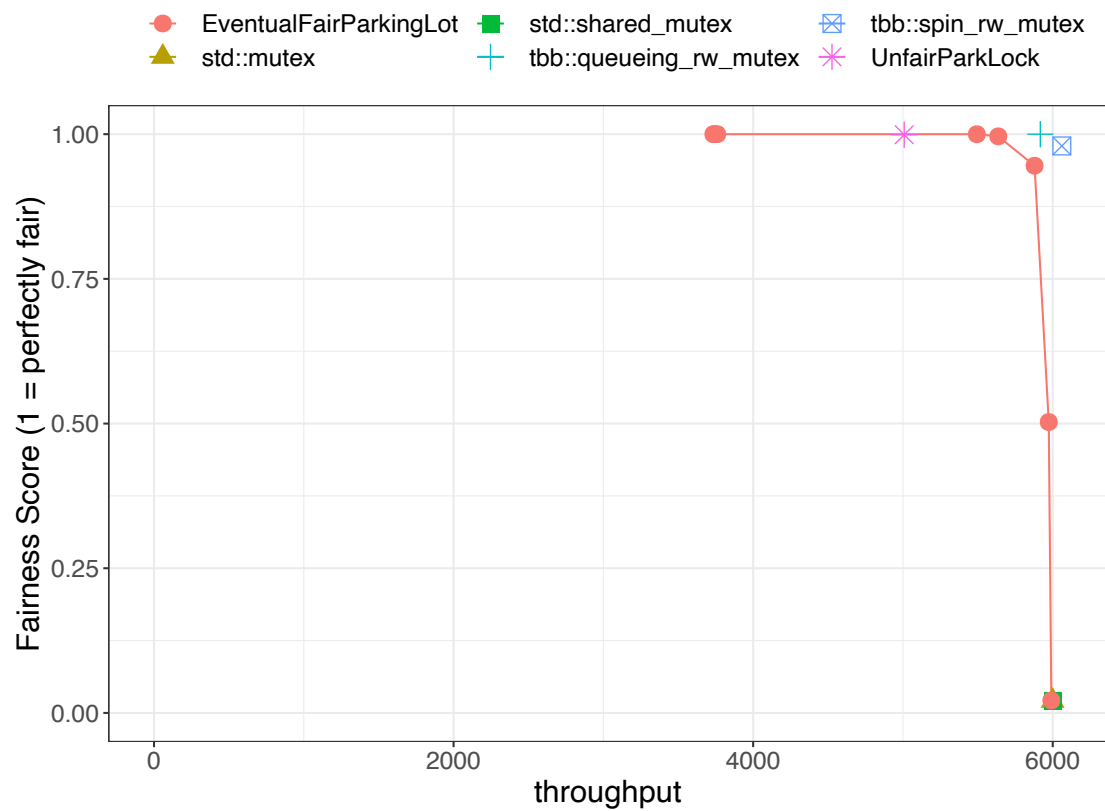


Figure 23: Fairness vs. Throughput – We compare different fairness settings of our lock with different fairness scores and throughputs of others.
 Critical section length = 100 microseconds (Intel Xeon 6212U, 48 Threads)

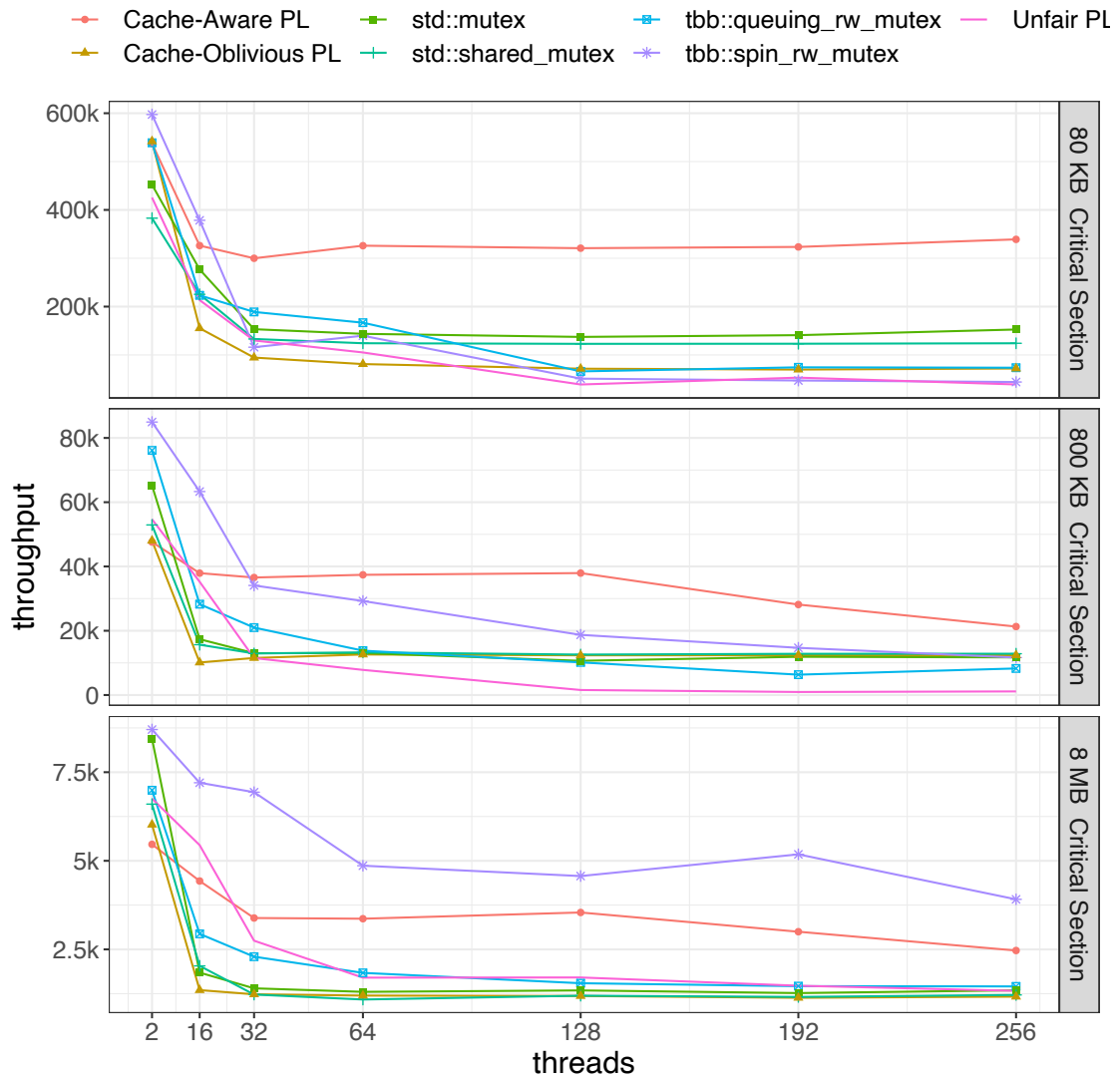


Figure 24: Topology Awareness – Measuring the effects of topology awareness when processing critical sections on a dual-socket machine with 16 CCXs (separate L3 Caches) (AMD Epyc 7713).

In Figure 24, we plot the throughput for increasing threads and different critical section sizes. A lock protects a vector of contiguous 64-bit integers. During a critical section, a thread increments all integers. The number of tuples ranges from 10k (80 KB) to 1 million (8 MB). For reference, Umbra’s default (and minimal) page size is 64 KB [77].

All used locks, except for our *cache-aware* parking lot, are cache-oblivious. For reference, we also include the numbers of our parking lot where the cache-awareness is disabled (i.e., we only use a single global waiting list instead of one per L3 node). We see that the performance of the cache-oblivious locks degrades when adding more threads of other CCX with separate L3 caches. We see another drop in performance when adding threads from the second socket (step from 64 to 128 threads). After this point, we have saturated all physical cores and use SMT (AMD’s “hyper-threads”). The performance plateaus, as the core-to-core latencies do not get significantly worse (cf. Section 3.3.2).

In contrast, to the cache-oblivious locks, our cache-aware parking lot can maintain a significantly better throughput. Shipping the lock ownership to threads on the same L3 cache node avoids expensive cache misses as the lock and the protected data are likely still cached. For a big critical section size of 8 MB, the `tbb::spin_rw_mutex` outperforms our lock as we cannot keep the lock on the same L3 node that long anymore due to fairness reasons. When looking at the fairness score of the spin lock we see a drop to only 0.27 in this scenario, while the parking lot stays almost perfectly fair. This unfairness introduces cache-friendly access patterns by accident. The spinlock also saves the cost of context switches which occur in the blocking locks now that the critical sections are long enough to increase the likelihood of a waiting thread starting sleeping.

3.5.7 Summary

We identified different dimensions and goals for locks in Section 3.2.8. So far, we have only summarized the high-level properties of the different locks. Now that we have tested the locks in various experiments, we can extend the previous table by practical performance implications. Table 6 summarizes our concrete results and highlights the strengths and weaknesses of every lock.

We saw that *fairness* is vital to keep the tail latencies low. However, strict fairness in combination with *blocking* contention handling introduces a hefty performance penalty due to expensive context switches. To avoid this overhead,

we relax the fairness in the *parking lot* to eventual fairness. We still get the benefits of low tail latencies without risking the starvation of any threads while keeping the high throughput of fast unfair locks. A spinning fair lock, like a *Queuing-RW-Mutex*, is no viable alternative for a database system due to the inherent problems of spinlocks when it comes to contention.

With the parking lot approach, we can make underlying locks fair without adding any in-place space. Also, the fast path in the lock acquisition is not affected when backing a lock up with a parking lot.

Another benefit of a parking lot is that we can organize the waiting threads in a cache-friendly way for better performance. During the startup of the database system, when we construct the global parking lot, we can dynamically identify the underlying cache topology of the machine and construct our *parking spaces* accordingly. Having a separate waiting queue for every L3 cache node resulted in superior performance in our experiments due to the reduced cache misses. When a lock is kept on the same L3 cache node, the lock and the protected data itself are very likely still in the cache for the following thread. The eventual fairness implementation naturally guarantees that no L3 cache node starves.

In summary, we conclude that our eventual fair parking lot implementation meets all the goals we identified in Section 3.2 and thus is an attractive option for a database system.

3.6 RELATED WORK

Despite the importance of scalability and concurrency for a system, there is surprisingly little research on latches in database systems [6]. The database community focuses mainly on the synchronization of index structures or high-level concurrency control [32, 59, 103, 22, 50, 82].

In contrast, the system's community studied locks extensively over the past decades and also made advances in reflecting modern architectures [13, 93, 80].

Over the last years, there were several proposals for making locks NUMA-aware [63, 84, 40, 17]. However, those locks do not integrate perfectly into database systems, as they were predominantly designed for use in kernel space and rely on spinning or use a lot of in-place memory [19]. There are also adaptive NUMA-aware locks that fall back to blocking, but they require additional ad-hoc memory allocations [41].

Table 6: Summary of locks and their performance

	RW-Spinlock	RW-Mutex	Queuing-RW-Mutex	Unfair-ParkingLot	Fair-ParkingLot	EventualFair-ParkingLot
	spin_rw_mutex	shared_mutex	queuing_rw_mutex	rw_mutex + PL	rw_mutex + F-PL	rw_mutex + EF-PL
	tbb	std	tbb [49]	Sec. 2.2.4 and [6]	Sec. 3.4.1	Sec. 3.4.1
Tail latencies	high	high	low	high	high	low
Fairness	no	no	yes	no	yes	yes (bounded)
Throughput	high	med.	med.	high	low	high
Uncontended Access	fast	med.	med.	fast	fast	fast
Contention Handling	spinning	blocking (OS)	local spinning	blocking (PL)	blocking (PL*)	blocking (PL)
Cache Aware	no	no	no	no	no	yes
Locking Algorithm	Thunder	Barging	Handoff	Thunder	Handoff*	Adaptive
Space summption	con-8 bytes	56 bytes	8 bytes	8 bytes	8 bytes	8 bytes

(*) fair hand-overs require expensive context switches

Our eventual fair parking lot generalizes the concept of NUMA awareness to L3 cache awareness. It also supports arbitrary cache topologies like AMD's CCX groups.

Another aspect we cover in this chapter is the fairness of locks. Various proposals exist for fair locks in academia [49, 101, 4, 64]. Filip Pizlo was seemingly the first who implemented a relaxed version of fairness using an eventual fair parking lot [105]. The language *Go* adopted this concept to solve the problem of mutex starvation [31]. Our implementation builds upon the ideas of the original Webkit parking lot and extends it to support read-write mutexes and cache awareness.

3.7 CONCLUSION

Locks and synchronization are very complex topics: The various workloads, machines, and cache topologies make it almost impossible to find the one lock to “rule them all”.

However, in this chapter, we analyzed different locks and setups and identified a clear recommendation when building a lock for a general-purpose application like a database system.

The essential requirement for a database lock is that it should be small and fast enough to enable fine-grained concurrency for all data structures. We conclude that a 64-bit atomic integer is enough to implement a read-write lock that can protect all data structures we have in a DBMS. When protecting hierarchical data structures like trees, we recommend using an additional 64-bit field storing the lock's version to enable fast optimistic lock coupling without suffering from read-read contention. Our experiments showed that this setup is already optimal in the case of no contention.

With lock contention, new challenges arise. The first question is whether to spin or use a blocking approach. In this case, our experiments conclude that excessive spinning in user space is inherently dangerous and leads to problems such as heavy cache pollution, priority inversion in the scheduler, and waste of resources. We recommend relying on a kernel-supported blocking mechanism. Since the blocking locks of the standard library are very space-consuming and often slightly slower in the fast path, we recommend using a central infrastructure like a *parking lot* to add robust contention handling to every lock without additional in-place storage.

In the previous chapter, we already showed how a basic parking lot implementation works. We further extended the capabilities of a parking lot in this chapter and integrated appropriate fairness guarantees and cache topology awareness. Adding these features without a global parking lot would hardly be possible without hurting the uncontested fast path or increasing the local space consumption.

Our results indicate that the performance and versatility of a parking lot-based lock are ideal for a modern database system.

4

APPLYING OPTIMISTIC LOCKING IN AN MVCC DATABASE SYSTEM

Excerpts of this chapter have been published in [48].

4.1 MOTIVATION

After analyzing the different locks in the previous sections, we saw the superior performance of optimistic locking. We now show how optimistic locking can be applied to speed up an MVCC database system. As a baseline for our implementation, we use the MVCC database system HyPer [42, 78]. However, the techniques apply to any database system or tree-like index structure.

In this chapter, we revisit and rework the key components of a database system to make the entire system more scalable. First, we show how the tables can be synchronized optimistically allowing fast scans with hardly any overhead and efficient concurrent updates (Section 4.2). Second, we introduce optimistic index scans for optimal performance (Section 4.3). Then, after synchronizing the physical data storages of the database, we focus on the central MVCC logic, namely the transaction housekeeping. In Section 4.4, we adapt the management of MVCC transactions in HyPer to make it more scalable.

After enabling concurrent transactions, we suddenly are also more likely to run into serialization aborts caused by conflicting concurrent transactions. Therefore, we describe and discuss different strategies to handle high rates of aborts, i.e., logical contention in a transactional workload. We will briefly evaluate all changes in Section 4.6.

4.2 OPTIMISTIC SYNCHRONIZATION OF TABLES SCANS AND TUPLE ACCESSES

To show how we can best apply optimistic locking to tables, we first revisit the principle of MVCC and its implementation. MVCC is a well-known con-

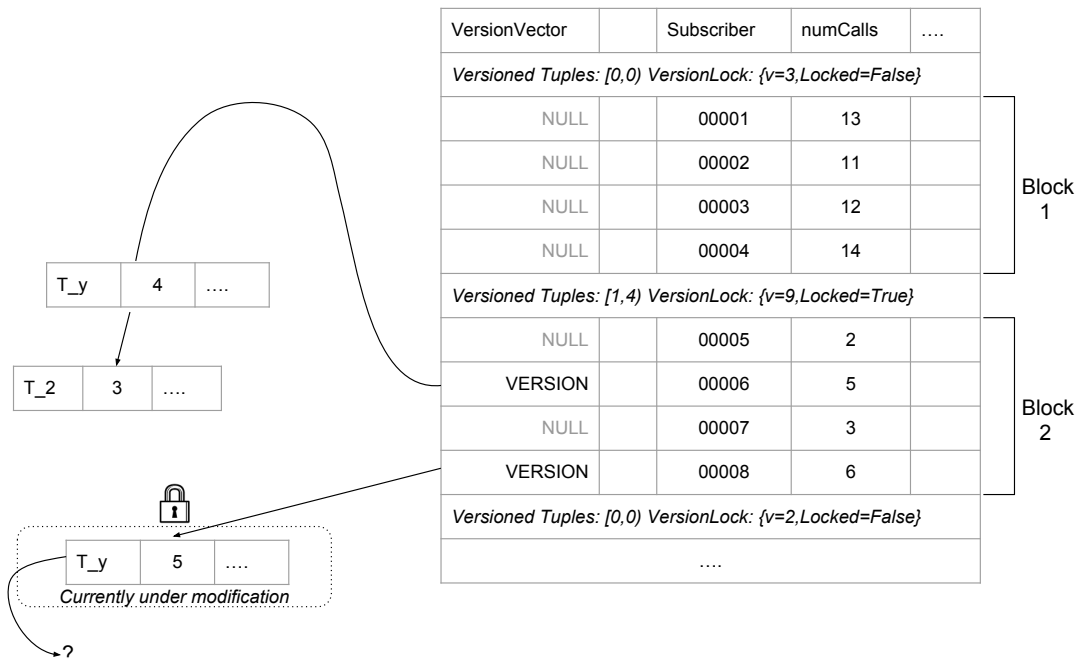


Figure 25: Adding optimistic locking to MVCC in HyPer

currency control technique used by many popular systems such as PostgreSQL or Oracle [109]. Figure 25 shows a table layout with MVCC. Every tuple can have a chain of versions stored in the VersionVector. If no version exists, the version pointer is null. Otherwise, transactions must retrieve the latest visible version¹.

In the following, we want to introduce an MVCC-specific optimization technique that was first introduced in HyPer [78]. The original purpose of this optimization is to optimize the read performance of scans. However, it is also an excellent fit to combine it with optimistic locking.

For faster scans, the tuples are grouped in blocks. For every block, the system maintains a summary, i.e., a range of the potentially versioned tuples². In Block 1, there is no versioned tuple, which is indicated by an empty interval ([0,0]). The version interval is encoded at the beginning of a block and allows the readers to process the entire block at full speed without checking for any versions. In Block 2, the first versioned tuple is at offset 1, and the last is at offset 3. Hence, the interval is [1,4). Only for those tuples that are within this interval, a reader must check for versions.

- ¹ The version of a tuple is visible to a reader when it was committed before the read transaction has started, i.e., its timestamp is smaller or equal to the start timestamp of the transaction
- ² A versioned tuple is a tuple with multiple versions. In contrast, an unversioned tuple exists in a single version that is visible to all transactions.

For optimal read performance, it is essential to keep the number of versions as low as possible. A version should be garbage collected as soon as it is no longer required. Garbage collection can either happen on a tuple- or transaction-level [106]. HyPer, for instance, implements the latter approach and removes all versions of a transaction once its commit timestamp is smaller than the start timestamp of all active transactions. Thus, the number of active versions highly depends on the transaction characteristics. Long-running transactions like complex OLAP queries generally lead to more active versions.

The ratio of versioned tuples can be estimated by considering the time of the longest-running transaction and the update rate. Assuming the slowest transaction takes 1 s and the database is updated at a rate of 10,000 TX/s.³ Until such a long-running query was completed, about 10,000 versions would accumulate. Thus, at most, 10,000 tuples are versioned at the same time. If the entire state contains 10 million tuples, that would translate to a ratio of 0.01 % of versioned data. Neumann et al. show that a ratio like this almost has no impact on scan performance [78]. They also argue that realistic scenarios have significantly lower update ratios.

We have prototyped this approach by implementing it in HyPer. Throughout the remainder of the chapter, we will refer to this implementation as HyPer-Parallel. However, the described optimizations and techniques apply to every system that uses MVCC. In fact, most MMDBs use MVCC and thus could benefit from this approach.

For optimistic locking, we extend MVCC by adding a VersionLock to protect each block. A VersionLock consists of a 64-bit version counter that encodes the lock state of the block in its highest bit.

When a reader processes a block, it retrieves its version in the beginning and validates it at the end to ensure that it has read a consistent state. If a concurrent writer has altered the block, the reader must process the block again. In cases when the block is already locked, the reader waits until it is released again. Using C++ templates and lambdas, this can be implemented by wrapping the existent scan code of a block into a lambda and pass it as a template argument to the VersionLock as shown in Listing 8. This optimistic latching comes at almost no additional cost for the reader since it only needs to check the version counter twice for every block, i.e., every 1024 tuples.

However, since we only latch optimistically, we can still experience race conditions when following the version chains. A version could be garbage col-

³ The numbers are taken from the update-heavy AIM workload that we discuss in [48].

Algorithm 8: Passing existing scan code as a lambda to VersionLock

```

1 <template typename Callback>
2 void readOperation(const Callback& scanBlock) {
3 restart:
4   uint64_t vStart = getVersion();
5   if (isLocked(vStart))
6     // Block is currently locked by a writer.
7     goto restart;
8   scanBlock();
9   uint64_t vEnd = getVersion();
10  if (vStart != vEnd) {
11    // A concurrent writer has altered the block.
12    goto restart;
13  }
14 }

```

lected while a reader tries to dereference its pointer. Therefore, we protect the access of a version chain by setting a lock bit in the head pointer of the version chain. In Figure 25, subscriber 00008 is currently modified and thus its version chain is locked.

If the head pointer points to a version chain, we set its lock bit in a single store instruction. This is cheap compared to following the version chain, which results in random memory accesses. The cost of acquiring the lock while accessing a versioned tuple is hidden by the memory latencies that occur while loading the version. Fortunately, as shown above, the percentage of versioned tuples is negligibly small and thus, locking tuples only happens in rare cases.

4.3 OPTIMISTIC SYNCHRONIZATION OF INDEX STRUCTURES

Besides the raw storage layer of a table (in our case, a column store), the index structures need to be synchronized. We synchronized HyPer’s primary index structure, the Adaptive Radix Trie (ART), using optimistic lock coupling [58, 59]. This technique synchronizes concurrent inserts, deletions, and lookups optimistically. Like the blocks in the table, every node gets a version lock. When the version of a node was modified, the operation restarts from the root. Again, we need some precautions regarding memory accesses since we

only lock optimistically. Thus, we cannot release memory immediately when removing a node, as it might still, be used by concurrent operations. We use epoch-based memory management that keeps nodes intact as long as other operations might access them. Nodes are marked as obsolete and are only deleted when all active operations left the current epoch.

While point lookups such as inserts, key lookups, and deletes can be synchronized efficiently using (optimistic) lock coupling [59], range scans need some more precautions and considerations. In a range scan, we traverse the tree until we find the first qualifying element on the left side. Then we “produce” all matching elements to our query pipeline. With pessimistic lock-coupling, we can guarantee that all produced output tuples are valid. When we only lock the nodes optimistically, we must buffer the elements first, as we might have read inconsistent data. Only when the optimistic validation has succeeded can we produce all output tuples. When the validation fails, we must discard all read elements and restart the scan. To avoid starvation, we perform range lookups in chunks of up to 1024 tuples as shown in Figure 26 and Listing 9. Each scan chunk stops when it finds the last qualifying element or when the output buffer is full. In the latter case, the iterator stores the key of the last read element for the next call.

A valid alternative would be to use hybrid locks (cf. Section 2.2.5): especially for B Trees with larger nodes as HyPer’s ART. Traversing to the leaves can be done efficiently using optimistic lock coupling. Only when reaching and reading the leaf nodes would the system acquire the node’s lock shared. At this point, read-read contention is unlikely or can be neglected compared to the cost of scanning an entire page. Query processing itself would also become easier as all read tuples can be directly pushed into the data pipelines without having to deal with restarts. When the tuples were read pessimistically, one can safely push them into query pipelines. With optimistic locking, one must buffer them first, as the read data might become invalid and should not end up in the query pipelines, as reverting this is generally not possible.

4.4 SCALABLE TRANSACTION MANAGEMENT

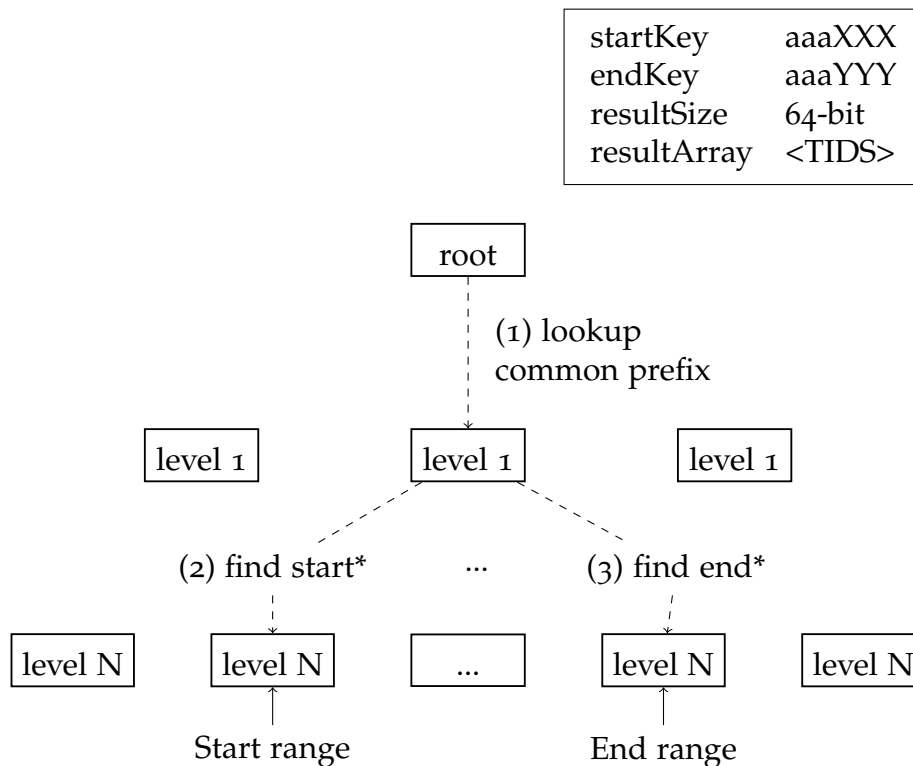
Finally, we revisit the design of transaction management in MVCC. In general, transactions can be categorized into two lists: one for the active, currently running transactions, and one for the committed transactions. A committed

Algorithm 9: Optimistic range scans in trees

```

1  struct FindStart {
2      byte* startKey
3      uint64_t keyLength
4      bool& needRestart
5      CopyHelper& copy
6  };
7
8  void FindStart::findStart(node, parent, pVersion, level) {
9      if (node->isLeaf())
10         if (node->key >= startKey)
11             copy.leaf2Results(node)
12         return
13
14         // Lock coupling
15         v = node->readLock(needRestart)
16         if (needRestart)
17             return
18         parent->unlockRead(pVersion, needRestart)
19         if (needRestart)
20             return
21
22         // Extract (copy!) all matching (child,key) pairs
23         startLevel = (keyLength > level) ? key[level] : 0
24         childrenKeys = node->getChildrenKeyPairs(startLevel, v, needRestart)
25         if (needRestart)
26             return
27
28         // Recurse
29         foreach ( (child,key) in childrenKeys) {
30             if (key == startKey)
31                 findStart(child, node, v, level + 1)
32             else
33                 copy.recursively2Results(child)
34             if (needRestart)
35                 return
36         }
37         // Validate read data (version)
38         node->unlockRead(v, needRestart);
39 }

```



*recurse down and copy values of matching keys to result array

Figure 26: Optimistic Range Scans – When the validation of a node fails, the scan is restarted. To avoid frequent restarts, the maximum number of retrieved results is limited to 1024. If there are more matching tuples, the scan has to be resumed with the last matching key as the new startKey.

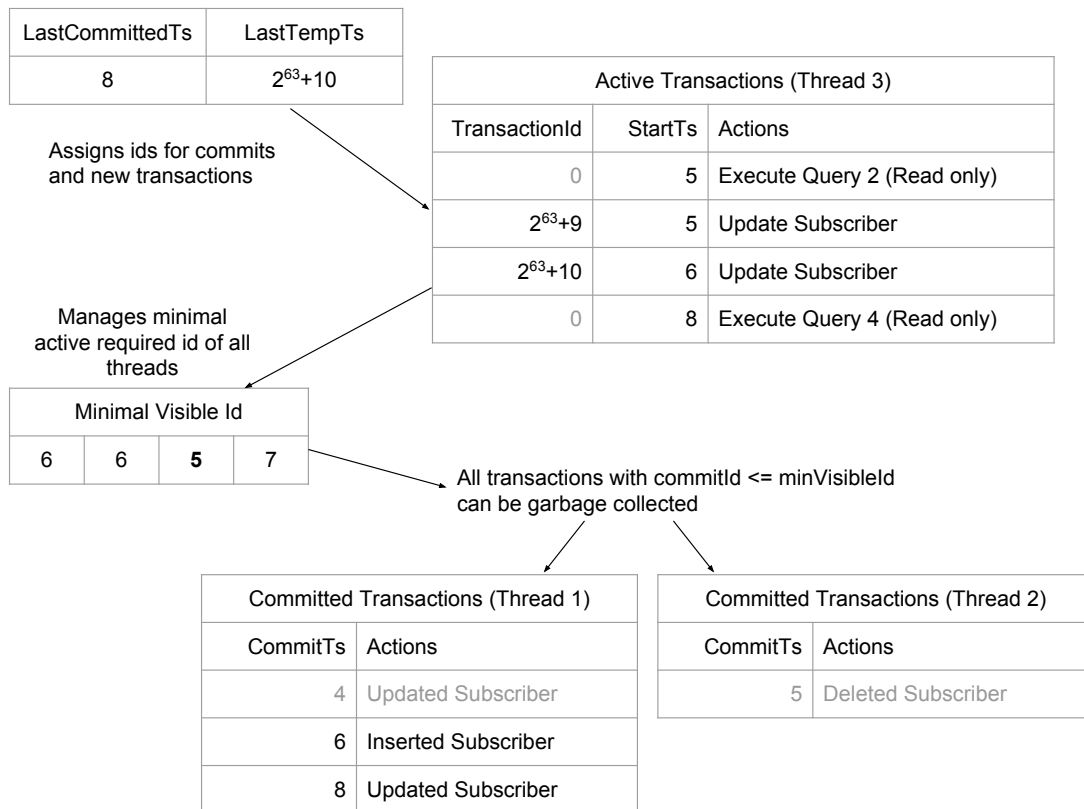


Figure 27: Using thread local lists to manage and garbage collect transactions

transaction can be discarded when its versions are no longer required by any active transaction, i.e., there is no active transaction that had started before the transaction was committed.

Synchronizing linked lists requires the use of global locks, which do not scale well for a large number of concurrent transactions. Thus, every thread manages lists of active and committed transactions in our implementation. Figure 27 shows an example of these lists.

To decide which of the committed transactions can be garbage collected, we need to know the smallest start timestamp of all active transactions. We solve this by maintaining a global array that contains the minimal start timestamp for each thread. We can garbage collect all transactions that were committed before the oldest transaction started.

More components that need synchronization are the two 64-bit MVCC counters: One starts from 0 to assign commit-timestamps, and one starts from 2^{63} . By starting from 2^{63} , we guarantee that temporary, uncommitted versions are not visible to any active transaction that starts at the latest commit timestamp. In our current implementation, we made these counters atomic. A global

counter may be a bottleneck under high throughput. However, in workloads such as Huawei-AIM, transactions are expensive enough such that there is no significant contention on the global counter. Using a global counter for transaction management is a common practice even in high-performance key-value stores [87].

For better performance, we only access the counters lazily, i.e., right before we start to write a new version. Read-only transactions will never update the counters. In future work, we plan to implement more sophisticated techniques to handle the MVCC timestamps [62, 109].

4.5 HANDLING LOGICAL CONTENTION AND DATA SKEW

The performance of MMDBs can collapse when multiple transactions contend on the same data [75]. To ensure consistency, transactions are executed (almost) serially using locks and aborts. This leads to idling cores and poor performance. Competing threads constantly invalidate cached data. In particular, optimistic concurrency control systems like MVCC are vulnerable to contention since conflicting transactions are aborted [62]. The changes made by aborted transactions have to be rolled back, which further increases the level of contention on the data.

One way to avoid physical contention is to assign threads exclusively to disjoint data partitions. Thereby, threads do not interfere with each other, and there is no physical contention on the data. This approach is used by both Flink and the hand-crafted C++ implementation of the Huawei-AIM workload. For this workload, this is a reasonable solution since the data is perfectly partitionable, and there are only single-row updates. In general, however, partitioning has two significant drawbacks: Finding a good partitioning can be difficult, and executing cross-partition transactions introduces a significant overhead [75].

Like many MMDBs, HyPer follows the more general-purpose approach of not partitioning data. Thereby, it supports workloads that cannot be easily partitioned. The managed state is globally consistent and not only per partition. This allows for multi-row transactions, which are limited or impossible in partitioned systems. With MVCC and optimistic locking, multi-row transactions can benefit performance due to caching effects.

A drawback of non-partitioning MMDBs is their vulnerability to skewed workloads. In contrast to partitioned (streaming) systems, threads do not exclusively own parts of the table. Thus, they all compete for the same contended data and locks. This can lead to several aborts due to serialization errors and dramatically decreased performance. Without regulation, the performance might drop below the single-threaded throughput due to the high number of aborts.

A common approach to tackle contention is to back off the number of active threads for a random duration, i.e., send them to sleep. The maximum backoff time should be chosen adaptively to the workload. We implemented a backoff strategy similar to the Cicada system that uses hill climbing to determine the optimal backoff time depending on change in throughput [62].

4.6 PERFORMANCE EVALUATION

This section evaluates the performance of integrating optimistic locking into HyPer. We keep this evaluation brief as we already evaluated optimistic locking in database workloads in Section 2.4. Additionally, we already studied the OLTP throughput of *HyPerParallel* extensively in our comparison with dedicated streaming systems and would like to point the reader to this paper [48].

This section will focus on two new aspects. First, we analyze the handling of conflicting transactions. And second, now that we have discussed the implementation of optimistic range scans in Section 4.3, we run TPC-H with (primary key) indexes enabled to see its effect compared to pessimistic locks.

We ran the experiments on an Ubuntu 17.04 machine with an Intel Xeon E5-2660 v2 CPU (2.20GHz, 3.00GHz maximum turbo boost) and 256GB DDR3 RAM. The machine has two NUMA sockets with 10 physical cores (20 hyper-threads) each, resulting in 20 physical cores (40 hyper-threads). The sockets communicate using a high-speed QPI interconnect (16GB/s).

4.6.1 Read Scalability of Optimistic Locking

After analyzing the write scalability, we now look at the read performance of optimistic locking compared to pessimistic locking in Figure 28. We compare optimistic locking to a spinlock (`tbb::spin_rw_mutex`) and an OS-supported `std::shared_mutex`. Therefore, we replace all table and index locks in HyPer

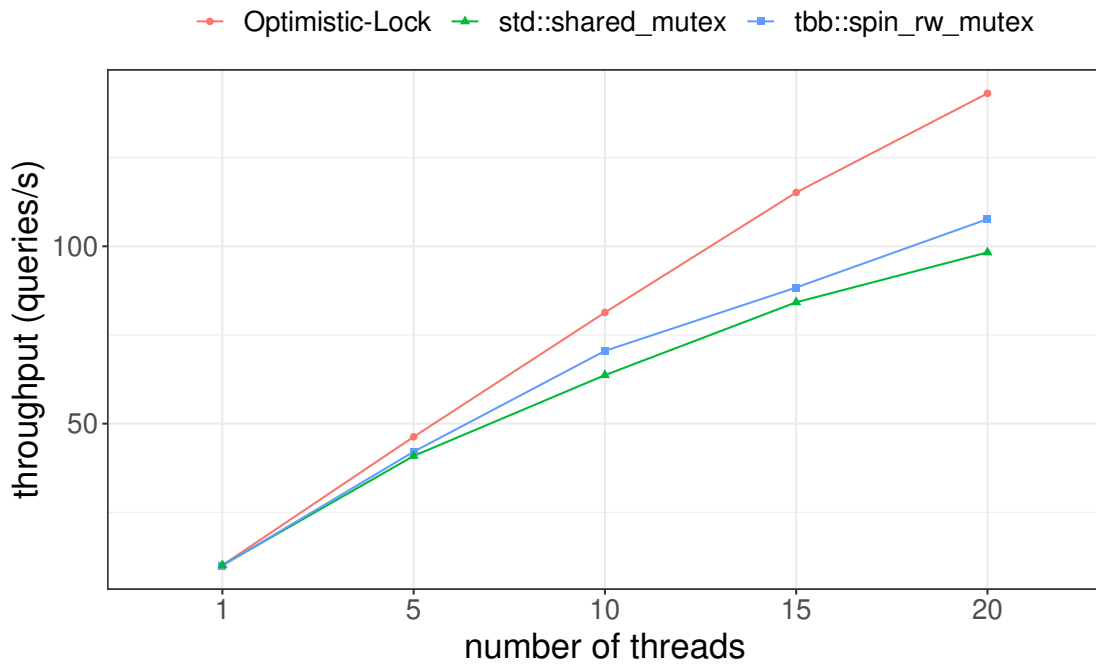


Figure 28: TPC-H SF10 – *The scalability of the pessimistic latches suffers from read-read contention in the used index structures*

with the respective lock implementation. We run TPC-H scale factor 10 with an increasing number of analytical threads. Optimistic locking scales almost perfectly, while the other locks suffer from “read-read contention” in the primary key indexes. As optimistic locking never does atomic writes during reading, it does not create any cache contention during reading.

4.6.2 Handling of Skew and Serialization Conflicts

In this experiment, we study the effect of skew on the event processing performance of the different systems. Therefore, we use the Huawei-AIM workload, which maintains call statistics of users [8]. It maintains a table of 10 million users with up to 546 aggregates (columns) per user. Every call event updates the stats of a single user. Thus, there is only a conflict if two transactions try to update the same customer concurrently.

When following a uniform distribution, there should hardly be any conflicts. However, we presume that, in reality, the events are skewed (i.e., there will be people that make significantly more calls than others). Therefore, we test the skew handling capabilities of our system with events following different Zipfian distributions.

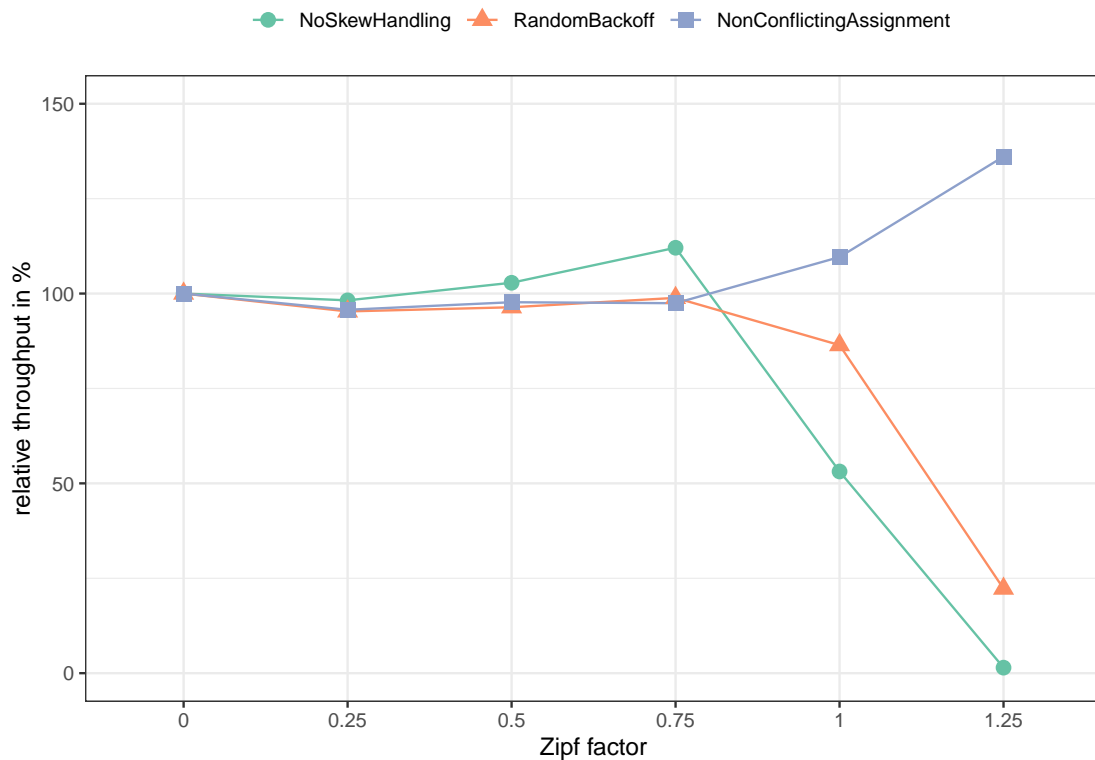


Figure 29: Skew handling in HyPer – Throughput of skew handling techniques using 10 threads with increasing skew

Figure 29 shows the throughput for different skew handling techniques with increasing skew relative to their performance under a uniformly distributed workload. With NoSkewHandling, HyPerParallel becomes vulnerable to serializability issues. If two transactions try to update the same subscriber concurrently, only one can succeed, and the other one gets aborted by the optimistic concurrency control. With increasing skew, this leads to an increasing number of aborts due to conflicts. Our experiments reached this point at a Zipf factor of 1.0. Without skew handling, the throughput is almost halved. The performance drops significantly at the highest skew level (Zipf factor of 1.25) due to the high number of aborts. On average, each transaction has to be executed three times until it succeeds.

To reduce the high number of aborts, we have added a random backoff to the event processing (cf. RandomBackoff in Section 4.5). The maximum backoff time is adapted based on the current throughput. For a Zipf factor of 1.25, this approach reduces the number of aborts by a factor of four and increases the throughput by more than $15\times$. This suggests that the number of aborts significantly impacts the throughput.

For a deeper investigation of this hypothesis, we designed a third contention handler (`NonConflictingAssignment`) that guarantees execution without any logical conflicts that would require aborts. A conflict can only occur when two transactions try to update the same record. By using a fixed (modulo-based) assignment from subscribers to threads, we ensure that we will never end up in a situation where two or more threads try to update the same subscriber. Note that there still might be physical contention on the indexes and locks since we do not partition the underlying data. For instance, if a thread updates Subscriber 1 and another thread updates Subscriber 2, there is no logical (serialization) conflict. However, both threads might compete for the same latches as the subscribers reside in the same physical block (cf. Section 4.2). The results verify our hypothesis that aborting transactions is expensive: `NonConflictingAssignment` achieves a speedup of 136% for the highest contention level. Physical contention, such as latching or accessing index structures, turns out not to be an issue and instead leads to beneficial caching effects.

Despite the superior performance of `NonConflictingAssignment`, we would recommend implementing it along with a `RandomBackoff` mechanism. This gives the system more flexibility for other workloads that might not allow `NonConflictingAssignment`. For instance, in cases where there are more complex transactions that update multiple rows. The performance of `NonConflictingAssignment` is not affected by adding `RandomBackoff` since `RandomBackoff` only “backs off” when there is a conflict.

4.7 SUMMARY

This chapter showed how optimistic locking could be integrated into an MVCC database system. We covered the different scalability choke points when transforming HyPer into a scalable OLTP system. We showed how table scans, tuple accesses, and index scans could be optimistically synchronized. Then, we redesigned the transaction management of HyPer to make it scalable for concurrent transactions.

Those steps are crucial when making an MVCC database system ready for high real-time update rates without sacrificing the read-only performance. All physical data structures are now prepared for mixed read and write accesses. However, running full HTAP workloads still poses new challenges to the system because of the inherent problems of MVCC with mixed workloads that

contain relatively long-running transactions. The next chapter focuses on this challenge, mainly on handling MVCC versions efficiently in the presence of long-running queries.

5

SCALABLE GARBAGE COLLECTION FOR IN-MEMORY MVCC SYSTEMS

Excerpts of this chapter have been published in [7].

To support Hybrid Transaction and Analytical Processing (HTAP), database systems generally rely on Multi-Version Concurrency Control (MVCC). While MVCC elegantly enables lightweight isolation of readers and writers, it also generates outdated tuple versions, which, eventually, have to be reclaimed. Surprisingly, we have found that in HTAP workloads, this reclamation of old versions, i.e., garbage collection, often becomes the performance bottleneck.

It turns out that in the presence of long-running queries, state-of-the-art garbage collectors are too coarse-grained. As a consequence, the number of versions grows quickly slowing down the entire system. Moreover, the standard background cleaning approach makes the system vulnerable to sudden spikes in workloads.

In this chapter, we propose a novel garbage collection (GC) approach that prunes obsolete versions eagerly. Its seamless integration into the transaction processing keeps the GC overhead minimal and ensures good scalability. We show that our approach handles mixed workloads well and also speeds up pure OLTP workloads like TPC-C compared to existing state-of-the-art approaches.

5.1 MOTIVATION

Multi-Version Concurrency Control (MVCC) is the most common concurrency control mechanism in database systems. Depending on the implementation, it guarantees snapshot isolation or full serializability if complemented with precision locking [78]. MVCC has become the default for many commercial systems such as MemSQL [72], MySQL [74], Microsoft SQL Server [96], Hekaton [51], NuoDB [79], PostgreSQL [89], SAP HANA [25], and Oracle [81] and state-of-the-art research systems like HyPer [42] and Peloton [86].

The core idea of MVCC is simple yet powerful: whenever a tuple is updated, its previous version is kept alive by the system. Thereby, transactions can work

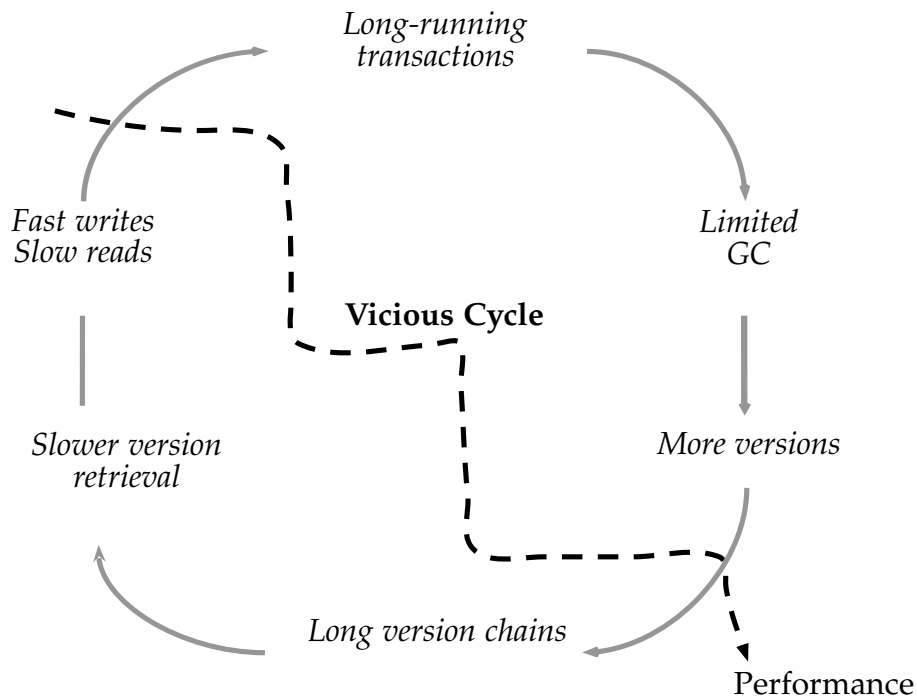


Figure 30: MVCC’s vicious cycle of garbage – Old versions cannot be garbage collected as long as there are long-running transactions that have to retrieve them

on a consistent snapshot of the data without blocking others. In contrast to other concurrency control protocols, readers can access older snapshots of the tuple, while writers are creating new versions. Although multi-versioning itself is non-blocking and scalable, it has inherent problems in mixed workloads. If there are many updates in the presence of long-running transactions, the number of active versions grows quickly. No version can be discarded as long as it might be needed by an active transaction.

For this reason, long-running transactions can lead to a “vicious cycle” as depicted in Figure Figure 30. During the lifetime of a transaction, newly-added versions cannot be garbage collected. The number of active versions accumulates and leads to long version chains. With increasing chain lengths, it becomes more expensive to retrieve the required versions. Version retrievals slow down long-running transactions further, which amplifies the effects even more. Write transactions are initially hardly affected by longer version chains as they do not have to traverse the entire chain. They only add new versions to the beginning of the chain. Thereby, the gap between fast write transactions and slow read transactions increases, quickly producing more and more versions. At some point, the write performance is also affected by the increasing contention on the version chains as the insertion of new versions is blocked

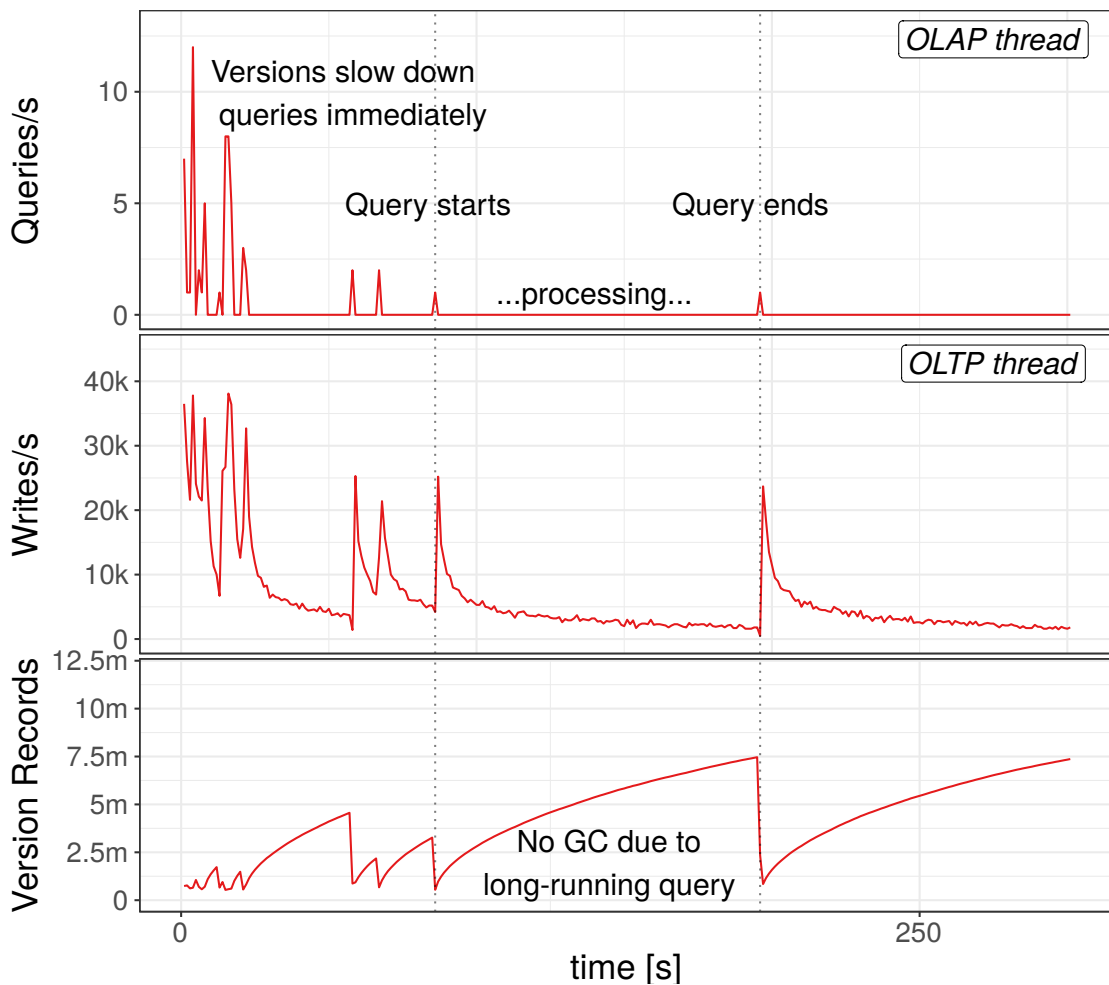


Figure 31: Practical Impacts – *The system’s performance drops within minutes in a mixed workload using a standard garbage collection strategy*

while the chain is latched for GC. The system also loses processing time for transactions when the threads clean the versions in the foreground.

In Figure Figure 31 we visualize the practical implications of the described “vicious cycle” by monitoring an MVCC system in the mixed CH benchmark¹. The OLTP thread continuously runs short-lived TPC-C style transactions, while the OLAP thread issues analytical queries. We see that the read performance collapses within seconds, while the writes are slowed down by long periods of GC. With higher write volumes or more concurrent readers, the negative effects would be even more pronounced. However, even low-volume workloads can run into this problem as soon as GC is blocked by a very long-running transaction (e.g., by an interactive user transaction).

¹ Section 5.2.2 describes this experiment in more detail.

The fact that GC is a major practical problem, causing increased memory usage, contention, and CPU spikes, has been observed by others [30, 61]. Nevertheless, in comparison with the number of papers on MVCC protocols and implementations, there is little research on GC. Except for of SAP HANA [52] and Hekaton [51], most research papers discuss GC only cursorily.

In this chapter, we show that the garbage collector is a crucial component of an MVCC system. Its implementation can have a huge impact on the system’s overall performance as it affects the management of transactions. Thus, it is important for all classes of workloads—not only mixed, “garbage-heavy” workloads [48, 47]. Our experimental results emphasize the importance of GC in modern many-core database systems.

As a solution, we propose *Steam*—a lean and lock-free GC design that outperforms previous implementations. *Steam* prunes every version chain eagerly whenever it traverses one. It removes all versions that are not required by any active transaction but would be missed by the standard high watermark approach used by most systems.

The remainder of this chapter is organized as follows. Section Section 5.2 introduces basic version management and garbage collection in MVCC systems and challenges regarding mixed workloads and scalability. We then provide an in-depth survey of existing GCs and design decisions in Section Section 5.4. In Section Section 5.3, we propose our scalable and robust garbage collector *Steam* that decreases the vulnerability to long-running transactions. We present our experimental evaluation of *Steam* in comparison to different state-of-the-art GC implementations in Section Section 5.5. Lastly, we conclude with related work on HTAP workloads and garbage collection in Section Section 5.6.

5.2 VERSIONING IN MVCC

MVCC is a concurrency control protocol that “backs up” old versions of tuples, whenever tuples are modified. For every tuple, a transaction can retrieve the version that was valid when the transaction started. Thereby, all transactions can observe a consistent snapshot of the table.

The versions of a tuple are managed in an ordered chain of version records. Every version record contains the old version of the tuple and a timestamp indicating its visibility. Under snapshot isolation, a version is visible to a transaction if it was committed before its start. Hence, the timestamp equals the

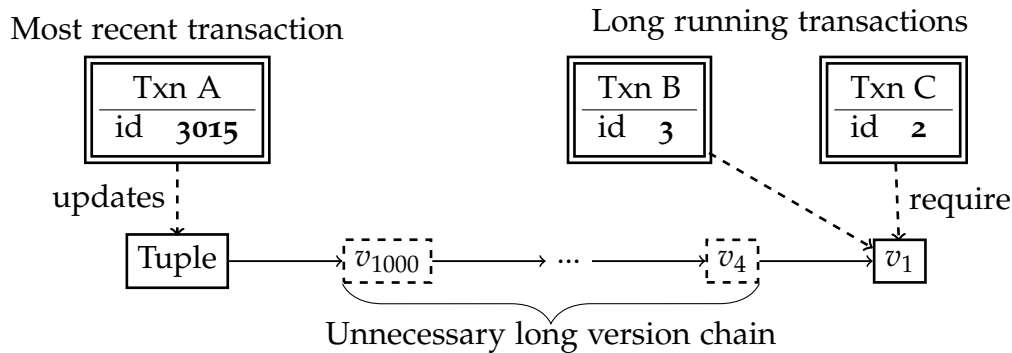


Figure 32: Long version chain – Containing many unnecessary versions that are not GC'ed by traditional approaches

transaction's commit timestamp or a high temporary number, if it is still in-flight [78].

MVCC can maintain multiple versions (snapshots) of a tuple, whereas every update adds a new version record to the chain. The chain is ordered by the timestamp to facilitate the retrieval of visible versions.

Figure Figure 32 shows a version chain for a tuple that was updated multiple times. Since Transaction B and C started before v_4 was committed, they have to traverse the chain (to the very end in this case) to retrieve the visible version v_1 .

5.2.1 Identifying Obsolete Versions

Before discussing efficient garbage collection, we revisit when it is safe to remove a version. In general, a version must be preserved as long as an active transaction requires it to observe a consistent snapshot of the database. Essentially this means, that all versions that are visible to an active transaction must be kept. It does not matter whether the versions will be actually retrieved since the database system generally cannot predict the accessed tuples of a transaction—especially in the case of interactive user queries. Therefore, it always has to keep the visible versions as long as they could be accessed in future.

The set of visible versions is determined by the currently active transactions. When a version is no longer needed by any active transaction, it can be removed safely. Future transactions will not need them because they will already work on newer snapshots of the database. Hence, the required lifetime of every version only depends on the currently active transactions.

In the best case, a garbage collector can identify and remove all unnecessary versions. Looking at Figure 32: version record v_1 must not be garbage collected because it is required by Transactions B and C. All the preceding version records could be garbage collected safely and the length of the chain could be reduced significantly from 1000 to only 1 version. However, traditional garbage collectors only keep track of the start timestamp of the oldest active transaction. Thereby, they only get a crude estimation of the reclaimable version records. Essentially, only the versions that were committed before the start of the oldest active transaction are identified as obsolete. This leads to several “missed” versions in the case of multiple updates and long-running transactions. To overcome this problem, we propose a more fine-grained approach in Section 5.3.3 that prunes the unnecessary in-between versions.

5.2.2 Practical Impacts of GC

Figure 31 demonstrates the practical weaknesses of a standard GC. For this experiment, we ran the mixed CH benchmark which combines the transactional TPC-C and analytical TPC-H workload [11]. One OLAP and OLTP thread are enough to overstrain the capabilities of a traditional high watermark GC. Having only one warehouse, the isolated query execution times are reasonably fast (5-500 ms). However, compared to the duration of a write (0.02 ms), some of the queries are already long-running enough to run into the “vicious cycle”. By adding more threads and/or warehouses the effects would be even worse.

The query throughput drops significantly after some seconds and queries start to last seconds (instead of milliseconds as before). These long-running queries show up in the topmost plot as the increasing periods of 0 queries/s. As long as the query is running, the number of version records stack up. This leads to the “shark fin” appearance in the number of version records. Only when the reader is completed, the writer starts to clean up the version records. For these periods of GC, it cannot achieve any additional write progress. Over time, the effects get worse and the amplitude of the number of version records increases while the read and write performance drops to almost 0. The query latencies increase significantly by the additional version retrieval work while the write processing suffers from the additional contention caused by the GC. In this setup—with only one write thread—the back pressure on the GC thread is already too high and the number of versions grows constantly. Especially

the effects on the read performance are tremendous if the GC thread cannot catch up with the write thread(s). At some point, the entire system would run out of memory.

In summary, traditional garbage collectors have several fundamental limitations: (1) scalability due to global synchronization, (2) vulnerability to long-living transactions caused by its (3) inaccuracy in garbage identification. The general high watermark approach cannot clean in-between versions long version chains.

5.3 STEAM GARBAGE COLLECTION

Garbage collection of versions is inherently important in an MVCC system as it keeps the memory footprint low and reduces the number of expensive version retrievals. In this section, we propose an efficient and robust solution for garbage collection in MVCC systems. We target three main areas: scalability (\rightarrow 5.3.2), long-running transactions (\rightarrow 5.3.3), and memory-efficient design (\rightarrow 5.3.4).

5.3.1 Basic Design

Steam builds on HyPer’s MVCC implementation and extends it to become more robust and scalable [78]. To keep track of the active and committed transactions, HyPer uses two linked lists as sketched in Figure 33.

While HANA and Hekaton use different data structures (a reference-counted list and a map), the high-level properties are the same. All implementations implicitly keep the transactions ordered and adding or removing of a transaction can be done in constant time. To start a new transaction, the system appends it to the *active transactions* list. When an active transaction commits, the system moves it to the *committed transactions* list to preserve the versions it created. Completed read-only transactions, that did not create any tuple versions, are discarded directly.

By appending new or committed transactions to the lists, the transaction lists are implicitly ordered by their timestamps. This ordering allows one to retrieve the minimum *startTs* efficiently by looking at the first element of the *active transactions* list. The versions of a *committed transaction* with $commitId \leq \min(startTs)$ can be reclaimed safely. Since the *committed transaction* list is also

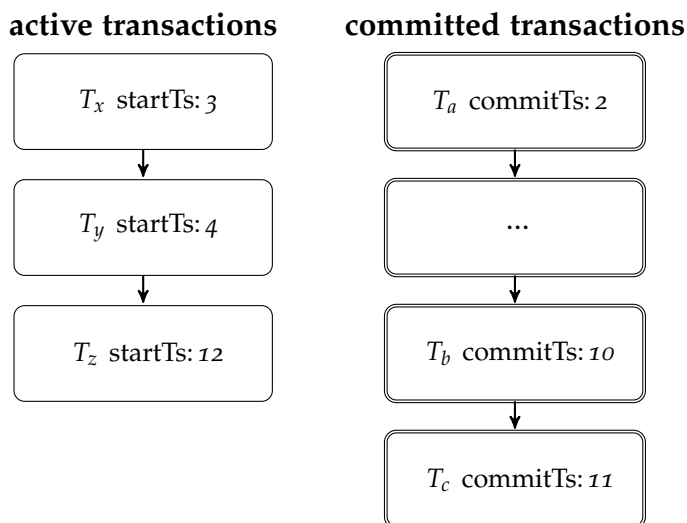


Figure 33: Transaction lists – Ordered for fast GC

ordered, the system can reclaim all transactions until it hits a transaction that was committed after the oldest active transaction.

5.3.2 Scalable Synchronization

While the previously described basic design offers constant access times for GC operations, its scalability is limited by the global transaction lists: Both lists need to be protected by a global mutex. For scalability reasons, we aim to avoid data structures that introduce global contention. Hekaton avoids a global mutex by using a latch-free transaction map for this problem. Steam, in contrast, follows the paradigm that it is best to use algorithms that do not require synchronization at all [22]. For GC, we exploit the domain-specific fact that the correctness is not affected by keeping versions slightly longer than necessary—the versions can still be reclaimed in the “next round” [30]. Steam’s implementation does not require any synchronized communication at all. Instead of using global lists, every thread in Steam manages a disjoint subset of transactions. A thread only shares the information about its thread-local minimum globally by exposing it using an atomic 64-bit integer. This thread-local *startTs* can be read by other threads to determine the global minimum.

The local minimum always corresponds to the first active transaction. If there is no active transaction, it is set to the highest possible value ($2^{64} - 1$). In Figure 34 the local minimums are 4, 3, and, 12. To determine the global minimum for GC, every thread scans the local minimums of the other threads.

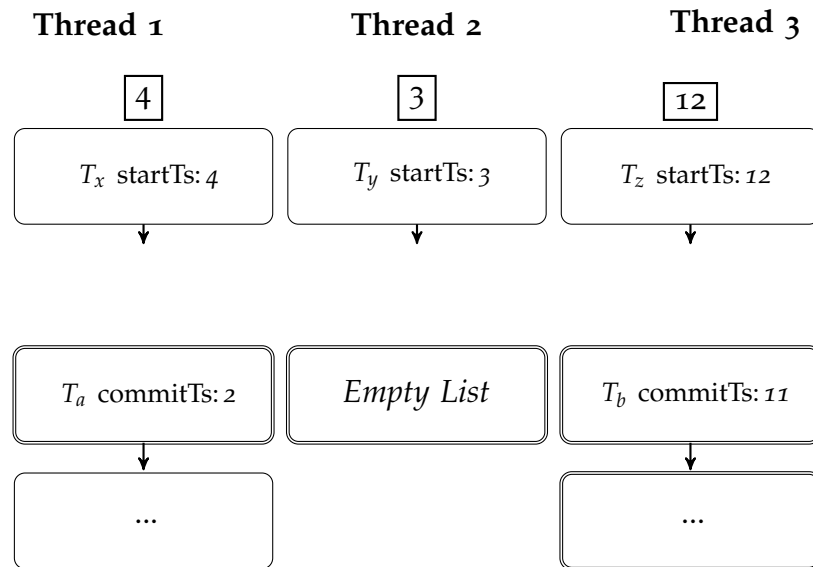


Figure 34: Thread-local design – Each thread manages a subset of the transactions

Although this design does not require any latching, the global minimum can still be determined in $O(\#threads)$. Updating the thread-local minimum does not introduce any write contention either since every thread updates only its own *minStartTs*.

Managing all transactions in thread-local data structures reduces contention. On the downside, this can lead to problems when a thread becomes inactive due to a lack of work. Since every thread cleans its obsolete versions during transaction processing, GC can be delayed if the thread becomes idle. To avoid this problem, the scheduler periodically checks if threads have become inactive and triggers GC if necessary.

5.3.3 Eager Pruning of Obsolete Versions

During initial testing, we noticed significant performance degradations in mixed workloads. Slow OLAP queries block the collection of garbage because the global minimum is not advanced as long as a long-running query is active. Depending on the complexity of the analytical query, this can pause GC for a long time. With concurrent update transactions, the number of versions goes up quickly over the lifetime of a query. This can easily lead to the vicious cycle as described in Section 5.1. In practice, this effect can be amplified further by skewed updates which leads to even longer version chains.

Figure 32 shows how the versions of a tuple can form a long chain in which the majority of versions is useless for the active transactions. The useless versions slow down the long-running transactions when they have to traverse the entire chain to retrieve the required versions in the end. For this reason, we designed *Eager Pruning of Obsolete Versions* (EPO) that removes all versions that are not required by any active transaction. To identify obsolete versions, every thread periodically retrieves the start timestamps of the currently active transactions and stores them in a sorted list. The active timestamps are fetched efficiently without additional synchronization as described later in Section 5.3.3. Throughout the transaction processing, the thread identifies and removes all versions that are not required by any of the currently active transactions. Whenever a thread touches a version chain, it applies the following algorithm to prune all obsolete versions:

```

input : active timestamps A (sorted)
output: pruned version chain
1  $v_{current} \leftarrow \text{getFirstVersion}(\text{chain})$ 
2 for  $a_i$  in A do
3    $v_{visible} \leftarrow \text{getNextVisibleVersion}(a_i, v_{current})$ 
4   // prune obsolete in-between versions
5   for  $v$  in  $(v_{current}, v_{visible})$  do
7     if  $\text{attrs}(v) \not\subseteq \text{attrs}(v_{visible})$  then
8       |  $\text{merge}(v, v_{visible})$ 
9     end
10     $\text{chain.remove}(v)$ 
11  end
12   $v_{current} \leftarrow v_{visible}$ 
13 end

```

Algorithm 1: Prune obsolete versions

We only store the changed attributes in the version record to save memory. For this reason, we have to check whether all of v 's attributes are covered by $v_{visible}$. If there are additional attributes, we merge them into the final version. Systems that store the entire tuple would not need this check and could discard the in-between versions directly.

Figure 35 shows the pruning of a version chain for one active transaction started at timestamp 20. It shows the relatively-simple case when all attributes are covered by $v_{visible}$ and the more complex case, when the in-between versions contain additional attributes. In this case, we add the missing versions to the final version. When an attribute is updated multiple times, we overwrite it

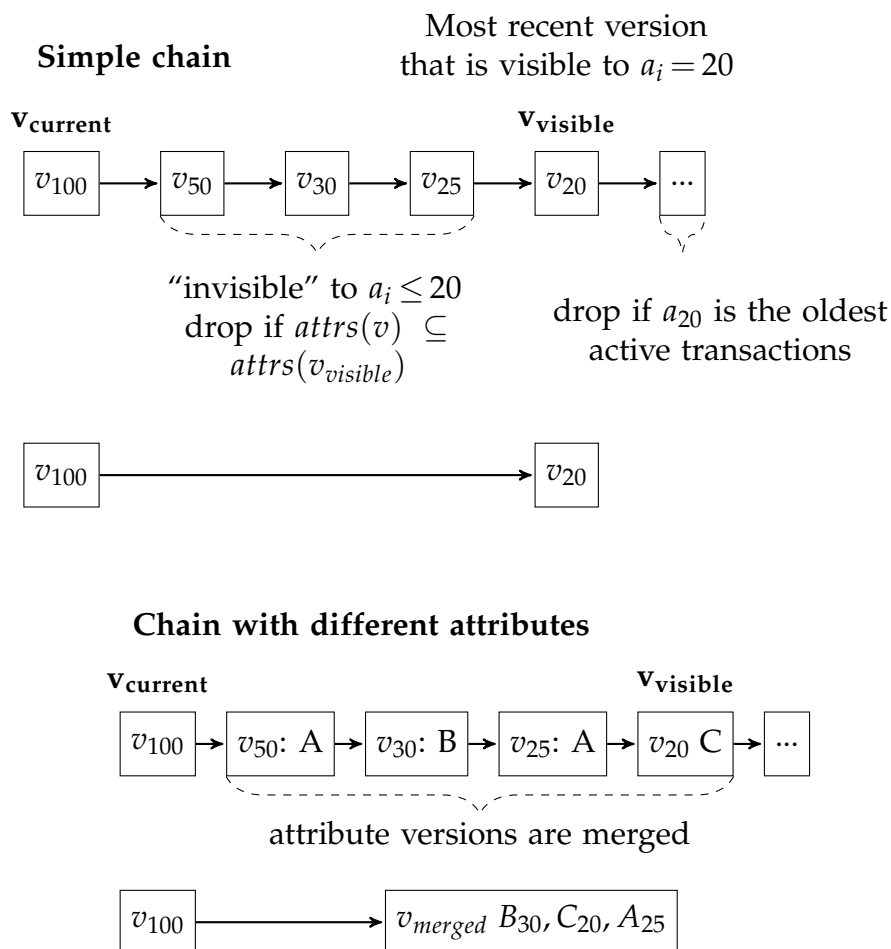


Figure 35: Prunable version chain – Example for an active transaction with id 20

when we find an older version of it while approaching the visible version v_{visible} . In our example, A_{50} is overwritten by A_{25} . After the pruning, v_{current} is set to the current value of v_{visible} and v_{visible} is advanced to the version that is visible to the next older (smaller) active id. As we only have one active transaction in our example, we can stop at this point.

Since the version chain and the active timestamps are sorted and duplicate-free, every version is only touched once by the algorithm.

Short-Lived Transactions

EPO is designed for mixed workloads in which some transactions (mostly OLAP queries) are significantly slower than others. If all transactions are equally fast, it does not help as the commit timestamps hardly diverge from the id of the oldest active transaction.

Table 7: Comparison with HANA’s Interval GC

HANA	Steam
Dedicated GC thread scans all committed versions lazily every 10 s causing additional version and latching	Every thread scans the accessed version chains eagerly “piggybacking” the costs while the chain is locked anyway

A standard GC using a global minimum already works perfectly fine here. Thus, creating a set of active transactions will hardly pay off, as the number of reducible version chains is small. Ideally, we can avoid the overhead of retrieving the current set of transaction timestamps.

However, in general, the characteristics of a workload cannot be known by the database system and change over time. So instead of turning EPO off, we reduce its overhead without compromising its effectiveness in mixed workloads.

The only measurable overhead of the approach is the creation of the sorted list of currently active transactions. The creation of the list only adds a few cycles to the processing of every transaction (for a system using 10 worker threads that are 10 load instructions² and sorting them) but it is still noticeable in high volume micro-benchmarks.

To reduce this overhead, every thread reuses its lists of active transactions if it is still reasonably up-to-date. Thereby, the costs are amortized over multiple short-lived transactions and the overhead becomes negligible. For transactions running for more than 1 ms the costs of fetching the active transaction timestamps become insignificantly small. The quality of EPO is not affected as the set of long-running transactions changes significantly less frequently than the active transactions lists are updated.

During micro-benchmarks with cheap key-value update transactions, we noticed that the update period can be set to as low as 5 ms without causing any measurable overhead. This update period is still significantly smaller than the lifetime of even “short long-running” transactions.

HANA’s Interval-Based GC

HANA’s interval GC builds on a similar technique to shorten unnecessary long version chains, yet it differs in important aspects, which are summarized

² We only schedule as many concurrent transactions as we have threads.

Table 8: Data Layout of Version Records

	Update	Delete	Insert	Bytes
<i>Common Header</i>				
Type	✓	✓	✓	1
Version	✓	✓	✓	4
RelationId	✓	✓	✓	2
<i>Additional Fields</i>				
Next Pointer	✓	✓	–	4
TupleId	✓	✓	–	4
NumTuples	–	–	✓	4
AttributeMask	✓	–	–	4
<i>Payload</i>				
BeforeImages	✓	–	–	<i>var</i>
Tuple Ids	–	–	✓	$8 \times t$
Total Bytes	$19 + var$	15	$11 + 8 \times t$	

in Table 7. The biggest difference is how the version chains are accessed for pruning. In Steam, the pruning happens during every update of a tuple, i.e., whenever the version chain is extended by a new version. Thereby, a chain will never grow to more versions than the current number of active transactions and will never contain obsolete versions.

In HANA, in contrast, the pruning is done by a dedicated background thread which is triggered only every 10 seconds. When HANA’s GC thread is triggered, it scans the set of versions that were committed after the start of the oldest active transaction. For each of these versions, it checks if it is obsolete within its corresponding version chain using a merge-based algorithm similar to ours. This causes additional chain accesses, whereas Steam can “piggyback” this work on normal processing. Since HANA calls the interval-based GC only periodically, the version chains are not pruned and grow until the GC is invoked again.

5.3.4 Layout of Version Records

The design a version record should be space and computationally efficient. All operations that involve versions (insert, update, delete, lookup, and roll-back) should work as efficiently as possible. Additionally, the layout should be in favor of GC itself, especially our algorithm for pruning intermediate versions.

Table 8 shows the basic layout of a version record. It has a *Type* (Insert/Update/Delete) and visibility information encoded in the *Version*. At commit time, the *Version* is set to the commit timestamp, which makes the version visible to all future transactions. To guarantee atomic commits, the *Version* includes a lock bit, which is used when a transaction commits multiple versions at the same time.

When a transaction is rolled back, it uses the *RelationId* and *TupleId* to identify and restore the tuples in the relation. The fields are also used during GC to identify the tuple that owns the version chain. The version chain itself is implemented as a linked list using the *Next Pointer* field. The *Next Pointer* either points to the next version record in the chain or *NULL* if there is none.

For all types of version records except for deletes, we need some additional fields or variations. For deletes, it is enough to store the timestamp when a tuple has become invisible due to its deletion.

For inserts, we adapt the data layout by reinterpreting the attributes *TupleId* and *Next Pointer* to maintain a list of inserted tuple ids. This allows us to handle bulk-inserts more efficiently because we can use a single version record for all inserted tuples of the same relation. Sharing insert version records decreases the memory footprint (previously every inserted tuple required an own version record) and improves the commit latency. We can now commit multiple versions atomically by updating only a single *Version*. This optimization is possible since new tuples can only be inserted into previously empty slots. Thus, we can reuse the *Next Pointer* field to maintain a list of inserted *Tuple Ids*. For MVCC, we only need the information when the inserted tuple becomes visible. The tuple id list can be further compressed for bulk-inserts by storing ranges of subsequent tuples.

Update version records require the most fields as they contain the tuple's previous version (*Before Images*). To save space, we only store the versions of the changed attributes instead of a full copy of the tuple. Therefore, the version record needs to explicitly indicate which attributes it contains. For all relations with less than 64 attributes, we therefore use a 64-bit *Attribute Mask*, where every changed attribute is marked by a bit. When the relation has more columns, we indicate the changed attributes using a list of the ids of all changed attributes.

While the *Attribute Mask* saves space compared to the list, it also allows us to perform the check if a version record is covered by another (cf. Algorithm line 7) using a single bitwise or-operation. If the *bit-wise or* of the attribute

masks of v_x and v_y equals the attribute mask of v_x , all attributes of v_y are covered by v_x .

5.4 GARBAGE COLLECTION SURVEY

Our survey compares the GC implementations of modern in-memory MVCC systems with our novel approach Steam, which we describe in detail in Section 5.3.

Steam is a highly scalable garbage collector that builds on HyPer’s transaction and version management [78]. Long version chains are avoided by pruning them precisely based on the currently active transactions. This is done using an interval-based algorithm similar to that in HANA, except that the version pruning does not happen in the background but is actively done in the foreground by piggy-backing it onto transaction processing [52]. A chain is pruned eagerly whenever it would grow due to an update or insert. This makes the costs of pruning negligibly small as the chain is already latched and accessed anyway by the corresponding update operation.

Hekaton also cleans versions during regular transaction processing [51]. In contrast to Steam, it cleans only those obsolete versions that are traversed during scans, whereas Steam already removes obsolete versions before a reader might have to traverse them. Essentially, Steam prunes a version chain whenever it would grow due to the insertion of a new version—limiting the length of a chain to the number of active transactions. Additionally, Hekaton only reclaims versions based on a more coarse-grained high watermark criterion, while Steam cleans all obsolete versions of a chain.

On a high-level, Steam can be seen as a practical combination and extension of various existing techniques found in HANA, Hekaton, and HyPer. As will show experimentally, seemingly-minor differences have a dramatic impact on performance, scalability, and reliability. In the remainder of the section, we discuss different design decisions in more details and summarize them in Table 9.

TRACKING LEVEL Database systems use different granularities to track versions for garbage collection. The most fine-grained approach is GC on a *tuple-level*. The GC identifies obsolete versions by scanning over individual tuples. Commonly this is implemented using a background vacuum process that is called periodically. However, it is also possible to find and clean the versions in the foreground during regular transaction processing. For instance,

Table 9: Garbage Collection Overview – Categorizing different GC implementations of main-memory database systems

	Tracking Level	Frequency (Precision)	Version Storage	Identification	Removal
BOHM [23]	Txn Batch	Batch (watermark)	Write Set (Full-N20)	Epoch Guard (FG)	Interspersed
Deuteronomy [60]	Epoch	Threshold (watermark)	Hash Table (Full-N20) ¹	Epoch Guard (FG)	Interspersed
ERMIA [46]	Epoch	Threshold (watermark)	Logs (Full-N20)	Epoch Guard (FG)	Interspersed
HANA [52]	Tuple/Txn/Table	1/10s (watermark/exact)	Hash Table (Full-N20) ²	Snapshot Tracker (BG)	Background
Hekaton [15, 16, 51]	Transaction	1 min (watermark) ³	Relation (Full-02N)	Txn Map (BG)	On-the-fly+Inter. ⁴
HyPer [78]	Transaction	Commit (watermark)	Undo Log (Delta-N20)	Global Txn List (FG)	Interspersed
Peloton [86]	Epoch	Threshold (watermark)	Hash Table (Full-N20)	Global Txn List (FG)	Background
Steam	Tuple/Txn	Version Access (exact)	Undo Log (Delta-N20)	Local Txn Lists (FG)	On-creation+Inter.

¹ The version records in the hash table only contain a logical version offset while the actual data is stored in a separate version manager.

² HANA keeps the oldest version in-place.

³ Default value: Hekaton changes the GC frequency according to the workload.

⁴ GC work is assigned (“distributed”) by the background thread.

Hekaton's worker threads clean up all obsolete versions they see during query processing. Since this approach only cleans the traversed versions, Hekaton still needs an additional background thread to find the remaining versions [16].

Alternatively, the system can collect versions based on transactions. All versions created by the same transaction share the same commit timestamp. Thus, multiple obsolete versions can be identified and cleaned at once. While this makes memory management and version management easier, it might delay the reclamation of individual versions compared to the more fine-grained tuple-level approach.

Epoch-based systems go a step further by grouping multiple transactions into one epoch. An epoch is advanced based on a threshold criterion like the amount of allocated memory or the number of versions. BOHM also uses epochs, but since it executes transactions in batches, it also tracks GC on a batch level.

The coarsest granularity is to reclaim versions per table. This makes sense when it is certain that a given set of transactions will never access a table. Only then the system can remove all of the table's versions without having to wait for the completion of these transactions. Since this only works for special workloads with a fixed set of given operations, e.g., stored procedures or prepared statements, this approach is rarely used. HANA is the only system we are aware of that applies this approach as an extension to its tuple and transaction-level GC [52]. In general, the database system cannot predict with certainty which tables will be accessed during the lifetime of a transaction.

FREQUENCY AND PRECISION Frequency and precision indicate how quickly and thoroughly a GC identifies and cleans obsolete versions. If a GC is not triggered regularly or does not work precisely, it keeps versions longer than necessary. The epoch-based systems control GC by advancing their global epoch based on a certain *threshold* count or memory limit. Thus, the frequency highly depends on the threshold setting.

Systems building on a background thread for GC, trigger the background thread *periodically*. Thus, the frequency of GC depends on how often the background thread is called. Since HANA and Hekaton use the background thread to refresh their high watermark, garbage collection decisions are made based on outdated information if the GC is called too infrequently. In the worst case,

GC is stalled until the next invocation of the background thread. Systems like Hekaton, change the interval adaptively based on the current load [51].

BOHM's organizes and executes its transactions in *batches*. GC is done at the end of a batch to ensure that all of its transactions have finished executing. Only versions of previously executed batches, except for the latest state of a tuple, can be GC'ed safely.

Besides the frequency of GC, its thoroughness is mostly determined by the way a GC identifies versions as removable. Timestamp-based identification is not as thorough as an interval-based approach. The timestamp approach is more approximate because it only removes versions whose strictly chronological timestamps have fallen behind the *high watermark* which is set by the minimum start timestamp of the currently active transactions. Since the high watermark is bound to the oldest active transaction, long-running transactions can block the entire GC progress as long as they are active. In these cases, an interval-based GC can still make progress by excising obsolete versions from the middle of chains. In general, an interval-based GC only keeps required versions and thereby cleans the database *exactly*.

VERSION STORAGE Most systems store the version records in global data structures like hash tables. This allows the system to reclaim every single version independently. The downside is that the standard case, where all versions of an entire transaction fall behind the watermark, becomes more complex, as the versions have to be identified in the global storage. Depending on the implementation, this can require a periodical background vacuum process.

For this reason, HyPer and Steam store their versions directly within the transaction, namely the *Undo Log*. When a transaction falls behind the high watermark, all of its versions can be reclaimed together as their memory is owned by the transaction object. Nevertheless, single versions can still be pruned (unlinked) from version chains. Only the reclamation of their memory is delayed until the owning transaction object is released. In general, using the transaction's undo log as version storage is also appealing since the undo log is needed for rollbacks anyway. Using an undo log entry as a version record is straightforward as the stored-before images contain all information to restore the previous version of a tuple.

For space reasons, we only store the *delta*, i.e., the changed attributes, in the version records. If a system stores the *entire tuple*, updating wide tables or tables

with var-size attributes like strings or BLOBs can lead to several unnecessary copy operations [106].

Hekaton’s version management is special in the sense that it does not use a contiguous table space with in-place tuples. The versions of a tuple are only accessible from indexes. For this reason, Hekaton does not distinguish between a version record and a tuple. Additionally, it is the only of the considered system that orders the records from oldest-to-newest (O2N). This order forces transactions to traverse the entire chain to find the latest version which makes the system’s performance highly dependent on its ability to prune old versions quickly [106]. O2N-ordering also makes the detection of write-write conflicts more expensive as the transactions have to traverse the entire chain to detect the existence of a conflicting version. The same holds for rollbacks which also need to traverse entire chains to revert and remove previously installed versions.

IDENTIFICATION If commit timestamps are assigned monotonically, they can be used to identify obsolete versions. All versions committed before the start of the oldest active transaction can be reclaimed safely. The start timestamp of the oldest active transaction can be determined in constant time when the active transactions are managed in an ordered data structure like a *global txn list*, or a *txn map*.

Since pure timestamp-based approaches miss in-between versions as discussed in Section 5.2.1, systems like HANA and Steam complement it with a more fine-grained interval-based approach. While this approach keeps the lengths of version chains minimal, it is also more complex to implement it. The systems have to keep track of all active transactions and perform interval-based intersections for every version chain. HANA does this by tracking all transactions that started at the same time using a reference-counted list (“*Global STS Tracker*” [52]). In Section 5.3.3, we propose a more scalable alternative implementation using *local txn lists*.

For a more coarse-grained garbage collection, it is also possible to control the lifetimes of versions in epochs. This essentially approximates the more exact timestamp-based watermark used by the other systems. Nevertheless, epoch-based memory management is an appealing technique in database systems as it can be used to control the reclamation of all kinds of objects—not only versions. When a transaction starts, it registers itself in the current epoch by entering the epoch. This causes the *epoch guard* to postpone all memory deallocations/version removals made by the transaction until all other threads have

left this epoch and thus will not access them anymore. While managing the versions in epochs limits the precision of the GC, it allows a system to execute transactions without having monotonically increasing transaction timestamps. For instance, in timestamp ordering-based MVCC systems like Deuteronomy or BOHM versions might be created or accessed in a different order than their logical timestamps suggest [23, 60].

Independent of the chosen data structure, the identification which versions are obsolete can either be done periodically by a background (*BG*) thread or actively in the foreground (*FG*).

REMOVAL In HANA, the entire GC work is done by a dedicated background thread which is triggered periodically. Hekaton cleans all versions *on-the-fly* during transaction processing. Whenever a thread traverses an obsolete version, it removes it from the chain. Note, that this only works for *O₂N*, when the obsolete (old) versions are stored in the beginning and thus are always traversed by the transactions. To clean infrequently-visited tuples as well, Hekaton runs a background thread that scans the entire database for versions that were missed so far. The background thread then assigns the removal of those versions to the worker threads which intersperse the GC work with their regular transaction processing.

A common pattern in epoch-based systems is to add committed versions along with the current epoch information to a free list. When a transaction requires a new version, it checks whether it can reclaim an old version from the free list based on the current epoch. Thereby, version removal essentially happens interspersed with normal transaction processing. However, the epoch guard should periodically release more than the newly required versions. Otherwise, the overall number of versions can only go up over time as all reused versions eventually end up in the free-list again. Deuteronomy addresses this by limiting the maximum number of versions. When the hard limit is reached, no more version creations are permitted and the threads are co-opted into performing GC until the number of versions is under control again [60].

HyPer and Steam also perform the entire GC work in the foreground by *interspersing* the GC tasks between the execution of transactions. If there are obsolete versions, the worker threads reclaim them directly after every commit. Thereby, GC becomes a natural part of the transaction processing without the need for an additional background thread. This makes the system self-regulating and robust to peaks at the cost of a slightly increased commit latency. Steam, additionally, prunes obsolete versions *on-creation* whenever it inserts a

new version into a chain. Thereby, Steam ensures that the “polluters” are responsible for the removal of garbage, which relieves the (potentially already slow) readers.

5.5 EVALUATION

In this section, we experimentally evaluate the different GC designs discussed in Section 5.4. To compare their performance, we implemented and integrated these GC approaches into HyPer [78]. For a fair apples-to-apples comparison, we only change the GC while the other components such as the storage layer or the query engine stay the same.

To distinguish our implementations from the original systems we put their names into quotes, e.g., ‘Hekaton’. In our evaluation, we do not include BOHM of our survey in Section 5.4 as its GC is specifically designed for executing transactions in batches, in which concurrency control and the actual transaction execution are strictly separated into two phases [23]. Epoch-based GC—as used by BOHM—is represented by ‘Deuteronomy’ and ‘Ermia’.

We monitor the systems’ performance and capabilities by running the CH benchmark for several minutes. The CH benchmark is a challenging stress test for GCs because its short-lived OLTP transactions face long-living queries [11, 29, 90]. To better understand the general characteristics of the different systems we run some additional experiments. We analyze the scalability and overhead of each approach using the TPC-C benchmark. TPC-C is a pure OLTP benchmark without long-running transactions that could lead to the “vicious cycle of garbage”. To evaluate different workload characteristics, we run the updates along with varying percentages of concurrent reads. We also explore the effects of skewed updates as they can be particularly challenging for garbage collectors by leading to potentially long version chains. Finally, we evaluate the effectiveness of *EPO* in keeping version chains short in isolation.

Table 10 summarizes the key features of our different GC implementations. All systems order the chains from N_2O . The high watermark is either defined as the start timestamp of the oldest active transaction or epoch. All versions that were committed before that point in time are obsolete, as all active transactions already work on more recent snapshots of the data. Additionally, ‘Hana’ and Steam use a more *exact* form of GC that prunes intermediate versions in

Table 10: Configuration and Setup

	Watermark	Exact	Frequ.	Find/Clean
‘Deuter.’	Epoch (∞)	–	100 txs	FG
‘Ermitia’	Epoch (3)	–	1 tx	FG
‘Hana’	Txn	Lazy	1 ms	BG
‘Hekaton’	Txn	–	1 ms	BG \Rightarrow FG
Steam	Txn	Eager	cont.	FG

chains (cf. Section 5.3.3 for details). While ‘Deuteronomy’ increases its epoch-ids monotonically, ‘Ermitia’ uses a three-phase epoch-guard³.

Another important implementation detail is the *frequency* of garbage collection. For the epoch-based systems, this is the minimal number of committed transactions before the global epoch is advanced and for ‘Hana’ and ‘Hekaton’ this is the time when the background GC thread is invoked. It turns out that the default settings of the systems are not always suitable, so we hand-tuned them to the optimal values. In Section 5.5.4 we show how big the effect of a poorly chosen GC frequency is. Since Steam runs GC continuously whenever a version chain is accessed, there is no need to find and set an optimal interval.

In ‘Hana’, the GC work is done solely by the background thread (BG). ‘Hekaton’ uses the background thread only to refresh the global minimum and to identify obsolete versions. When it finds obsolete versions, it assigns the task of removing them to the worker threads. The other systems intersperse the entire GC work (identification and removal) with their normal transaction processing. Steam additionally prunes version chains eagerly whenever it accesses a version chain.

We evaluate the different approaches on an Ubuntu 18.10 machine with an Intel Xeon E5-2660 v2 CPU (2.20GHz, 3.00GHz maximum turbo boost) and 256GB DDR3 RAM. The machine has two NUMA sockets with 10 physical cores (20 “Hyper-Threads”) each, resulting in a total of 20 physical cores (40 “Hyper-Threads”). The sockets communicate using a high-speed QPI interconnect (16 GB/s).

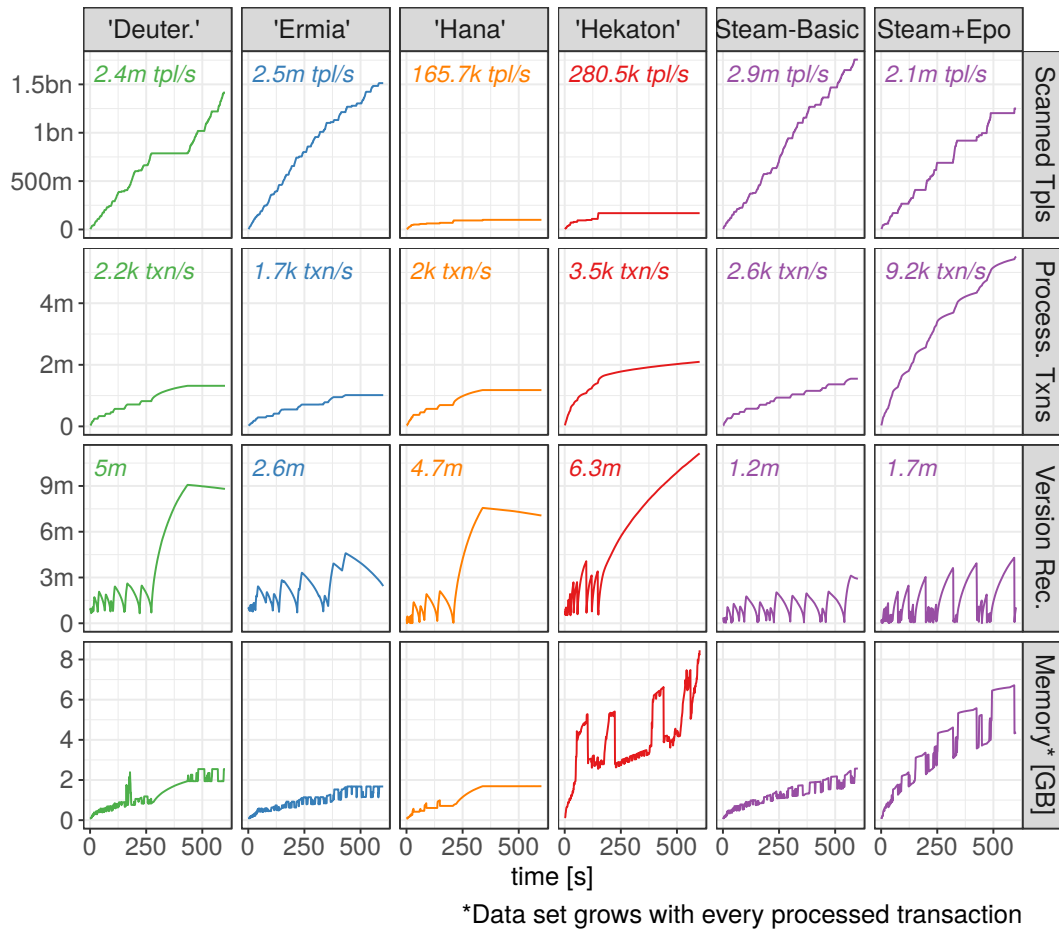


Figure 36: Performance over time – CH benchmark with 1 OLAP and 1 OLTP thread. (Mean values shown in italics)

5.5.1 Garbage Collection Over Time

In this experiment, we put critical stress on the GC by running the mixed CH benchmark. This tests the vulnerability of every approach to long-running transactions and the “vicious cycle” of garbage.

The CH benchmark combines TPC-C write transactions with queries inspired by the TPC-H benchmark. This creates a demanding mix of short-lived write transactions and long-running queries. The gap between short-lived writes and long OLAP queries increases over time as the data set grows with the number of processed transactions⁴. This makes our workload particularly challenging for fast systems like Steam that maintain a high write rate throughout the entire experiment. For comparison, it would take ‘Ernia’ 8356

³ used code from <https://github.com/ermia-db/ermia>

⁴ Every delivery transaction “delivers” 10 orders. Having 45% new-orders and only 4% delivery transactions, approximately 11% of the new orders remain undelivered.

seconds and thereby about $13\times$ as long as Steam to process the same number of transactions reaching the same level of GC complexity.

To account for the data growth, we normalize the query performance by plotting the number of scanned tuples instead of the raw query throughput, following Funke et al.'s suggestion [29] to normalize the query performance using the increasing cardinalities of the relations. The increasing data size is also the reason why the used memory increases over time independently of the number of used/GC'ed versions.

Figure 36 shows the read, write, version record, and memory statistics over 10 minutes. Pruning all versions eagerly that are not required by any active transaction using EPO proves to be an effective addition to Steam. Rather surprisingly the main improvements can be seen in the write throughput (roughly $3\times$ compared to the second-best solution) while the read performance stays about the same. This is due to the fact that the main consumer of long version chains are not long-running queries but GC.

During GC we always have to traverse the entire chain to remove the oldest (obsolete) versions, whereas queries just have to retrieve the version that was valid when they started. For this reason, GC benefits most from short chains leaving more time for actual transaction processing. The increased speed of GC becomes visual when looking at the shapes of the version record curves: while the number of version records goes down gradually in all systems at the end of a long-running query, it drops almost immediately and very sharply when using EPO. This happens because hardly any GC has to be done anymore: most version records are already pruned eagerly from the chains and the remaining version records can be identified very quickly as the owning chains have a maximum length of 2, i.e., the number of active transactions. We analyze and compare those GC performance stats in details in the later Section 5.5.7.

As a side-effect, due to the highly improved write performance, the overall used memory increases faster than without using EPO. This can be accounted to the nature of the CH benchmark as described above: the data set grows with every processed transaction. What this means, in turn, is that reads also get more expensive as they have to scan more data (cf. memory plot). The increased query response times lead to bigger gaps between the short-lived writes and the long-lived queries, which is why the number of version records is a little bit higher with EPO. However, the average number of active version records only goes up by 42%, whereas the number of writes (which can be directly translated to the number of produced version records) increases significantly by 354%.

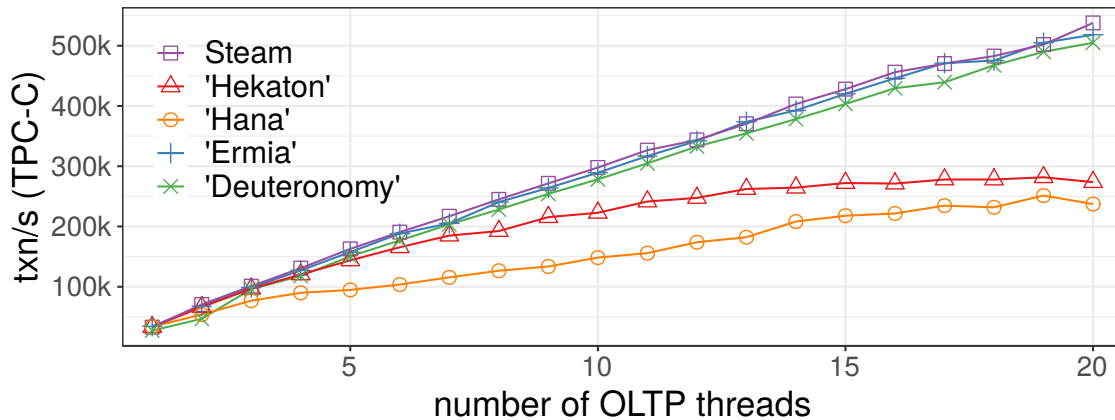


Figure 37: TPC-C – Performance for increasing number of OLTP threads (100 warehouses)

The epoch-based systems ‘Deuteronomy’ and ‘Ernia’ conceptually follow the same approach as the basic version of Steam using a watermark only. For this reason, the performance looks quite similar. There is only a slight setback compared to the basic version of Steam, which is probably caused by the epochs being a little bit too coarse-grained for a mixed workload and that maintaining the global epoch introduces a small overhead.

‘Hana’ runs into more problems because it does the GC work exclusively in its background thread. With increasing gaps between the quick writers and the slow readers, the number of versions becomes too big and the single background thread becomes overwhelmed by the work.

‘Hekaton’ cleans the versions in the foreground, but it offloads the GC control, i.e., maintaining the high watermark and assignment of GC work, to the background thread. This detached workflow increases the GC latency to a point, where it gets out of control and the number of versions grows quickly.

5.5.2 TPC-C

While the previous experiment analyzed a mixed workload, we now want to show that the design and choice of a GC is also critical in pure OLTP workloads without any long-running transactions. Since we only interchange the GC, we can directly compare the overhead and scalability of the different approaches.

The TPC-C numbers in Figure 37 show that the foreground-based systems ‘Ernia’, ‘Deuteronomy’, and Steam scale best. ‘Hana’ falls slightly behind because it uses a centralized “Global Snapshot Tracker” that requires a global mutex.

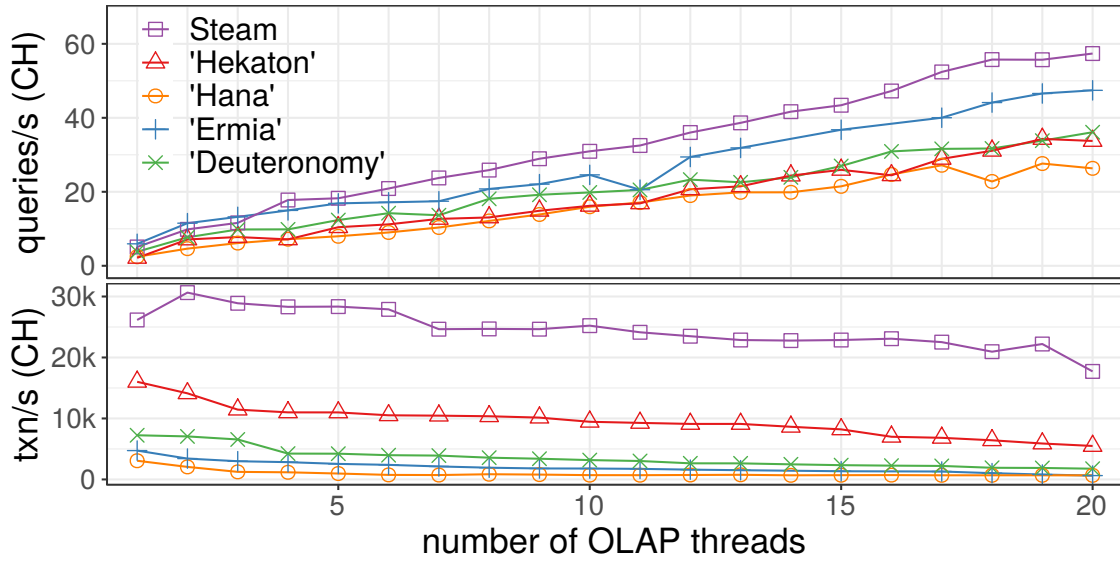


Figure 38: **CH benchmark** – Performance for increasing number of OLAP threads using 1 OLTP thread

While ‘Hekaton’ is superior to ‘Hana’, it is still limited by the use of its background thread which coordinates the GC. The background thread periodically retrieves the global minimum from the global transaction map and populates it to the threads. Additionally, it collects obsolete versions and assigns them to the work queues of the threads. While this allows the workers to remove the garbage cooperatively, there is still the single-threaded phase of identifying the garbage and “distributing” it. Furthermore, there is a small but constant synchronization overhead caused by the global transaction map. Although it is implemented latch-free, it still falls behind the thread-local implementations of Steam and the epoch-based solutions. This aligns well with recent findings that synchronous communication should be avoided and using latch-free data structures can even have worse performance than traditional locking [22, 103].

These results indicate that GC has a big impact on the system’s performance in every kind of high-volume workload and not only in mixed workloads. For efficient GC global data structures and synchronous communication have to be avoided. In Section 5.5.5 we will see even bigger impacts on the system’s scalability when running “cheap” key-value update transactions instead of TPC-C. When the transaction rate becomes very high, the maintenance of a global epoch starts to become a notable bottleneck.

5.5.3 Scalability in Mixed Workloads

In this section, we take another look at the CH benchmark. This time, we focus on the scalability by varying the number of read threads. In contrast to the previous time-bound experiment, now, every system processes a fixed number of 1 million TPC-C transactions. This makes the throughput numbers more comparable, as the query response times increase with every processed transaction due to growing data [29].

Figure 38 shows that the throughput of the single OLTP thread is highly affected by concurrent OLAP threads. This can be accounted to effects caused by the vicious cycle of garbage. As seen in Section 5.5.1, the versions accumulate quickly over time slowing down the readers. When the read transactions get slower, the version records have to be retained longer which amplifies this effect further. Additionally, the GC work and the slow readers create increased contention on the tuple latches as they require more time to retrieve a version. Hence, it is crucial to keep the number of version records as low as possible.

Steam's EPO reduces the number of versions effectively by pruning the version chains eagerly. This makes its GC and write performance superior to the other systems which struggle because their GC is too coarse-grained (epochs/high watermark). Even 'Hana' which also uses precise cleaning cannot keep up with Steam since its background pruning is not as effective as Steam's eager pruning (cf .Section 5.3.3 for a detailed comparison). At higher numbers of active read transactions, Steam's write performance degrades slightly because of the increasing likelihood that more versions have to be kept in the chains. Ideally, all transactions started at the same time and Steam only needs to keep one version per chain. This can be achieved by batching the start of readers in groups (similar to a group commit). Having fewer start timestamps improves the performance and effectiveness of EPO. Therefore, the performance could be improved slightly by artificially delaying some queries so that all queries share the same start timestamp. An evaluation of this idea showed gains of a few percents—at the cost of increased query latencies.

5.5.4 Garbage Collection Frequency

In Steam, GC happens continuously: Version chains are pruned whenever they are updated. Thus, the frequency is implicitly given and self-regulated by the workload. For the other systems, the frequency has to be explicitly set by

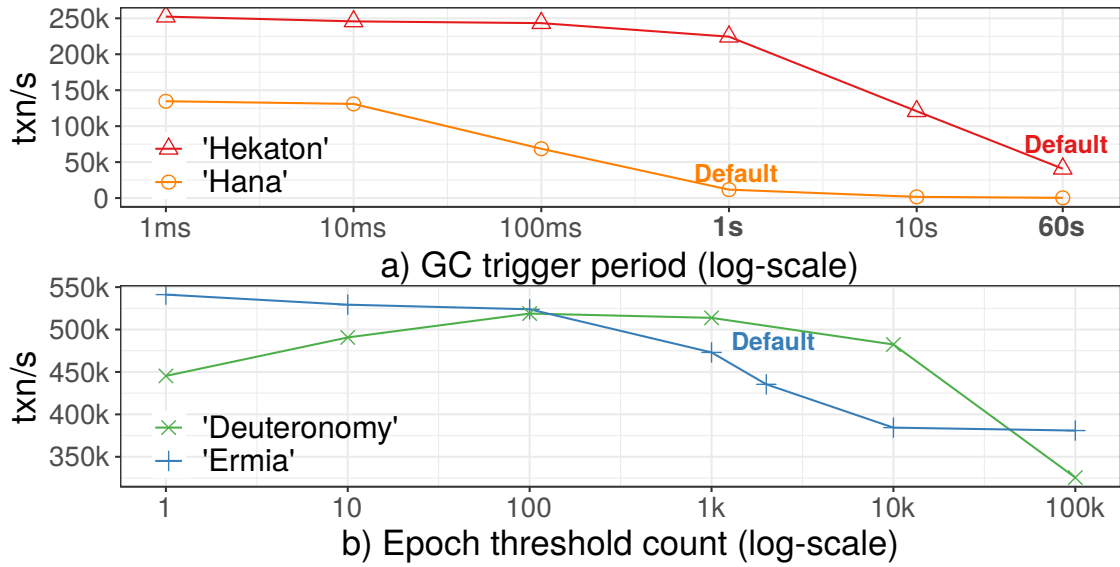


Figure 39: GC Frequency – Varying *a)* the period when the GC thread is triggered or *b)* the count of committed transactions before an epoch might be advanced (TPC-C, 20 OLTP threads)

a parameter which is either a time period in which the background GC thread is triggered (*a*) or a threshold that has to be reached before the global epoch is advanced (*b*).

The optimal period depends on the workload and the performance of the system. A faster system with high update rates generates more versions and has to be cleaned more frequently. To determine the optimal setting for the use with HyPer, we run TPC-C with different GC frequencies. Figure 39 shows the throughput when varying the trigger frequency from 1 ms to 60s and epoch thresholds from 1 to 100k processed transactions.

For all systems, we see the best results when we trigger the GC as frequently as possible. For the background-thread approaches, we achieved the best results by setting the period to 1 ms. The period time cannot be decreased further, as the processing time of the GC thread would exceed its invocation intervals.

For the epoch-based systems, it is also best to set the epoch threshold as low as possible. This means that the system tries to advance the global epoch after every single committed transaction. However, refreshing the global epoch is not for free as this requires entering a critical section and/or scanning of other thread-local epochs. While the three-phase epoch-guard of 'Ermitia' handles this case very efficiently, refreshing the global epoch in 'Deuteronomy' which uses infinite epochs is more expensive. For this reason, the best threshold setting for 'Deuteronomy' is slightly higher at 100. This gives the best tradeoff between fast (immediate) GC and the overhead for refreshing the global epoch.

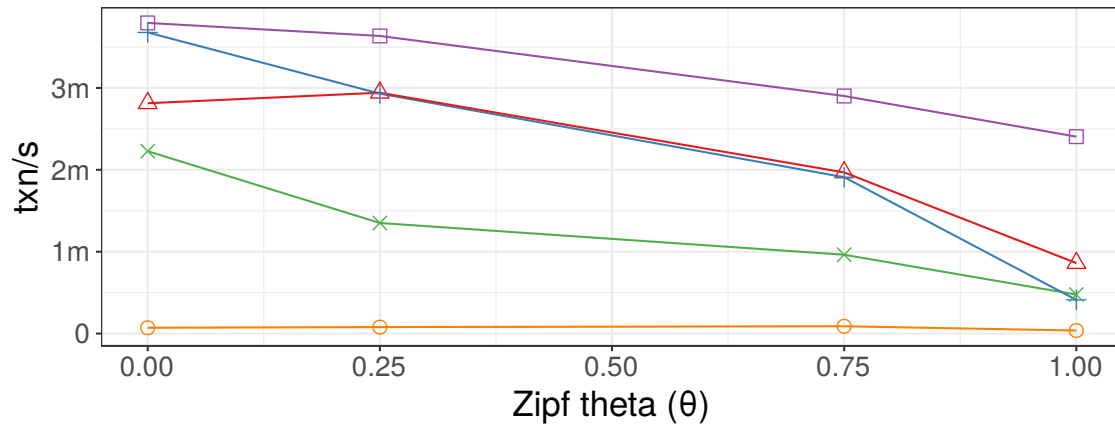


Figure 40: Cheap key-value updates – Increasing the skew in key-value updates (using 20 OLTP threads)

This experiment shows that the choice GC frequency can have a tremendous effect on the system’s performance. There is a difference of more than $500\times$ only by changing the frequency parameter. In practice, this could create critical instability if the system does not adjust this setting timely. This indicates that the frequency should be chosen based on the workload, i.e., the number of produced garbage (transactions) and not a fixed time interval. Otherwise, the back pressure on the GC can easily become too high. Even in the worst measured configuration, the epoch-based systems that control GC based on the number of processed transactions outperform the best time-interval-based GC. In Steam, we take this concept even a step further by pruning the chains eagerly whenever a new version is added.

5.5.5 Skew

When all updates are distributed evenly, every version chain tends to be equally short. However, in the real world, we often have skewed workloads. When certain tuples are updated more often their version chains get longer making GC more expensive. To measure the effectiveness of the GCs in skewed scenarios, we run key-value updates on a table using different Zipfian distributions. Figure 40 shows the throughput for theta values from 0.0 (no skew) to 1.0 (significant skew).

Steam is robust to skew because it deeply integrates GC into the transaction processing. Version chains that would become long can be pruned while, or rather before they grow (during an update). Other systems delay GC for longer: in particular, the time-based systems ‘Hana’ and ‘Hekaton’ which trig-

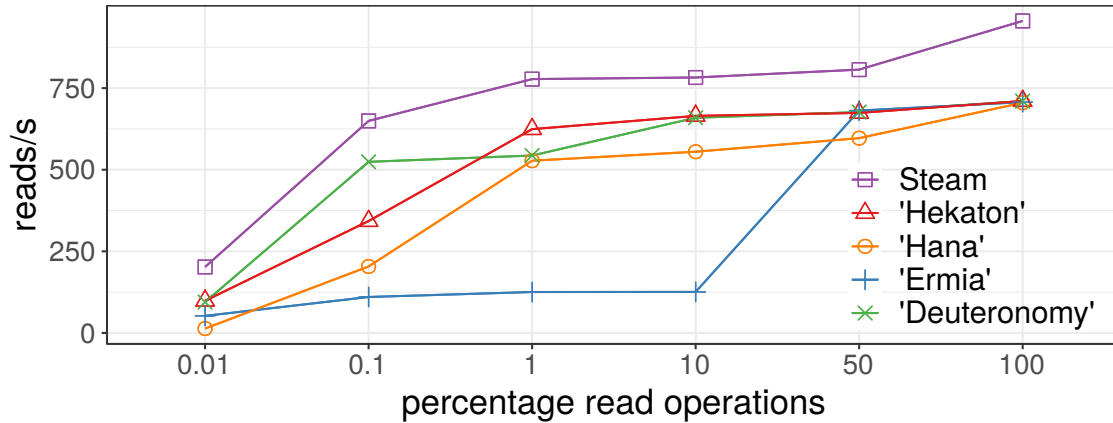


Figure 41: Varying read-write ratios – Mixing table scans and key-value update transactions (20 threads)

ger GC only periodically, can be affected most. In the worst case, when only one tuple is updated all the time, the length of its version chain grows to the current number of updates per GC interval. At a throughput of 10,000 txn/s that would generate a chain of 10,000 versions assuming a GC interval of 1 s (default for HANA). In our experimental results, this effect is mostly diminished because we decreased the GC to 1 ms, but we can still see the systems falling behind Steam.

Unfortunately, the results for 'Hana' and to some degree 'Deuteronomy' are not very meaningful for increased skew as their performance is mostly dominated by their limited scalability. The results for a theta value of 0.0 indicate an overhead in high-volume workloads. This can be accounted to the use of a global mutex for the snapshot tracker ('Hana') and a relatively expensive refreshing of the global epoch counter in 'Deuteronomy'. By contrast, the three-phase epoch manager of 'Ernia' scales significantly better.

5.5.6 Varying Read-Write Ratios

In this experiment, we analyze how effective each approach is for different read/write setups. We run two kinds of transactions: write transactions updating tuples and read-only transactions doing full table scans, whereas all transactions operate on the same table. We vary the ratio of reads and writes by increasing the percentage of read operations every thread performs. Figure 41 shows the number of read operations for a decreasing number of writes.

The read performance increases as expected when the workload mix shifts towards being read-only, whereas Steam performs best in all setups. Especially

in the read-only case, Steam’s minimal overhead is clearly visible: A read-only thread never retrieves the set of active transaction ids (including the global minimum). This is only done when it has recently committed versions (i.e., its committed transaction list is not empty), or lazily during its first update operation. In the read-only case, every thread only has to signal its currently active transaction by adding it to its thread-local list. By contrast, all other systems require at least a basic form of synchronization, i.e., entering an epoch, or registering the transaction in a globally shared transaction map/tracker.

In the more write-heavy cases, EPO helps Steam to control the number of versions speeding up the readers. For high numbers of writes (<10% reads), ‘Ermia’ falls behind the other systems. While its three-phase epoch guard showed very good scalability in the other experiments, it seems to be too coarse-grained now. The more fine-grained infinite epochs of ‘Deuteronomy’ perform significantly better in these cases.

5.5.7 Eager Pruning of Obsolete Versions

To avoid long version chains in mixed workloads, we implemented *EPO* (cf. Section 5.3.3) to prune the chains eagerly whenever a new version is inserted. *EPO* removes all versions as soon as they are not required by any active transaction anymore.

Table 11 shows that this reduces the number of traversed versions significantly in the CH benchmark. Steam processes the given set of transactions 5× faster using *EPO*. Without the optimization, the GC cannot keep the number of versions down effectively since the high watermark approach is too coarse-grained. The version chains grow quickly hitting a maximum length of 30287. When the optimization is enabled, the maximum length goes down to two versions. The “optimized” chain only keeps the most recent version of the writer and an older version that is visible to the reader.

Rather surprisingly, or even paradoxically, the more thorough and fine-granular we clean our system, the less time we spend cleaning. Using *EPO*, the system spends less than 100ms in total on GC, while it requires 1.5s using the standard watermark approach. This performance difference becomes clear when we look at the number of traversed versions: it is reduced from 1.2 billion to only 4.2 million. Since *EPO* keeps version chains short at all time, it is always cheap to identify and reclaim obsolete versions. In particular, when an entire transaction falls behind the “watermark” and we finalize its

Table 11: Effect of using EPO – CH benchmark with 1 read thread and 1 write thread running 300k transactions in total

	Standard Watermark	EPO Exact		
Version Removal (GC)				
<i>Traversed Versions</i>	1,197m	4.2m		
<i>Avg. Chain Length (max)</i>	287.43 (30287)	1.07 (2)		
Table Scans (Queries)				
<i>Traversed Versions</i>	120m	37m		
<i>Avg. Chain Length (max)</i>	1.00 (141)	1.00 (2)		
Breakdown	Time	[%]	Time	[%]
Fetch Active Txn-Ids	<1 ms	0.01	<1 ms	0.01
Prune Chains (EPO)	–	–	8.4 ms	0.07
Finalize Entire Txns	1.5 s	4.47	81 ms	0.68
Version Retrieval (Scan)	4.2 s	12.26	1.1 s	8.79
Queries/s	4.8		5.1	
Transactions/s	6554		30,580	

versions, most of its versions are already removed from the chains by EPO (thus, GC of them is a no-op) or belong to very short chains which makes unlinking them from the chain fairly cheap.

We also see that maintaining the set of active transaction ids does not add any overhead. For the watermark-approach, all thread-local minimums have to be fetched anyway. The additional sorting step required by EPO is negligible cheap since at most #-threads integers are sorted.

Faster GC is very beneficial for transaction processing in general, as slow, interspersed GC work can stall the processing of writes. Thus, faster GC gives the worker threads more time to process transactions.

The average length of version chains is significantly higher during version removal than it is during table scans. This happens because some tuples (counter and warehouse statistics) are updated frequently, but are never read by any query [11]. The readers mainly access parts of the tables that are updated evenly. Thus, the positive effect of EPO is not as big for queries as it is for the writes. The maximum chain length during a table scan is “only” 141 without the optimization. With the optimization, the scans have to retrieve $3.24\times$ fewer versions. This is reflected by a slightly improved query performance. The benefit would be significantly higher if the readers would need to access the frequently updated tuples with lengths of more than 30,000.

5.6 RELATED WORK

In recent years, the performance of systems in mixed workloads (HTAP) was studied extensively [108, 50, 98, 106, 83, 3]. Several systems were developed focusing on scalability in high volume OLTP workloads [85, 91, 34, 62, 46]. A reoccurring topic is to optimize the concurrency control protocol, e.g., by tuning the validation phase or reordering transaction [20, 33, 92, 102]. Although most of the papers mention the use and importance of an efficient garbage collector, the implementation is either described only briefly or not mentioned at all. Recent work on GC is mostly related to large data systems in which the challenges and tasks are very different and not comparable to version reclamation in MVCC systems [107, 65]. In summary, most components of MVCC systems are well-understood, studied, and optimized but there is little research on efficient GC — despite its big impacts on performance.

Handling of long-living transactions is an inherent problem of MVCC systems studied by others. Lee et al. [52] describe practical solutions to this problem such as: (1) flushing old versions to disk if main memory is exceeded, (2) aborting long-running transactions (user gets an error), and (3) closing transactions as soon as possible (e.g., after query results are materialized). However, these solutions are not applicable to high volume workloads. One proposal for such workloads is to create virtual memory snapshots (forks) for read-only queries [73, 95]. However, this strongly affects the overall scalability of the system as it requires a shared mutex per column.

Modern and fast OLTP systems like TicToc or Silo often use a single-version approach instead of MVCC [109, 100, 21, 97]. A single-version system only maintains the latest version of a tuple and thus there is no need for garbage collection. This makes them particularly fast in OLTP workloads. However, by default, they are not designed to handle OLAP or mixed workloads as they would have to maintain a large read set. Since this is can easily lead to aborts, Silo also allows creating snapshots of the data by storing old tuple versions. Due to the costs of snapshots creation, snapshots are only taken periodically, i.e., every second, which results in slightly stale data [100].

Systems that apply Serialization Graph Testing (SGT) instead of timestamps have to keep a transaction and its items until its existence does not influence any other or future transactions [21, 35].

5.7 SUMMARY

In this chapter, we showed the importance of garbage collection for in-memory MVCC systems on modern many-core systems. We find that GC should be based on thread-local data structures and asynchronous communication for optimal performance. Further, it is crucial for HTAP workloads of short-lived writes and long-running reads to keep the number of active versions as low as possible. With traditional high watermark-based approaches, a single long-running transaction blocks GC progress during its lifetime.

Our novel, scalable GC *Steam* speeds up transaction processing and garbage removal by pruning all obsolete versions eagerly whenever a new version is added. Thereby, *Steam* effectively limits the length of chains to the number of active transactions. Besides HTAP workloads, our experimental results indicate that *Steam* benefits all kind of workloads from write-only to read-only. Its seamless integration into transaction processing enables superior performance compared to other state-to-the-art GC approaches which detach GC from transaction processing.

Due to its effectiveness, this approach was already successfully adopted by new systems [14, 28, 27].

6 | CONCLUSION

The main goal of this dissertation was to improve the scalability and concurrency of modern database systems. We first analyzed locks and synchronization in depth to find an optimal synchronization approach for a database system.

We showed the superior performance of optimistic locking, especially in hierarchical index structures. To support optimistic and pessimistic shared locking, we introduced a hybrid read-write lock that could protect all physical data structures used in a DBMS efficiently.

On top of the hybrid lock, we assembled a versatile parking lot for waiting threads that adds reasonable fairness, contention handling, and cache topology awareness to every lock. The appealing idea of a parking lot is that it adds all of these features without sacrificing any “fast path” performance or requiring any additional in-place space for the locks. Its versatility and performance characteristics make a hybrid lock backed by a parking lot ideal for a general-purpose DBMS.

In the second part of this thesis, we focussed on the scalability of concurrent transactions in an MVCC database system. Running transactions in parallel brings new challenges, such as frequent aborts under high contention. We discussed different backoff and partitioning strategies to handle or avoid those aborts in Section 4.5. After that, we focussed on an even more fundamental problem of MVCC: the handling of transactions in the presence of long-running queries. This scenario stresses traditional version garbage collectors and can lead to long version chains. We proposed a scalable new garbage collection strategy to tackle this problem. Instead of garbage-collecting old versions only after the completion of a long-running transaction, we now prune versions eagerly whenever we update a tuple. This approach keeps the lengths of the version chains to their required minimum. The seamless integration of garbage collection into transaction processing enables superior performance and scales naturally with the workload.

BIBLIOGRAPHY

- [1] AMD. *HPC Tuning Guide for AMD EPYC[®] processors*. <http://developer.amd.com/wp-content/resources/56420.pdf>. 2018.
- [2] Abdelhalim Amer, Huiwei Lu, Yanjie Wei, Jeff Hammond, Satoshi Matsuoka, and Pavan Balaji. “Locking aspects in multithreaded MPI implementations”. In: *Argonne National Lab., Tech. Rep. P6005-0516* (2016).
- [3] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. “The Case For Heterogeneous HTAP”. In: *CIDR*. 2017.
- [4] Naama Ben-David and Guy E Blelloch. “Fast and Fair Randomized Wait-Free Locks”. In: *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*. 2022, pp. 187–197.
- [5] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. “HOT: A Height Optimized Trie Index for Main-Memory Database Systems”. In: *SIGMOD*. 2018, pp. 521–534.
- [6] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. “Scalable and robust latches for database systems”. In: *DaMoN*. 2020.
- [7] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. “Scalable Garbage Collection for In-Memory MVCC Systems”. In: *VLDB* (2019).
- [8] Lucas Braun, Thomas Etter, Georgios Gasparis, Martin Kaufmann, Donald Kossmann, Daniel Widmer, Aharon Avitzur, Anthony Iliopoulos, Eliezer Levy, and Ning Liang. “Analytics in Motion: High Performance Event-Processing AND Real-Time Analytics in the Same Database”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015, pp. 251–264.
- [9] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J Marathe, and Nir Shavit. “NUMA-aware reader-writer locks”. In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2013, pp. 157–166.

- [10] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. "Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems". In: *VLDB*. 2001.
- [11] Richard Cole et al. "The Mixed Workload CH-benCHmark". In: *Proceedings of the Fourth International Workshop on Testing Database Systems*. Athens, Greece, 2011.
- [12] Travis Craig. *Building FIFO and priorityqueuing spin locks from atomic swap*. Tech. rep. Citeseer, 1993.
- [13] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. "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*. 2013.
- [14] Dean De Leo and Peter Boncz. "Teseo and the analysis of structural dynamic graphs". In: *Proceedings of the VLDB Endowment* 6 (2021).
- [15] Kalen Delaney. "SQL Server In-Memory OLTP internals overview". In: *White Paper of SQL Server* (2014).
- [16] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. "Hekaton: SQL Server's memory-optimized OLTP engine". In: *SIGMOD*. 2013.
- [17] Dave Dice and Alex Kogan. "Compact NUMA-Aware Locks". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. Dresden, Germany, 2019.
- [18] Dave Dice and Alex Kogan. "Hemlock: Compact and Scalable Mutual Exclusion". In: *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*. 2021, pp. 173–183.
- [19] David Dice, Virendra J. Marathe, and Nir Shavit. "Lock Cohorting: A General Technique for Designing NUMA Locks". In: *ACM SIGPLAN*. New Orleans, Louisiana, USA, 2012.
- [20] Bailu Ding, Lucja Kot, and Johannes Gehrke. "Improving optimistic concurrency control through transaction batching and operation reordering". In: *PVLDB* 2 (2018).
- [21] Dominik Durner and Thomas Neumann. "No False Negatives: Accepting All Useful Schedules in a Fast Serializable Many-Core System". In: *ICDE*. 2019.

- [22] Jose M. Faleiro and Daniel J. Abadi. “Latch-free Synchronization in Database Systems: Silver Bullet or Fool’s Gold?” In: *CIDR*. 2017.
- [23] Jose M. Faleiro and Daniel J. Abadi. “Rethinking serializable multiversion concurrency control”. In: *PVLDB* 11 (2015).
- [24] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. “Unlocking Energy”. In: *USENIX ATC 16*. Denver, CO, June 2016.
- [25] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. “SAP HANA Database: Data Management for Modern Business Applications”. In: *SIGMOD Record* 4 (2012).
- [26] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. “Fuss, futexes and furwocks: Fast userlevel locking in linux”. In: *AUUG Conference Proceedings*. 2002, pp. 479–495.
- [27] Michael Freitag, Alfons Kemper, and Thomas Neumann. “Memory-optimized multi-version concurrency control for disk-based database systems”. In: *Proceedings of the VLDB Endowment* 11 (2022).
- [28] Per Fuchs, Domagoj Margan, and Jana Giceva. “Sortledton: a universal, transactional graph data structure”. In: *Proceedings of the VLDB Endowment* 6 (2022).
- [29] Florian Funke, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Anisoara Nica, Meikel Poess, and Michael Seibold. “Metrics for Measuring the Performance of the Mixed Workload CH-benCHmark”. In: *TPCTC*. 2011.
- [30] Andrew Pavlo. *Multi-Version Concurrency Control (Garbage Collection)*. <https://15721.courses.cs.cmu.edu/spring2019/slides/05-mvcc3.pdf>. Jan. 2019.
- [31] Vincent Blanchon. *Go: Mutex and Starvation*. <https://medium.com/a-journey-with-go/go-mutex-and-starvation-3f4f4e75ad50>.
- [32] Goetz Graefe. “A survey of B-tree locking techniques”. In: *ACM Trans. Database Syst.* 3 (2010).
- [33] Jinwei Guo, Peng Cai, Jiahao Wang, Weining Qian, and Aoying Zhou. “Adaptive optimistic concurrency control for heterogeneous workloads”. In: *Proceedings of the VLDB Endowment* 5 (2019).

- [34] Aditya Gurajada, Dheren Gala, Fei Zhou, Amit Pathak, and Zhan-Feng Ma. “BTrim: hybrid in-memory database architecture for extreme transaction processing in VLDBs”. In: *PVLDB* 12 (2018).
- [35] Thanasis Hadzilacos and Nihalis Yannakakis. “Deleting Completed Transactions”. In: *JCSS* 2 (1989).
- [36] Intel. *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>. 2019.
- [37] Intel. “Intel[®] VTune[™] Profiler User Guide”. In: 2020. Chap. CPU Metrics Reference.
- [38] Intel. *Speculative locking (Transactional Lock Elision)*. <https://software.intel.com/en-us/node/506266>. 2019.
- [39] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. “A quantitative measure of fairness and discrimination”. In: *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA* (1984).
- [40] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. “Scalable and practical locking with shuffling”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 586–599.
- [41] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. “Scalable {NUMA-aware} Blocking Synchronization Primitives”. In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 2017.
- [42] Alfons Kemper and Thomas Neumann. “HyPer: A Hybrid OLTP&OLAP Main Memory Database System based on Virtual Memory Snapshots”. In: *ICDE*. 2011.
- [43] Alfons Kemper and Thomas Neumann. “Hyper: Hybrid oltp&olap high performance database system”. In: (2010).
- [44] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. “Everything you always wanted to know about compiled and vectorized queries but were afraid to ask”. In: *Proceedings of the VLDB Endowment* 13 (2018).
- [45] Timo Kersten, Viktor Leis, and Thomas Neumann. “Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra”. In: *VLDB J.* 5 (2021).

- [46] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. “ERMIA: Fast memory-optimized database system for heterogeneous workloads”. In: *SIGMOD*. 2016.
- [47] Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. “Analytics on Fast Data: Main-Memory Database Systems versus Modern Streaming Systems”. In: *EDBT*. 2017.
- [48] Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. “Scalable Analytics on Fast Data”. In: *ACM TODS* (Jan. 2019).
- [49] Orran Krieger, Michael Stumm, Ronald C. Unrau, and Jonathan Hanna. “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*. 1993.
- [50] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwillig. “High-performance concurrency control mechanisms for main-memory databases”. In: *PVLDB* 4 (2011).
- [51] Per-Åke Larson, Mike Zwillig, and Kevin Farlee. “The Hekaton Memory-Optimized OLTP Engine”. In: *IEEE Data Eng. Bull.* 2 (2013).
- [52] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. “Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA”. In: *SIGMOD*. 2016.
- [53] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. “Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age”. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 2014.
- [54] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. “LeanStore: In-Memory Data Management beyond Main Memory”. In: *ICDE*. 2018, pp. 185–196.
- [55] Viktor Leis, Michael Haubenschild, and Thomas Neumann. “Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method”. In: *IEEE Data Eng. Bull.* 1 (2019).
- [56] Viktor Leis, Alfons Kemper, and Thomas Neumann. “Exploiting hardware transactional memory in main-memory databases”. In: *ICDE*. 2014, pp. 580–591.

- [57] Viktor Leis, Alfons Kemper, and Thomas Neumann. "Scaling HTM-Supported Database Transactions to Many Cores". In: *IEEE* 2 (2016).
- [58] Viktor Leis, Alfons Kemper, and Thomas Neumann. "The adaptive radix tree: ARTful indexing for main-memory databases". In: *IEEE*. 2013.
- [59] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. "The ART of practical synchronization". In: *DaMoN*. 2016.
- [60] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. "High Performance Transactions in Deuteronomy". In: *CIDR*. 2015.
- [61] Liang Li, Gang Wu, Guoren Wang, and Ye Yuan. "Accelerating Hybrid Transactional/Analytical Processing Using Consistent Dual-Snapshot". In: *DASFAA*. 2019.
- [62] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. "Cicada: Dependably fast multi-core in-memory transactions". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017.
- [63] Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, and Haibo Chen. "CLOF: A Compositional Lock Framework for Multi-Level NUMA Systems". In: *SIGOPS*. Virtual Event, Germany, 2021.
- [64] Sandeep Lodha and Ajay Kshemkalyani. "A fair distributed mutual exclusion algorithm". In: *IEEE Transactions on Parallel and Distributed Systems* 6 (2000), pp. 537–549.
- [65] Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. "Lifetime-based memory management for distributed data processing systems". In: *PVLDB* 12 (2016).
- [66] Darko Makreshanski, Justin J. Levandoski, and Ryan Stutsman. "To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-Free Indexing". In: *PVLDB* 11 (2015).
- [67] Berenice Mann. *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>. 2019.
- [68] Linux Programmer's Manual. *futex - fast user-space locking*. <http://man7.org/linux/man-pages/man2/futex.2.html>. 2020.

- [69] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. “Cache craftiness for fast multicore key-value storage.” In: *EuroSys*. 2012.
- [70] Jose F Martinez and Josep Torrellas. “Speculative Locks: Concurrent Execution of Critical Sections in Shared-Memory Multiprocessors”. In: *High Performance Memory Systems*. 2004.
- [71] John M. Mellor-Crummey and Michael L. Scott. “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors”. In: *ACM Trans. Comput. Syst.* 1 (1991), pp. 21–65.
- [72] *MemSQL*. <https://www.memsql.com/>.
- [73] Henrik Mühe, Alfons Kemper, and Thomas Neumann. “Executing Long-Running Transactions in Synchronization-Free Main Memory Database Systems”. In: *CIDR*. 2013.
- [74] *MySQL*. <https://www.mysql.com/>.
- [75] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Tappan Morris. “Phase Reconciliation for Contended In-Memory Transactions”. In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. 2014, pp. 511–524.
- [76] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware”. In: *PVLDB* 9 (2011).
- [77] Thomas Neumann and Michael J. Freitag. “Umbra: A Disk-Based System with In-Memory Performance”. In: *CIDR*. 2020.
- [78] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. “Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems”. In: *SIGMOD*. 2015.
- [79] *NuoDB*. <http://www.nuodb.com/>.
- [80] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, et al. “VSync: push-button verification and optimization for synchronization primitives on weak memory models”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 530–545.
- [81] *Oracle*. <https://www.oracle.com/database/>.

- [82] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. “FPtree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory”. In: *SIGMOD*. 2016.
- [83] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. “Hybrid Transactional/Analytical Processing: A Survey”. In: *SIGMOD*. Chicago, Illinois, USA, 2017.
- [84] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. “{Application-Informed} Kernel Synchronization Primitives”. In: *OSDI 22*. 2022.
- [85] Jignesh M Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. “Quickstep: A data platform based on the scaling-up approach”. In: *PVLDB 6* (2018).
- [86] *Peloton*. <https://pelotondb.io/>.
- [87] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquin, and Donald Kossmann. “Fast Scans on Key-Value Stores”. In: *PVLDB 11* (2017), pp. 1526–1537.
- [88] Filip Pizlo. *Locking in WebKit*. <https://webkit.org/blog/6161/locking-in-webkit/>. 2016.
- [89] *PostgreSQL*. <https://www.postgresql.org/>.
- [90] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. “Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling”. In: *TPCTC*. 2014.
- [91] Robin Rehrmann, Carsten Binnig, Alexander Böhm, Kihong Kim, Wolfgang Lehner, and Amr Rizk. “OLTPshare: the case for sharing in OLTP workloads”. In: *PVLDB 12* (2018).
- [92] Colin Reid, Philip A Bernstein, Ming Wu, and Xinhao Yuan. “Optimistic concurrency control by melding trees”. In: *PVLDB 11* (2011).
- [93] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. “Evaluating the Cost of Atomic Operations on Modern Architectures”. In: *International Conference on Parallel Architectures and Compilation, PACT*. 2015.
- [94] Michael L. Scott and William N. Scherer III. “Scalable queue-based spin locks with timeout”. In: *SIGPLAN*. 2001.

- [95] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. “Accelerating Analytical Processing in MVCC using Fine-Granular High-Frequency Virtual Snapshotting”. In: *Proceedings of the 2018 International Conference on Management of Data*. 2018.
- [96] *Microsoft SQL Server*. <https://www.microsoft.com/en-us/sql-server/>.
- [97] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. “The End of an Architectural Era (It’s Time for a Complete Rewrite)”. In: *VLDB*. 2007.
- [98] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. “Contention-aware lock scheduling for transactional databases”. In: *PVLDB* 5 (2018).
- [99] Linus Torvalds. *No nuances, just buggy code*. <https://www.realworldtech.com/forum/?threadid=189711&curpostid=189723>. 2020.
- [100] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. “Speedy transactions in multicore in-memory databases”. In: *SOSP*. 2013.
- [101] Enrique Vallejo, Ramon Beivide, Adrian Cristal, Tim Harris, Fernando Vallejo, Osman Unsal, and Mateo Valero. “Architectural support for fair reader-writer locking”. In: *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 2010, pp. 275–286.
- [102] Tianzheng Wang and Hideaki Kimura. “Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores”. In: *PVLDB* 2 (2016).
- [103] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. “Building a Bw-Tree Takes More Than Just Buzz Words”. In: *SIGMOD*. 2018.
- [104] *Webkit Lock Implementation*. <https://github.com/WebKit/WebKit/blob/fde0f31b8e7849e3eb1bf6c82439b4b9a5d31baf/Source/WTF/wtf/LockAlgorithmInlines.h>.
- [105] Filip Pizlo. *WTF::Lock should be fair eventually*. <https://github.com/WebKit/WebKit/commit/24e899259cf1724af10cb1bf1b8bb740d5b69c4b>. 2016.
- [106] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. “An Empirical Evaluation of In-Memory Multi-Version Concurrency Control”. In: *PVLDB* 7 (2017).

- [107] Lijie Xu, Tian Guo, Wensheng Dou, Wei Wang, and Jun Wei. “An Experimental Evaluation of Garbage Collectors on Big Data Applications”. In: *PVLDB* 1 (Sept. 2018).
- [108] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. “Staring into the abyss: An evaluation of concurrency control with one thousand cores”. In: *PVLDB* 3 (2014).
- [109] Xiangyao Yu, Andrew Pavlo, Daniel Sánchez, and Srinivas Devadas. “TicToc: Time Traveling Optimistic Concurrency Control”. In: *SIGMOD*. 2016.
- [110] Alexander Zuepke and Robert Kaiser. “Deterministic Futexes: Addressing WCET and Bounded Interference Concerns”. In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019, pp. 65–76.