Fast and Robust Transaction Processing on Emerging Hardware

by

Tianzheng Wang

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2018 by Tianzheng Wang
Abstract

Fast and Robust Transaction Processing on Emerging Hardware

Tianzheng Wang
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2018

Database engines must adapt to the underlying hardware for high-performance transaction execution. Conventional database engines are built on the storage-centric architecture assuming small DRAM, limited parallelism, and slow network. Emerging hardware, including byte-addressable non-volatile memory (NVRAM), massively parallel processors, and ultra-fast interconnect, has lead to significant changes in the design and implementation of modern database engines. Modern database engines, despite being optimized for multicore hardware, still have unscalable components and unnecessary data redundancy that hinder performance on future servers with large core counts (up to 1000). Database engines should also be robust against various workloads, including traditional short, write-intensive transactions and emerging long, read-mostly, and high-conflict transactions. Existing database engines fail to address these problems, leaving the vast opportunity of accommodating heterogeneous, read-mostly and high-conflict workloads that are enabled by emerging hardware largely unexplored.

These problems must be tackled to make database engines perform fast and robustly on modern high-end and future hardware. In this thesis, we first remove the unscalable logging bottleneck by utilizing NVRAM. NVRAM allows us to revive distributed logging, a once prohibitive but effective solution to centralized logging bottlenecks. Avoiding unscalable logging is crucial in building fast main-memory databases that can potentially scale up to 1000 cores in future servers, but is not enough for handling the inconvenient workloads robustly. We continue on devising concurrency control mechanisms that handle a wide spectrum of workloads robustly. Finally, we demonstrate how high availability solutions can be built to guarantee both data safety and freshness using emerging hardware and by eliminating unnecessary data redundancy through indirection.

This thesis serves as a toolbox and provides the guidelines for the design and implementation of high-performance and robust database engines and other concurrent systems. The solutions have been evaluated on both mainstream high-end servers and precursors of future hardware that could feature as many as 1000 cores and NVRAM. The techniques are applicable to both modern and future hardware that is yet to come.
Acknowledgements

This thesis would not be possible without the help from many people. First I would like to express my sincere gratitude to my advisor, Ryan Johnson, who took me as his student, introduced me to the field of database systems and generously helped me in every step of the way toward this thesis and in my career. He is both hands-off and hands-on, by letting me explore various topics and my own ideas, and patiently providing hands-on help and attention to allow me to get started on database systems research.

I would like to thank Ippokratis Pandis, who is not my advisor, nor a member of my thesis committee, but has in fact been almost like a co-advisor throughout my graduate study. I guess I am a super lucky guy to have worked with such a great collaborator and mentor.

Hideaki Kimura took me as his intern at Hewlett Packard Labs in the summer and fall of 2015, and was the one who introduced me to the fascinating field of synchronization. He trusted me and generously shared his knowledge in databases and parallel programming. Milind Chabbi also gave me a lot of help on synchronization. Without Hideaki and Milind, I would not have come to know so much more about main-memory databases and parallel programming.

Besides Ryan, I also learned a lot about the theoretical side of database systems from Alan Fekete when working on the serial safety net. I am especially grateful for Alan’s tremendous help on this work and advice in general.

I would also like to thank Justin Levandoski and Paul Larson, working with whom I really started to learn much about database indexes. Both Justin and Paul shared with me a lot of valuable insights about indexing and systems programming.

In addition to being a thesis committee member, Angela Demke Brown also agreed to be my supervisor after Ryan left the department. Pretty much everything I learned about advanced operating systems research is from her. Cristiana Amza not only agreed to join my thesis committee, but also graciously agreed to chair the checkpoint meetings even though she had a tight schedule ahead. I am also grateful to Divyakant Agrawal (UCSB), Bianca Schroeder and Gennady Pekhimenko for being on my external committee. The committee gave me many constructive comments that greatly improved this thesis.

A significant part of my work depends on cutting-edge hardware generously supported by Hewlett Packard Labs, thanks to many people’s help: Hideaki Kimura, Harumi Kuno, Haris Volos, and Robert Chapman. Working with Ippokratis Pandis, Amazon Web Services provided support through AWS credits for my work on replication, which also greatly benefited from the Apt cluster at Utah and CloudLab.

Zili Shao was the one who introduced me to embedded systems and systems research in general when I was an undergrad. He has always motivated me with his vision and enthusiasm. Without him perhaps I would not have continued with graduate studies and had so much fun along the way. I am also grateful to Duo Liu for discussions we had on various topics, research or otherwise.

My graduate studies at Toronto would not have been so interesting without my friends and labmates. Especially, Kang-Nyeon Kim has been a wonderful friend and collaborator both academically and socially, and provided great help to my work. I am thankful to the members of the Systems group at the University of Toronto, Mike (Dai) Qin, Nosayba El-Sayed, James Gleeson, Peter Goodman, Ioan Stefanovici, Sahil Suneja, and Junji Zhi, for their delightful discussions and valuable feedback on my work. Also, I am lucky to have the friendship and support from many wonderful people, to just name a few: Tingliang Guo, Yue Wang, Haijun Xia and Erkang (Eric) Zhu.

Last but not least, I would like to thank my family (and extended family, too!) for their enormous support and encouragement during this journey.
# Contents

Acknowledgements iii  
Contents iv  
List of Tables ix  
List of Figures x  

1 Introduction 1  
1.1 Online Transaction Processing 1  
1.2 The Interplay between Hardware and OLTP Engines 1  
1.2.1 Storage 2  
1.2.2 Compute 2  
1.2.3 Networking 3  
1.3 The Achilles’ Heels of High-Performance OLTP Engines 3  
1.3.1 Unscalable Centralized Logging 4  
1.3.2 Fragile Concurrency Control 4  
1.3.3 Crippled Hot Standbys 5  
1.4 Mitigating the Weaknesses 5  
1.4.1 Eliminating the Logging Bottleneck 5  
1.4.2 Devising Robust Concurrency Control Mechanisms 6  
1.4.3 Making Hot Standbys Fresh 7  
1.5 Contributions and Roadmap 7  

2 Background and Environment 10  
2.1 Transaction Processing Basics 10  
2.1.1 ACID Properties 10  
2.1.2 Isolation Levels 10  
2.1.3 Concurrency Control Protocols 12  
2.1.4 Durability 15  
2.2 Database Engine Architectures 17  
2.2.1 Traditional Disk-based OLTP Engines 17  
2.2.2 Modern Main-Memory OLTP Engines 18  
2.3 Synchronization 19  
2.3.1 Atomic Instructions 19
4.6.1 Hierarchical Dependency Tracking ........................................... 66
4.6.2 Predicates and Phantoms ......................................................... 67
4.7 SSN in Simulation ................................................................. 67
  4.7.1 Simulation Framework ......................................................... 67
  4.7.2 Microbenchmark Description ............................................... 68
  4.7.3 SSN and Other Schemes Under Contention .............................. 69
  4.7.4 Transactions with Varying Write Intensity .............................. 70
  4.7.5 Interference between Readers and Writers ............................. 70
  4.7.6 Writer Abort Rate due to Safe Snapshots .............................. 71
4.8 SSN in Action ................................................................. 72
  4.8.1 Benchmarks ........................................................................... 73
  4.8.2 Experimental Setup ............................................................. 73
  4.8.3 Traditional OLTP Workloads ............................................... 74
  4.8.4 Read-Mostly Transactions ................................................... 75
  4.8.5 Safe Retry and High-Contention Workloads ......................... 77
4.9 Summary and Discussion ....................................................... 79

5 Mostly-Optimistic Concurrency Control ................................... 80
  5.1 Introduction ............................................................................. 80
  5.2 Key Challenges and Principles ............................................... 81
    5.2.1 Impact of Massive Parallelism ........................................... 82
    5.2.2 Issues in Existing Databases .............................................. 83
    5.2.3 Key Principles of MOCC .................................................... 84
  5.3 Mostly Optimistic Concurrency Control .................................. 84
    5.3.1 Keeping Decentralized OCC’s Benefits ................................ 84
    5.3.2 Temperature Statistics ...................................................... 86
    5.3.3 MOCC Protocols ............................................................... 86
    5.3.4 Discussion ......................................................................... 90
  5.4 MOCC Queuing Lock ............................................................ 91
    5.4.1 The Need for Native, Queue-based Locks ............................ 91
    5.4.2 MQL Algorithms ............................................................... 92
    5.4.3 Instant-Try Interfaces ......................................................... 99
    5.4.4 Implementation ............................................................... 101
    5.4.5 Using MQL in MOCC ......................................................... 103
  5.5 Evaluation ............................................................................. 104
    5.5.1 Setup .............................................................................. 105
    5.5.2 TPC-C: Low Contention, Low Conflict ............................ 106
    5.5.3 High Contention, No Conflict YCSB ................................. 107
    5.5.4 High Contention, High Conflict YCSB ............................... 108
    5.5.5 Varying Contention YCSB ................................................. 109
    5.5.6 Multi-table, Shifting Workloads ....................................... 110
    5.5.7 Long Scan Workloads ...................................................... 111
  5.6 Summary and Discussion .................................................... 112
6 MCS Locks that Welcome Guests

6.1 Introduction ......................................................... 113
  6.1.1 MCS Adoption Challenges and Key Observations .................... 115
  6.1.2 Summary .......................................................... 116

6.2 Desiderata ............................................................ 117
  6.2.1 High Scalability .................................................. 117
  6.2.2 Simplicity, Applicability, and Pluggability ......................... 117

6.3 Basic MCSg Locks .................................................... 118
  6.3.1 Guests ............................................................ 119
  6.3.2 Regular Users ..................................................... 120
  6.3.3 Key Properties of MCSg .......................................... 120

6.4 MCSg++ Extensions .................................................. 121
  6.4.1 Issues with MCSg ................................................ 121
  6.4.2 Guaranteed Guest Lock Acquisition ................................ 122
  6.4.3 Reducing Non-FIFO Behaviors .................................... 126
  6.4.4 Discussion ........................................................ 127

6.5 Evaluation ............................................................ 128
  6.5.1 Setup .............................................................. 128
  6.5.2 Maintaining MCS’s Scalability .................................... 129
  6.5.3 MCSg as a MCS’s Drop-in Replacement for MCS ..................... 131
  6.5.4 Getting More Fairness with MCSg++ ................................ 132

6.6 Summary and Discussion ............................................. 135

7 Fast and Fresh Hot Standbys ........................................ 137

7.1 Introduction .......................................................... 137
  7.1.1 The Freshness Gap ............................................... 138
  7.1.2 Query Fresh ....................................................... 138
  7.1.3 Contributions and Chapter Organization .......................... 139

7.2 Background ............................................................ 139
  7.2.1 System Model and Assumptions .................................... 139
  7.2.2 RDMA over Fast Networks ....................................... 140
  7.2.3 RDMA over NVRAM ................................................ 140

7.3 Query Fresh ............................................................ 141
  7.3.1 RDMA and NVRAM based Log Shipping ............................. 141
  7.3.2 Replay Pipelining ................................................ 143
  7.3.3 Basic Append-Only Storage ...................................... 144
  7.3.4 Append-Only Storage for Log Shipping ............................ 145
  7.3.5 Discussion ........................................................ 148

7.4 Evaluation ............................................................ 149
  7.4.1 Implementation .................................................... 150
  7.4.2 Hardware .......................................................... 150
  7.4.3 Experimental Setup .............................................. 151
  7.4.4 Network Bandwidth and Log Data Rate ............................ 152
  7.4.5 End-to-End Comparisons ........................................ 152
7.4.6 Effect of Replay Policy .................................................. 155
7.4.7 Effect of Indirection ....................................................... 155
7.4.8 Bootstrapping .......................................................... 156
7.4.9 Commit Latency ......................................................... 157
7.4.10 Persistence Delay ......................................................... 157
7.5 Summary ................................................................. 158

8 Related Work ............................................................... 160
8.1 Logging ................................................................. 160
  8.1.1 Utilizing NVRAM ...................................................... 160
  8.1.2 Optimizing for Parallel Processors .................................. 160
8.2 Concurrency Control .................................................... 161
  8.2.1 Pessimistic Locking .................................................. 161
  8.2.2 Optimistic Concurrency Control ..................................... 161
  8.2.3 Hybrid Approaches .................................................. 162
  8.2.4 Multi-Version Concurrency Control .................................. 162
  8.2.5 Pre-Analysis for Serializable Execution ............................... 163
  8.2.6 Separating Concurrency Control and Transaction Execution ....... 163
8.3 Synchronization Primitives .............................................. 163
  8.3.1 CLH Lock .......................................................... 163
  8.3.2 Advances in MCS and CLH Locks .................................... 163
  8.3.3 Queuing Locks that Support Guests .................................... 164
  8.3.4 Lock Cohorting .................................................... 164
8.4 Primary-Backup Replication ........................................... 165
  8.4.1 Utilizing Fast Networks ............................................... 165
  8.4.2 Append-only Storage and Indirection ................................. 165
  8.4.3 Utilizing Logging Techniques .......................................... 166

9 Concluding Remarks ......................................................... 167

Bibliography ................................................................. 168
# List of Tables

3.1 Comparing page-level and transaction-level log space partitioning. ........................................ 34

4.1 Metadata required by SSN. ........................................................................................................ 55

4.2 Lock modes and their commit actions. .................................................................................... 66

4.3 Throughput (TPS) of individual TPC-EH transactions under multi-version CC schemes with 60 threads. ............................................................................................................. 76

5.1 Possible and initial values of MQL queue node fields. ............................................................ 93

5.2 Using MQL in MOCC. ............................................................................................................ 104

5.3 Hardware for Experiments. ..................................................................................................... 105

5.4 TPC-C throughput (low contention, low conflict) on GryphonHawk. MOCC behaves like OCC. PCC has a moderate overhead for read locks. ERMIA is slower due to frequent and costly interthread communication. ................................................................. 106

5.5 Long Scan Workload on GryphonHawk. ................................................................................ 111

6.1 Throughput of TPC-C’s Payment transaction under high contention (one warehouse, 192 threads). MCSg achieves the same performance as MCS yet provides a standard interface for guests. ......................................................................................................................... 131

6.2 Microbenchmark comparison of MCS, MCSg, MCSg++ and C-MCSg-MCS with 240 regular users (no guests). ........................................................................................................... 132

7.1 Throughput and log data rate of a standalone server, vs. the primary’s throughput in a dual-node cluster under 56Gbps Infiniband RDMA and 10Gbps Ethernet TCP/IP. .... 152
List of Figures

2.1 A typical dual-node, hot standby system using log shipping. The primary persists and ships log records through the network to the backup, which replays the log and serves read-only queries. ................................................................. 16

2.2 The architecture of a conventional disk-based database engine. Data is assumed to be disk-resident and cached in a buffer pool. If the requested data is not cached in the buffer pool, the worker thread typically blocks on the I/O request that fetches the desired data from disk. Worker threads must coordinate with centralized lock and log managers for concurrency control and durability, respectively. ............................................. 17

3.1 Time breakdown of running the Update Location transaction of TATP on Shore-MT, a modern OLTP engine designed for multicores. Log contention is a major source of overhead, especially so under high parallelism. .................................................. 28

3.2 The processor affinity problem under distributed logging. (a) Threads may cross NUMA node boundaries to insert log records. (b) Threads only insert to the log that is local to its running processor. ............................................................... 31

3.3 Protecting committed work with passive group commit. The passive group commit daemon keeps dequeuing committed transactions with GSN ≤ the minimum dgsn and signaling stragglers to make progress. ................................................................. 36

3.4 Scalability of distributed logging with write-intensive workloads. Distributed logging scales better as load increases. Transaction-level partitioning performs better due to local memory access. ................................................................. 40

3.5 Time breakdowns for write-intensive transactions with transaction-level log partitioning. 40

3.6 Scalability of distributed logging with more realistic TATP (left) and TPC-C (right) workloads (transaction mixes). ................................................................. 41

3.7 Time breakdown for TPC-C full mix with TX-V. ........................................ 41

3.8 Scalability of distributed logging with passive group commit. PG-V cannot scale after 32 threads due to single-node log buffer allocation, but PG-V with malloc shows similar scalability to TX-P. ................................................................. 42

4.1 Relative merits of existing CC schemes (solid dots) vs. the serial safety net (hollow dots). 45

4.2 A pictorial motivation and description of SSN. Subsets of the dependency graph are shown in a serial-temporal layout where forward and back edges always have positive and negative slopes, respectively. ................................................................. 51
4.3 SSN allows all schedules (a–e) that do not have “peaks,” and also “peaks” where no predecessor of $T$ violates the exclusion window. SSN rejects only case (f); other schemes tend to reject the “valleys” that arise frequently under MVCC.

4.4 A pathological scenario that (a) will deadlock under 2PL but will be serializable under SI; (b) the resulting serial-temporal plot when $T_3$ commits last under SI and (c) when $T_1$ commits last; (d) the serial-temporal plot that results under RC. The horizontal and vertical axes of (b)–(d) represent dependency and commit orders, respectively.

4.5 The bit positions in the readers bitmap serve as indexes to the centralized transaction table which records details on the transaction that is being run by each thread (we discuss the use of the “last cstamp” field later in Section 4.5.3).

4.6 The effect of contention for single-version (left) and multi-version (right) models, with 30 clients.

4.7 Effect of write-intensive transactions.

4.8 Interference between ten update transactions and a varying number of read-only queries under single (top) and multi (bottom) version models. Both RC+SSN and RCL+SSN suffer very high abort rates for readers.

4.9 Abort rate vs. safe snapshot frequency under workloads of class “S” (left) and “L” (right) that touch roughly 10 and 40 records per transaction, respectively.

4.10 Commit throughput of TPC-C (left), TPC-CC (middle), and throughput breakdown of TPC-CC running at 60 threads (right). Under low contention (TPC-C), OCC outperforms all the other schemes. The gap between OCC and SSI/SSN shrinks with more contention (TPC-CC). OCC favors write-intensive transactions; other schemes’ profiles are similar to the Ideal.

4.11 Commit throughput of the TPC-EH benchmark for the full mix (left) and the AssetEval transaction.

4.12 Throughput breakdown of the TPC-EH benchmark under 60 threads.

4.13 Commit (left) and abort (right) rates of TPC-CC. The number of warehouses is fixed to the number of worker threads. Aborted transactions are retried.

4.14 Abort rate of TPC-CC with retry due to SSI/SSN certification failures.

4.15 TPC-CC throughput (left) and abort rate due to certification failures (right) with fixed database size (15 warehouses). Aborted transactions are retried.

4.16 Commit ratio of TPC-CC with 15 warehouses running at 60 threads when aborted transactions are retried.

5.1 Read-only, highly contended YCSB. OCC scales better than 2PL on a deep memory hierarchy, which augments every single scalability issue to an unprecedented degree.

5.2 Read-write, highly contended YCSB. OCC is vulnerable to aborts under high conflict. In an extreme case, it makes almost no progress.

5.3 MOCC overview. MOCC is still an OCC-based protocol that inherits its designs in read/write sets and transaction-private log. In addition, in each page MOCC maintains an additional temperature and uses locking and OCC for hot and cold records, respectively.

5.4 MQL data structures. Requesters form a doubly-linked list of $q$nodes to handle cancellation.

5.5 The two principal cases a reader requester should handle if it has a reader predecessor.
5.6 Throughput of a read-only YCSB workload with high contention and no conflict on four machines with different scales. MOCC adds no overhead to FOEDUS (OCC), performing orders of magnitude faster than the other CC schemes. 107

5.7 Read-write YCSB on GryphonHawk with high contention and/or high conflict. MOCC achieves dramatically lower abort ratio than OCC and even PCC, while maintaining robust and the highest throughput for all cases. 108

5.8 Low contention YCSB (1M records) on GryphonHawk. MOCC/OCC/PCC all scale well and behave similarly. Orthrus is an order of magnitude slower due to the communication between CC and executor threads. 109

5.9 Varying-contention YCSB on GryphonHawk with 8 reads and 2 RMWs. 109

5.10 Commit throughput (left) and abort ratio (right) of the multi-table, shifting workload using MOCC on GryphonHawk. The workload changes its access pattern on a contended table every 0.1 second. Larger thresholds take longer to learn new hot spots. 110

6.1 Not all lock users are created equal. Frequent lock users need high scalability while occasional “guests” need context-free locking, rather than the best performance. MCSg satisfies both users. 114

6.2 An example of (rare) FIFO order violation that could happen under MCSg. Note that FIFO ordering within each group is preserved. 122

6.3 Lock throughput without guests. MCSg and MCSg++ keep the scalability of MCS. K42-MCS sacrifices some scalability due to its use of CAS; CLH’s node inheritance design puts more pressure on the interconnect, making it less scalable than MCS. 130

6.4 Lock throughput with 223 regular users and one guest for locks that support guests. The numbers for MCS and CLH with 224 regular users are used as a baseline. 131

6.5 Average latency for a guest to acquire the lock when running at 223 regular users and one guest. MCSg starves guests, while MCSg+ and MCSg++ significantly improves guests’ fairness. 133

6.6 Total throughput when running at a varying number of guests (224 users in total). MCSg favors regular users and starves guests, thus providing stable performance. K42-MCS and K42-MCS-TLS also show stable performance because they treat all users equally. MCSg+ and MCSg++ sacrifice total throughput to give guests fairness. 134

6.7 Guest throughput when running at a varying number of guests with a total number of users of 224. MCSg++ performs consistently better than MCSg+, because ticketing causes more bias toward guests; regular users are “throttled” by waiting for turns. 134

6.8 Regular user throughput when running at a varying number of guests. The number of total users is fixed to 224. The numbers for MCSg+ and MCSg++ drops faster than K42-MCS and CLH families due to their bias toward guests. Compared to MCSg+, MCSg++’s is more biased toward guests because with ticketing, regular users are “throttled” by waiting for turns. 135

6.9 The number of non-FIFO behaviors with 224 users in total. Ticketing in MCSg++ can reduce up to 70% of non-FIFO behaviors among regular user groups. 135
7.1 RDMA-based log shipping (primary server). (a) Steps to ship a batch of log records from the log buffer. Network I/O (2a) overlaps with local I/O (2b). Step 3 ensures durability on the remote side. (b) Using multi-buffering to reduce log buffer wait time.

7.2 Append-only storage with indirection on single-node systems. Each table employs an indirection array indexed by RIDs. Indexes map keys to RIDs; updates only modify the RID entry in the indirection array, no index operations are needed.

7.3 Append-only storage with indirection for log shipping. An additional replay array that is indexed by the same RIDs as the data array, stores the latest physical address of each record.

7.4 Relative merits of hot standby solutions. (a) Synchronous log shipping sacrifices performance and freshness in a slow network. (b) Asynchronous log shipping trades safety for performance. (c) Logical logging reduces network traffic, but is tightly coupled with concurrency control and trades utilization for freshness. (d) Query Fresh strikes a balance on all aspects.

7.5 Primary throughput. Sync bottlenecks on the network. Query Fresh supports up to 4–5 synchronous replicas. Async and Logical keep high performance but sacrifices safety and has limited applicability/resource utilization, respectively.

7.6 Backup freshness under 75% resource utilization (12 workers, 4 replay threads). Query Fresh keeps up with the primary (≥ 99% freshness). Others must trade utilization for more freshness.

7.7 Throughput of the primary (top) and backups running read-only transactions (bottom).

7.8 Throughput of the primary under pipelined (top) and synchronous replay policies.

7.9 Throughput of the primary with and without indirection.

7.10 Commit latency on the primary. Query Fresh with replay pipelining incurs negligible overhead over the standalone case as it hides network I/O and replay behind local I/O on the primary.

7.11 Throughput of the primary (top) and backups (bottom) with varying NVRAM delays.
Chapter 1

Introduction

As an enabling technology, transactions are ubiquitous in various systems, such as banking, telecommunication, and e-commerce. A transaction consists of multiple read and write operations that access records stored in a database. Processing transactions efficiently requires close interactions between software and hardware, and is the focus of this thesis. In the rest of this chapter, we first discuss the concept of online transaction processing (OLTP). We then discuss the roles hardware plays in building OLTP systems, along with the weaknesses of existing systems, as well as an overview of solutions proposed in this thesis.

1.1 Online Transaction Processing

Central to the transaction concept is the all-or-nothing property [59]: from the end user’s perspective, read and write operations belonging to the same transaction either happen as if all of them were done atomically, or not at all. For example, a funds transfer between two bank accounts must not end in a state where the funds are debited in the sender’s account, but never credited in the recipient’s account. Instead, the transaction should either commit or abort: the former means funds are correctly deducted in the sender’s account and credited in the recipient’s account, while the latter means the balances of both accounts remain the same as if the transaction never happened (unless some other transactions operated on these accounts). In addition, any committed changes to the database need to be durably stored in persistent storage. The database should recover from a crash or power failure to a previous consistent state without data corruption. Internally, these are guaranteed by an underlying online transaction processing (OLTP) engine\(^1\) which enforces the atomicity, consistency, isolation, and durability (ACID) properties [61]. In other words, the OLTP engine is designed to transform the database from one consistent state to another, and does so with high throughput (in terms of the number of transactions it processes in unit time) and low latency (i.e., the duration spent on processing each transaction).

1.2 The Interplay between Hardware and OLTP Engines

Getting high performance for transaction processing is no trivial task. The key is to efficiently handle concurrent transaction requests by fully utilizing the underlying hardware, including units for storage, compute and networking. These hardware components change rapidly, and have had significant impact

\(^1\) Also known as database engines or OLTP database systems. We use these terms interchangeably throughout this thesis.
on the design of database engines. Next, we give a high level overview of how recent database engine design has evolved as the hardware for compute, storage, and networking changes.

1.2.1 Storage

Conventional database engines employ a storage-centric architecture: data is disk-resident and cached in (usually much smaller) DRAM \[8,143\]. During forward processing, worker threads will block on I/O unless the requested data already resides in main memory. When a transaction commits, its log records must be flushed from the log buffer to disk before the worker thread can return the result (transaction committed or aborted) to clients; it is important to interleave threads for other tasks while I/O is in progress. Advances in storage media and software optimizations have largely eased or eliminated I/O as a bottleneck. For example, solid state drives and proper RAID settings can provide orders of magnitude better performance than a single hard disk. However, these devices’ random write performance is still not comparable to DRAM’s. One must carefully orchestrate the interactions with storage to achieve good I/O performance and avoid storage being a bottleneck \[77\]. Thus, many database engines employ group commit \[52\] which instead of forcing log records to disk at each transaction commit, flushes log records generated by multiple transactions in large chunks periodically. This avoids slow, random writes, and utilizes the fast, sequential write performance provided by modern storage devices.

Upcoming byte-addressable, non-volatile memory (NVRAM) technologies such as Intel 3D XPoint \[39\], phase change memory (PCM) \[180\], memristor \[157\], and STT-RAM \[66\] are blurring the distinction between memory and storage. They are both byte-addressable like DRAM, and durable like storage (e.g., disks and flash memory). They also promise close-to-DRAM performance, although some of them (e.g., PCM) have limited endurance that needs wear leveling done by the hardware or software to prevent early wear-outs. For database systems, NVRAM is an attractive candidate for various uses, such as logging \[48,51,174\] and building all-NVRAM single-level stores \[135\] that leverage the potential large capacity and durability of NVRAM.

1.2.2 Compute

Modern database servers are usually equipped with multicore processors (and in many cases, multiple sockets in a single machine) that provide high parallelism. Conventional database engines, however, by following the storage-centric architecture, are designed for processors with very limited parallelism. As a result, most database engines consist of centralized components built with disk-friendly data structures, each of which serves a specific role and interacts with other components. The rise of multicore processors has exposed various bottlenecks in conventional database engines, such as contention in the lock and log managers \[63\]. This has lead to the development of various recent high-performance database engines, by adapting conventional engines for multicores \[75–78,137,138\]. The key is to remove or reduce hot spots and bound the amount of communications and code that has to be run in a critical section so that they would not harm overall performance.

As DRAM price continues to drop, it is increasingly common for database servers to install a large amount of main memory, in the range of hundreds of gigabytes or terabytes. Main memory in modern database servers is often large enough to fit the whole database (or at least the working set), so a worker thread can execute a transaction without any I/O interruption. Taking advantage of large main memory and the high parallelism provided by modern processors, recent main-memory database
systems [42, 85, 89, 90, 102, 169] use memory-friendly (contrary to disk-friendly), lock-free (instead of latch-based) data structures, and ensure I/O is completely out of the critical path [40]. This has also lead to the adoption of lightweight concurrency control methods (notably optimistic concurrency control [93]), instead of (relatively heavy-weight) two-phase locking [46], to reduce whatever cost the database engine itself may add in addition to executing transaction logics. The result is a significant reduction in transaction latency and improvement in throughput.

Core count in high-end modern servers already ranges into the hundreds [64, 154]. Future servers will be equipped with even more cores (up to 1000) [65]. We expect the trend of adopting main-memory databases with high parallelism will continue in the future. Concurrency control algorithms will continue to be the key to utilizing such high parallelism efficiently in handling a wide spectrum of workloads.

1.2.3 Networking

To achieve high availability, database systems often need to replicate data among multiple nodes via the network. For decades distributed and highly-available database systems have been built around the assumption that network is the slowest part [10]. Especially, compared to the high memory bandwidth and therefore the high speed at which modern main-memory systems generate data, traditional Ethernet with 1–10Gbps bandwidth is far from enough to allow data to flow from one node to another without affecting performance. With such slow networks, database systems usually avoid as much communications between nodes as possible to optimize performance. Many distributed systems are sharded so that different nodes host different data and most transactions can be done within a single node.

One of the most popular approaches to high availability is log shipping: all writes are routed to a single primary node, which upon transaction commit, will propagate log records to secondary, standby nodes. Despite its simplicity, log shipping incurs extra delays on the critical path: to maintain strong safety, in a two-safe [61] system the primary cannot commit a transaction until standby servers have persisted the log records. Instead of shipping physical log data, deterministic databases [147,164,165] replicate transactional input and replay transactions in a deterministic manner on replicas to maintain data consistency. However, these solutions usually are tightly coupled with the underlying concurrency control mechanism and require well-defined transaction logic in advance using stored procedures, making it hard to implement and handle data-dependent/cursor-based transactions.

Database servers have recently started to adopt fast networks (such as InfiniBand and Converged Ethernet) that were only used in high-performance computing scenarios. In particular, InfiniBand is matching DRAM bandwidth [22], making data transfer between nodes no longer a major bottleneck and opening up new opportunities in designing highly available database systems.

1.3 The Achilles’ Heels of High-Performance OLTP Engines

Modern database engines have been able to achieve high performance through various optimizations tailored for modern hardware. But they are not perfect. First, many systems have left logging as a potential bottleneck for future massively parallel processors (although they perform well with moderate parallelism). Second, although modern main-memory systems provide high throughput, they do not perform robustly enough against a wide spectrum of workloads, such as high contention workloads and read-mostly transactions that feature long, analytical components. Ironically, read-mostly workloads are made practical by the availability of large main memory and parallel processors in the first place. Finally,
the combination of traditional slow network and fast data generation by main-memory databases makes it particularly hard to keep replicas fresh in hot standby settings that involve multiple nodes. We now discuss each of these shortcomings in detail.

1.3.1 Unscalable Centralized Logging

The log is naturally a centralized figure in virtually all database systems. It provides global ordering and a consistent view of the data stored in the database. A centralized log can easily become a major bottleneck, especially for machines with high parallelism. Although the footprints of concurrent threads may not overlap, they all have to compete for reserving space in the log buffer (sometimes multiple times per transaction), making the log buffer a hot spot. The most natural, immediate solution is to decentralize the log, which typically requires complex dependency tracking and frequent I/Os [174], voiding the efforts made by software optimizations that move logging I/O out of the critical path [77]. Thus, even most multicore-optimized systems opt for centralized logging with various software optimizations that keep logging out of the critical path [41, 42, 52, 77]. As database servers continue to evolve with higher parallelism and multi-socket hardware, centralized logging can easily become a serious bottleneck. Even on today’s high-end multi-socket servers with far fewer than 100 cores—not to mention future servers with higher parallelism—state-of-the-art centralized logging does not scale well, often not beyond a single socket [47, 78, 174]. Therefore, to truly scale database engines for future hardware with massive parallelism, logging must be made scalable.

1.3.2 Fragile Concurrency Control

Massively parallel processors and large DRAM have lead to various modern main-memory databases [42, 85, 87, 90, 102, 169]. A common design principle in these systems is that concurrency control (CC) should be as lightweight as possible because I/O is out of the critical path. CC should not block or employ data structures that are relatively heavy-weight in a main-memory environment. Components such as centralized lock tables should be avoided. A representative of such CC schemes is optimistic concurrency control (OCC) [93]. Under OCC, reads proceed without locking at all, and a transaction can commit as long as the validation phase found its read set remained intact.

Despite the lightweight nature of OCC, it is not a versatile choice: main-memory database engines require CC to be not only lightweight and fast, but more importantly, also robust. Backed by large memory capacity and massive parallelism, OLTP workloads are evolving to have more analytical components [74]. This trend is at least reflected in more recent standard benchmarks, such as TPC-E [168]. The abundant memory capacity often allows fitting the database in a single server, making it viable to run complex read-mostly transactions that feature significant amounts of reads but are not read-only. OCC is vulnerable to such read-mostly transactions, as it must verify that the whole read set of a committing transaction is not modified by other concurrent transactions. Classic optimizations for read-only workloads, such as running read-only queries on a consistent snapshot, are not applicable to read-mostly transactions. OCC is also highly vulnerable to high-contention workloads in which concurrent transactions’ footprints often overlap. Experiments have shown that under OCC, committing high-contention and read-mostly transactions are hard, sometimes almost impossible [89, 179]. Therefore, although previous main-memory database systems can provide high performance, they do so only for the “right” workloads, namely traditional OLTP that features mostly short reads and writes. We argue that in main-memory OLTP
engines, CC should be both fast and robust. That is, the CC scheme should provide high-performance for a wide spectrum of workloads, including traditional, high-contention, and read-mostly workloads.

1.3.3 Crippled Hot Standbys

Log shipping based hot standby solutions are a popular setup for high-availability in many database systems [67, 133, 134, 144, 161]. The system consists of a primary server that serves read/write transactions and one or more standby servers for read-only queries such as analytical tasks. The primary continuously ships log records of committed transactions to standby servers. Upon failure, a standby server can take over and become the new primary server, preferably in a short time. Modern databases generate data in a faster than ever speed. With slow network and unnecessary data redundancy, however, today’s hot standby solutions hardly fulfill their original, ideal goals of providing both safety guarantees and fresh data access [61]: the former requires data be durably replicated before commit, while the latter means queries running on replicas should be able to see the most recent updates happened on the primary server. Maintaining safety and freshness, however, often significantly lowers performance, for two reasons. First, network bandwidth is often not enough to transfer large amounts of log data to replicas. Second, more importantly, existing systems follow a dual-copy architecture, in which data is first stored in a durable log, and then to the “real” data store known as “the database”. Such design mandates the need for fully replaying log records on standby servers before data is available for analytical queries. Worse, in many systems this replay process is serial [184], much slower than data generation and arrival speeds.

To preserve performance, many hot standby solutions ship log records to backup nodes asynchronously, after transactions are committed on the primary node. This avoids the network bottleneck, but makes the standby servers stale and risks losing committed work that standby servers are yet to receive. Read-only queries routed to standby servers can only use whatever data is available on the standby server, limiting the freshness of analytical queries.

1.4 Mitigating the Weaknesses

We have summarized the major shortcomings of modern high-performance database systems: unscalable centralized logging, fragile concurrency control that does not handle the inconvenient workloads, and stale, unsafe hot standby solutions due to the dichotomy between high data generation and low network speeds, as well as data redundancy. Advances in hardware require database systems evolve with re-designs, and enable solutions to the weaknesses we have identified.

1.4.1 Eliminating the Logging Bottleneck

The non-volatility and byte-addressability of NVRAM open up opportunities to remove the logging bottleneck fundamentally, because it voids the “flush-before-commit” requirement found in existing OLTP engines. Log records are durable once they are written in NVRAM, which can be attached to the memory bus, side-by-side with DRAM and accessed via load and store instructions. An NVRAM-based log buffer alone in fact offers little help for centralized logging, other than reduced latency. However, it enables distributed logging which was prohibitively slow as committing a transaction often requires multiple I/Os to flush different log buffers on which the transaction has dependencies. However, a distributed log that uses NVRAM-backed log buffers will not run into this excessive I/O problem.
Meanwhile, distributing the log naturally resolves the centralized logging bottleneck. The only caveat for this approach to work is to guarantee data is correctly persisted in NVRAM upon transaction commit: modern processors employ multiple levels of volatile caches, so accesses to NVRAM on the memory bus will likely be cached as well. A crash before the data is persisted in NVRAM will risk losing committed work, unless the processor caches are also persistent, which is being explored, but unlikely to be adopted by the industry soon [128, 132, 158, 172, 193]. To solve this problem, we develop passive group commit, a lightweight mechanism that piggybacks on the existing group/pipelined commit [77] machinery to protect committed work. With NVRAM-based distributed logging, we have observed $\sim3\times$ performance improvement on a modern database prototype optimized for multicores [174]. This paves ways for designing high-performance main-memory OLTP engines without the logging bottleneck.

1.4.2 Devising Robust Concurrency Control Mechanisms

Handling read-mostly and high-contention workloads requires a departure from pure OCC. In particular, long reads in read-mostly transactions are the “right” workload for multi-versioned concurrency control (MVCC) [18], such as snapshot isolation (SI) [14]. With MVCC/SI, readers and writers do not block each other, thus allowing long read-mostly transactions to commit easily. However, these approaches by default are usually not serializable, even for certain read-only transactions [50]. Existing systems that employ MVCC often guarantee serializability by rejecting all overwrites like OCC does—the transaction must abort if any of its reads is updated by a concurrent transaction [42,102]. Such optimistic MVCC schemes void many valid serializable schedules and make read-mostly transactions almost equally hard to commit as OCC does. Serializable snapshot isolation (SSI) [25] allows more valid schedules than optimistic MVCC, but is not friendly to read-mostly workloads: both approaches must track the whole read footprint of each transaction, spending a considerable amount of CPU cycles on validating read sets at commit time, thus limiting throughput.

We devise the serial safety net (SSN), a serializability certifier that can be overlaid on top of any concurrency control mechanism that is as strong as read committed, including various MVCC schemes. SSN is lightweight and can be implemented in a lock-free manner that fits the paradigm of modern main-memory database systems. The gist of SSN is to track the immediate dependencies of a committing transaction in the form of a pair of easily computed timestamps and abort the transaction if committing it would lead to non-serializable schedules. SSN robustly handles a wide variety of workloads, including traditional OLTP and emerging read-mostly transactions. Because of its conciseness, SSN can guarantee serializability without having to track full read sets for read-mostly transactions, saving precious CPU cycles on real work.

As we have discussed in Section 1.3.2, modern OCC-based systems also lack robustness for workloads with high contention where certain “hot” records are updated by many concurrent transactions, especially so for future servers with massive parallelism. This pattern is common in many scenarios, such as reservation services for popular destinations and campaigns in e-commerce. Although SSN handles a wide variety of workloads well, it requires frequent interthread communication, which is expensive for large scale multi-socket and future 1000-core machines. We solve this problem with a fundamental observation that pessimistic locking, despite being more heavyweight, provides more robust and better performance under high contention than OCC does [8]. Therefore, we combine the best of both worlds and propose mostly-optimistic concurrency control (MOCC), a robust CC method that uses pessimistic locking for hot records, while still maintaining all the benefits of OCC for the rest of the data. The major challenges
are judiciously taking locks for hot records and devising an efficient locking mechanisms that suit modern and future parallel hardware, as well as making locking and OCC co-exist peacefully in the same system. We solve these problems by using approximate counters [127] to guide the lock-taking decisions. In addition, we design the MOCC queuing lock (MQL) for efficient locking. MQL combines strengths of existing MCS locks [119,120,153] to realize an efficient queue-based, cancellable reader-writer lock that suits main-memory database engines. With MQL, MOCC can acquire, release and re-acquire locks upon read accesses to prevent hot records from being clobbered by concurrent transactions. The result is a CC method that robustly handles both low-contention OLTP and high-contention/read-mostly workloads.

### 1.4.3 Making Hot Standbys Fresh

Fast networks (such as InfiniBand), remote direct memory access (RDMA) and NVRAM have made it possible to revive synchronous log shipping, arguably the easiest way to build a two-safe [61] system. Coupled with NVRAM that provides fast persistence, the primary server can use RDMA to ship log records stored in the log buffer directly to NVRAM-backed log buffers in backup servers at group commit boundaries, without delaying forward processing. Thus, the combination of RDMA and NVRAM allows us to remove the staleness caused by slow log data transfer and persistence delays.

To tackle the slow replay problem, we advocate an append-only storage architecture in which the log is the database. Worker threads access data tuples through indirection arrays [103,151] that map each record ID (RID) to the tuple’s physical location (in the durable log or main memory). If the tuple under access is not memory-resident, one could employ mechanisms like Anti-Caching [40] to keep I/O out of the critical path. Instead of the physical location (e.g., an offset in the disk or a virtual memory pointer), indexes map keys to RIDs. An RID never changes during the whole lifetime of the tuple it represents, so an update only needs to make sure the corresponding indirection array entry points to the right physical location. This design greatly simplifies and removes excessive data redundancy: many databases employ multiple secondary indexes (i.e., indexes on non-primary keys) for each table, and with indirection arrays, an update operation only needs to update the corresponding indirection array entry, without having to update any index associated with the table. Since the log is the database, upon receiving log records, a standby server only needs to update the indirection array entry to point to the record’s location in the durable log, without fully replaying the log records like in conventional architectures where one has to incarnate a tuple from the log to the “real” database in persistent storage.

We note that the techniques for making standby servers fresh—append-only storage and indirection arrays—are not unique to our system [102,103,151]. However, when combined together with fast networks in a hot standby setting, we are able to achieve a safe and fresh system. Moreover, either technique can be applied to existing systems to avoid data redundancy or reduce data update cost.

### 1.5 Contributions and Roadmap

This thesis makes the following contributions:

- **Chapter 3**: Utilizing fast NVRAM, we devise a practical distributed logging scheme to ease the logging bottleneck. Our approach allows a much simplified logging architecture and scalable performance that neither centralized nor distributed logging was able to provide previously.
Chapter 1. Introduction

- **Chapter 4:** We propose the serial safety net (SSN), a robust and high-performance concurrency control scheme that is suitable for today’s high-end servers. SSN enables emerging heterogeneous workloads that feature read-mostly transactions, thus allowing a much simpler programming model for applications.

- **Chapter 5:** In preparation for future processors with up to 1000 cores, we propose mostly-optimistic concurrency control (MOCC) which judiciously uses pessimistic locking to enhance OCC for high-contention workloads. MOCC scales well on machines with large core counts. Its scalability on a real, 288-core machine with 16 sockets strongly indicate that the design principles are well-suited for future parallel architectures.

- **Chapter 6:** Scalable synchronization primitives—in particular, queue-based spinlocks (e.g., the MCS lock [119])—are the basic building blocks of modern database engines. Unfortunately, such locks often require a non-standard locking interface, making their adoption in existing code bases hard. In the course of attacking weaknesses in concurrency control, we identify the need for queue-based spinlocks that support a standard locking interface. We propose MCSg to solve this problem. MCSg allows various database components to adopt the MCS lock for better performance without large-scale refactoring, a daunting and error-prone process.

- **Chapter 7:** We build Query Fresh, a log shipping system that provides both freshness and strong safety. The crux is an append-only storage architecture that significantly accelerates log replay on backup servers with much less resource compared to traditional approaches, thus leaving more resources for handling read-only queries and improving hardware utilization. Query Fresh leverages fast networks, NVRAM and RDMA for quick log data transfer. Combined with our append-only storage architecture, Query Fresh achieves strong safety and freshness without sacrificing much performance.

This thesis serves as a toolbox and provides guidelines for researchers and practitioners to build fast and robust OLTP engines. Despite the focus on emerging hardware, our solutions are also practical for modern hardware and hardware that already demonstrates the properties of future hardware.

Along with the above contributions, we built ERMIA [89], an open-source main-memory database that is optimized for today’s high-end servers. ERMIA uses SSN to handle various workloads well, especially read-mostly transactions. Query Fresh is built on top of ERMIA, as its hot standby solution. MOCC and MQL, as well as the MCSg lock are integral parts of the FOEDUS open-source database designed for future 1000-core servers. The source code is available at the following locations:

- Scalable NVRAM-based logging: [https://bitbucket.org/tzwang/shore-mt-nvm](https://bitbucket.org/tzwang/shore-mt-nvm).
- SSN, Query Fresh and ERMIA: [https://github.com/ermia-db](https://github.com/ermia-db).
- MOCC, MQL and MCSg: [https://github.com/hewlettpackard/foedus_code](https://github.com/hewlettpackard/foedus_code).

The rest of this thesis is organized as follows. Chapter 2 gives necessary background on transaction processing, the hardware this thesis focuses on, and the internals of OLTP engines. We also give an overview of the performance measuring methodology (including benchmarks) used throughout the thesis.

Chapters 3–7 detail the major contributions of this thesis. We first tackle the centralized logging bottleneck in Chapter 3, which details NVRAM-based distributed logging. We then introduce the serial
safety net in Chapter 4, an efficient concurrency control mechanism for today’s high-end servers with high degrees of parallelism and large main memory. Chapter 5 proposes mostly-optimistic concurrency control, which focuses on future servers with up to 1000 cores. In Chapter 6 we highlight the need for dual-interface MCS locks, and propose the MCSg lock, which eases the adoption of MCS locks in database engines and other systems. Chapter 7 describes details of Query Fresh. Chapter 8 covers related work. Finally, we conclude in Chapter 9.
Chapter 2

Background and Environment

This chapter lays out the necessary background for later chapters. We first start with the basic concepts in transaction processing and then continue with the designs of database engines that realize these logical level concepts. We then introduce the types of hardware that we plan to cover in this thesis. The last part of this chapter describes the experimental environment and the way we measure performance throughout this thesis. Readers already familiar with these topics can skim and/or fast forward to later chapters.

2.1 Transaction Processing Basics

A transaction consists of a series of read and write operations, each of which accesses a target record (also called “tuple”) stored in the database. The database is shared among multiple threads and/or processes, allowing concurrent transaction execution. For correctness, the transaction processing engine must properly schedule concurrent transactions, maintaining the ACID properties [61], described below.

2.1.1 ACID Properties

The ACID properties consist of four properties that must be enforced for each transaction: atomicity, consistency, isolation, and durability. We give a brief introduction to each of them, following the definitions given by Gray and Reuter [61].

• **Atomicity**: All operations (reads and writes) in a transaction must happen, or not at all.

• **Consistency**: A transaction must transform the database from one consistent state to another; it produces consistent results only.

• **Isolation**: A transaction executing concurrently with others must behave exactly as it would if it is the only transaction running in the system.

• **Durability**: Changes made by committed transactions must not be “forgotten” by the system, i.e., they must be durably stored in storage and get recovered correctly upon failures.

2.1.2 Isolation Levels

Among the various components in a database engine (described later), the concurrency control (CC) method is a central piece that is responsible for enforcing the ACID properties. In particular, the isolation
property is often relaxed to provide multiple “isolation levels”, mainly because the application desires more performance than strictness. Below we discuss these various isolation levels [14].

Read Uncommitted

Read Uncommitted is the least strict isolation level and allows a transaction $T_1$ to see updates done by another in-progress transaction $T_2$. Thus the data read by $T_1$ might become invalid in case $T_2$ did not commit, hence leading to wrong/inconsistent execution results. In fact, Read Uncommitted exhibits isolation failure because a transaction might see modifications done by another in-progress (not yet committed) transaction. Such anomalies are usually called *dirty read*.

Read Committed

Being stronger than Read Uncommitted, Read Committed (RC) guarantees each transaction only sees modifications done by committed transactions, i.e., it disallows dirty read. Under Read Committed, *lost update* is not possible, either. That is, the update done by a transaction $T_1$ will not be overwritten by another transaction before $T_1$ is committed or rolled back.

Repeatable Read

Repeatable Read provides all the guarantees Read Committed gives. In addition, it ensures a subsequent point read (i.e., not part of a scan) to the same record by the same transaction remains stable during the transaction’s lifetime (read stability). In other words, no other transaction can modify a tuple that is read by a transaction that is yet to commit or roll back.

Serializable

Repeatable Read guarantees read stability for point reads, i.e., reads that target individual records, but does not guarantee read stability for scans that target a range of keys. Thus, in the presence of concurrent inserts, under Repeatable Read, a second scan within the same transaction targeting the same range as the previous one might see additional records, called *phantoms*. Providing read stability for range scans is also called *phantom protection*.

Serializable provides phantom protection, in addition to what is guaranteed by Repeatable Read, i.e., a subsequent scan operation over the same key range will return the same result as before during the lifetime of a transaction. The subsequent scan should not see any new inserts in the specified range.

We refer to these erroneous phenomena (e.g., dirty read, lost update, and phantoms) as *anomalies*. Different concurrency control protocols, depending on their design and implementation, may exhibit different anomalies. Serializable forbids these anomalies and ensures that the result of a concurrent execution is equivalent to *some* serial execution (thus the name “Serializable”). Serializable is usually the most strict isolation level and provides the maximum correctness, however, it often comes with performance penalty. Thus, many database systems default to other isolation levels, e.g., Read Committed.

**Notes on serializability**. The various isolation levels we have discussed so far also coincide with what was defined by the ANSI SQL-92 standard [5]. However, as Berenson et al. [14] pointed out, the ANSI standard does not fully capture the correct (academic) definition of “serializable” (which is the definition we will be using throughout this thesis) [14,143]. In particular, snapshot isolation [14] exhibits various anomalies that can lead to non-serializable executions, but yet is “serializable” according to
the ANSI standard. The reason is that ANSI SQL-92 was devised based on lock-based concurrency control protocols, while in snapshot isolation reads do not take locks. We cover more details on different concurrency control protocols (including snapshot isolation) in the next section.

2.1.3 Concurrency Control Protocols

We have covered different isolation levels, which are in turn supported by the underlying concurrency control protocol. This section gives background on the concurrency control protocols that are of interest in this thesis. Concurrency control protocols are usually either pessimistic or optimistic. The former, often referred to as “pessimistic locking”, assumes conflicts are frequent so to access any record the transaction needs to take a lock that protects the record. The latter, however, assumes conflicts are rare, so locking is postponed until commit time.

Another way to categorize concurrency control methods is by looking at the way they organize data, i.e., multi- or single-versioned (multi-version concurrency control, or MVCC). While it is possible to combine two methods, one from each category (e.g., optimistic MVCC), this thesis focuses on (1) single-version pessimistic locking, (2) single-version optimistic concurrency control (OCC) [93], and (3) snapshot isolation that is multi-versioned.

Pessimistic Locking

Database systems that employ pessimistic locking associate each record with a lock that protects the record from concurrent accesses. Transactions must acquire locks upon record access. A lock can be held in either “shared” or “exclusive” mode. The shared mode allows read-only access and can be acquired by multiple concurrent reader transactions (denoted as “readers”); we call these concurrent reader requests “compatible” because they can be granted together. The exclusive mode is for modification and can be held only by a single writer transaction (denoted as “writer”). Similarly, a shared lock request is said to be “incompatible” with an exclusive lock request. This way, no reader (writer) can access a record that is locked by a writer (multiple readers). So readers and writers will block each other. In many systems a shared lock can be also “upgraded” to an exclusive lock; one way to implement this is to let the requester wait for all other concurrent readers to finish.

Under pessimistic locking, serializability is usually enforced by two-phase locking (2PL) [46]. The basic idea of 2PL is that once a transaction starts to release locks, it cannot acquire any more locks. Interested readers may refer to Eswaran et al. [46] for details. For full serializability, however, phantoms must be prevented. Most systems achieve this by using techniques such as key-range locking [106], which essentially locks the “gap” between two index keys so that any insertion into the target key range will conflict with the existing range lock and block.

Because the database engine has no control over which records (or in what order) will be accessed, deadlock is possible under 2PL. There are several ways to handle deadlocks, e.g., WaitDie, BlindDie and Dreadlock [92]. WaitDie is a traditional deadlock avoidance protocol where a lock-waiting transaction with younger timestamp is immediately aborted. BlindDie is a simplistic variant of WaitDie where the transaction is aborted regardless of timestamp. Dreadlock detects potential deadlocks by maintaining and spreading bitmaps that summarize recursive dependencies between waiting threads. When a thread finds its own fingerprint in the bitmap, Dreadlock predicts a risk of deadlock. Dreadlock might have false positives, but has no false negatives after sufficient cycles of checking and spreading bitmaps.
Optimistic Concurrency Control (OCC)

Unlike 2PL, which assumes conflicts are common (thus the name “pessimistic”), OCC [93] assumes conflicts among transactions are rare, i.e., in most cases transactions’ footprints do not overlap. Thus, it is reasonable to avoid locking when possible. Here we describe a more recent variant of OCC (called decentralized OCC) that has been adopted in many recent main-memory systems [90, 169]. Under decentralized OCC, a transaction is free to read any record without taking locks, and always keeps updates local during forward processing. Meanwhile, the transaction must track its whole footprint in private read and write sets for checks during commit time to guarantee serializability. At commit time, the transaction must (1) take locks for all the records it intends to modify (by examining its write set), and (2) verify the records read (maintained in its read set) remained the same as their initial accesses. If this is not the case, the transaction must abort; otherwise it proceeds to commit by publishing its writes.

A salient feature of OCC is its deadlock-freeness. By the time it needs to acquire locks (i.e., at commit time), the whole write set is known. The committing transaction simply sorts the write set by some consistent order (e.g., unique record IDs or record virtual addresses) and acquires locks one by one. No deadlock will arise as long as all transactions follow the same consistent order.

Snapshot Isolation

Snapshot Isolation [14] is a multi-version concurrency control (MVCC) [17] mechanism that maintains multiple versions for each record. Each version has a timestamp that denotes its creation time, which is also the “commit timestamp” of the transaction that created the version. Timestamps are unique, monotonically increasing numbers. A transaction executing under snapshot isolation begins by acquiring a “begin timestamp” and is only allowed to see versions that were created before its begin timestamp. Versions of the same record are usually chained together by a linked list. An update will append a new version in the version chain, and can only succeed if it can see all the record’s versions. To handle concurrent writes to the same record (write-write conflicts), most designs follow the “first-committer/writer-wins” principle [49], which specifies that a write can only succeed if it is the first generating a new version of a particular record. A reader simply traverses the version chain until it finds the latest version that was created before its begin timestamp. Note that traversing the version chain for a read operation does not require the transaction to take any lock. Thus, reads and writes never block each other. This gives snapshot isolation much performance benefits over other CC schemes that are mostly lock-based.

In terms of isolation level, snapshot isolation is stronger than Read Committed as it disallows non-repeatable read (transactions always read stable, committed versions). It is also free of phantoms as long as the committing transaction’s commit timestamp is larger than any concurrent reader’s begin timestamp [14]. Therefore, following the ANSI SQL-92 standard, it is easy to wrongfully conclude that snapshot isolation provides “serializability”. However, snapshot isolation exhibits anomalies that are not covered by the standard, notably write-skew [14] and the read-only anomaly [50].

Write-skew. The write-skew anomaly can happen with two transactions, each of which modifies a record that was read by the other. As a concrete example, consider two concurrent transactions $T_1$ and $T_2$ and a simple database with two records $A$ and $B$ both with an initial value of 0. Consider the following schedule. For clarity, we also show the result of each statement in the parenthesis that follows.
Chapter 2. Background and Environment

In this example, T1 and T2 began at the same time (Time = 1) with A = B = 0, so they had read the same snapshot. Because A = B = 0, both transactions executed the increment inside the if block shown above and successfully committed. As a result, A = B = 1. However, no serial execution could lead to this result: if T1 runs first, T2 will not be able to increment A as B was already incremented by T1. Similarly, if T2 runs first, T1 will not be able to increment B. In other words, a truly serializable execution will only lead to a result of A = 0, B = 1 or A = 1, B = 0, but never A = B = 1.

Read-only anomaly. Read-only transactions were considered to be always serializable under snapshot isolation, until Fekete et al. [50] discovered that reads conducted at the “wrong” moment could lead to non-serializable execution. The read-only anomaly is best understood via the following example. Consider the following history with three transactions that operated on the balances of a checking and a savings account. Suppose the withdrawal is covered as long as the total balance of both accounts after the withdrawal is at least 0; any overdraft will cause a penalty charge of 1 in the checking account.

<table>
<thead>
<tr>
<th>Time</th>
<th>T1: Begin</th>
<th>T2: Begin</th>
<th>T3 (read-only): Begin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s = Savings (0)</td>
<td>c = Checking (0)</td>
<td>s = Savings (0)</td>
</tr>
<tr>
<td>2</td>
<td>Savings = s + 20</td>
<td>Checking = c - 10 (-11)</td>
<td>Checking = c - 10 (-11)</td>
</tr>
<tr>
<td>3</td>
<td>Commit</td>
<td>s = Savings (20)</td>
<td>s = Savings (20)</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Commit</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let us consider only T1 and T2 first. Since T1 and T2 began at the same time, they operate on the same snapshot. They both successfully committed. The resulting balances are -11 and 20 in the checking and savings accounts, respectively. With only T1 and T2, this is a serializable execution that is equivalent to the serial execution of T2, T1.

Now we add T3 to the picture. T3 is a read-only transaction that only queries account balance and started after T1 committed, but before T2 committed. So the final execution result of this history is (1) total balance being 9 because of an overdraft charge of 1 in T2 and (2) T3 returning a balance of 20. This result, however, cannot happen in any serial execution.

Now we consider possible equivalent serial schedules. Since T3 began after T1 committed, in serial schedules it must be ordered after T1. Then there are three possible interleavings: T1 – T2 – T3, T1 – T3 – T2, and T2 – T1 – T3. With this, we can work out the final total balances for each case and the corresponding total balance seen by T3 (shown in parenthesis): 10 (10), 10 (20), 9 (9). None of these final results match that of the above history: 9 (20).

A large body of work has been done to add serializability to snapshot isolation and remove these anomalies [26,47,105,176]. In addition, some schemes provide safe snapshots [141,176] specifically for read-only transactions. Read-only transactions that run on such safe snapshots are guaranteed to be
serializable, without risk of anomalies. Typically, safe snapshots are passive mechanisms: a read-only transaction trying to take a safe snapshot will wait until all in-flight transactions have ended while monitoring unsafe accesses (i.e., dependencies among transactions) to determine whether its snapshot is safe. We cover related topics in more detail in Chapter 4, and devise the serial safety net (SSN), a cheap, bolt-on certifier that guarantees serializability for not only snapshot isolation, but also weaker models, such as Read Committed.

2.1.4 Durability

Ensuring the durability of committed data is crucial in almost any database system. The most popular solution is ARIES-style write-ahead logging [126]. Moreover, many applications require high availability, i.e., data needs to be replicated on multiple machines, so that once a server fails, another can quickly take over and continue the service.

Write-Ahead Logging

The basic idea of write-ahead logging is that any change to the database must be first made durable in a separate stable storage space, called “the log”, before it can be reflected in the “real” data store. The log consists of log records that describe changes to the database. Logging can be physical, logical, or a combination of both (physiological) [126]. Physical logging stores before and after images, whereas logical logging stores only the necessary information (e.g., the operation and input) needed for re-constructing the new data. Physiological logging uses physical and logical logging for redo and undo, respectively. Transactions record all updates in a single, global log before the dirty pages reach storage. To hide the performance difference between fast memory and slow disk, log records are cached in a centralized DRAM buffer and forced to disk only at transaction commit. The log is usually protected by some synchronization primitive (e.g., a spinlock), which must be obtained by the inserting transaction before writing to the log buffer.

When a transaction commits, its buffered log records must be forced to storage to ensure recoverability. In this thesis, we call it the “flush-before-commit” requirement. Each log record is uniquely identified by a monotonically increasing log sequence number (LSN). An LSN in each page indicates the latest log record that modified the page. During recovery, the log is analyzed and scanned forward to repeat history (“redo”), applying log records with an LSN greater than its targeted page. After redo, loser transactions are rolled back (“undo”). A compensation log record (CLR) is written after each undo operation to make sure log records are undone exactly once in spite of repeated crashes during recovery.

Modern main-memory database systems often use redo-only physical logging [89,90,169] that does not store before images in the log. During forward processing, a transaction holds its log records locally and does not store them in the log buffer until the transaction successfully pre-commits. So the log contains only data generated by committed transactions. Recovery becomes a single redo pass, without analysis and undo in ARIES [126]. As we will show in Chapter 7, redo-only physical logging makes it possible to achieve both freshness and safety (described next) in modern main-memory database systems.

Log Shipping for High Availability

With logging, maintaining high availability is straightforward: one simply needs to replicate and replay the log records generated by one server onto another. This is usually called “log shipping” and is widely
adopted in practice [67, 134, 144, 161]. A log shipping based replication system consists of one single primary server and one or multiple backup servers. Figure 2.1 shows a dual-node example. The single primary server accepts both reads and writes, while backups accept log records transferred from the primary and replay them continuously. To increase hardware utilization and throughput, one could route read-only queries to backups, however, a backup never accepts writes unless it fails over to become the new primary. Log replay transforms log records into “real” database records, and is necessary for serving read-only queries (if any) on backup servers. In case the primary fails, a backup will take over and become the new primary after replaying all log records. Thus, the speed of log replay also affects failover speed. New machines can join the cluster as a backup by requesting a recent checkpoint and the log records generated after the checkpoint from the primary.

Different log shipping strategies could lead to different levels of safety guarantees, i.e., whether committed work could be lost during a failover. The original concept of safety guarantees were given for dual-node pairs. Depending on whether the primary involves the backup at commit time, a dual-node system can be 1-safe, 2-safe, or very safe, as defined by Gray and Reuter [61]:

**Definition 2.1** 1-safe: transactions commit as soon as their log records are durably stored at the primary.

**Definition 2.2** 2-safe: transactions are not committed until their log records have been persisted at both the primary and backup. If the backup is down, the transaction can commit after persisting log records at the primary.

**Definition 2.3** Very safe: same as 2-safe, except that it does not allow transactions to commit if the backup is down.

Under 1-safe, log records are shipped in the background without involving the backup on the commit path (asynchronous log shipping). Therefore, the primary exhibits performance similar to a single-node system’s. Besides providing high primary performance, 1-safe can also be useful for reasons such as preventing human errors from propagating to all servers [144]. However, log records that are not yet shipped to the taking-over backup may appear to be lost, although it was “committed” on the then-active primary. In contrast, very safe and 2-safe involve the backup on the commit path, by not committing a transaction until its log records are persisted in the primary and backup. This often reduces primary performance, but provides stronger safety guarantees than 1-safe because it avoids the lost transactions problem (under 2-safe when the failed backup comes back online, it needs to consult the primary for log generated since its failure). Although these definitions were originally given for dual-node pairs, we can easily extend them to multi-node systems, by requiring log records be persisted in all backups upon
commit in Definitions 2.2 and 2.3. Moreover, one could also make Definition 2.2 more strict (or relax
Definition 2.3) to allow transactions to commit if a majority of nodes have persisted the log [37,134,170].

We say that a system guarantees strong safety if it is 2-safe, very-safe or follows the extended definitions
for multi-node systems. Providing strong safety requires synchronous log shipping, i.e., transferring log
records to backups and ensuring they are persisted in all or a majority of nodes upon commit.

2.2 Database Engine Architectures

The previous sections have discussed on a logical level what functionality a database engine needs to
satisfy. This section covers the actual design and implementation of database engines that employ the
concurrency control and durability mechanisms we have discussed. We begin with disk-based database
engines and then discuss main-memory ones.

2.2.1 Traditional Disk-based OLTP Engines

Most database engines employ a storage-centric architecture, in which the amount of DRAM is usually
much smaller than disk capacity. Data is assumed to be disk-resident [143]. We refer to these engines
as “traditional” and “disk-based”, in contrast to modern main-memory OLTP engines which assume
main memory is large enough to hold at least the working set (or even the whole database). The data
structures and algorithms used to build traditional engines are designed with the goal to handle and hide
the impact of frequent, blocking I/O. Figure 2.2 shows the internals of a traditional OLTP engine. For
clarity, we omit components that are not the focus of this thesis, such as indexes and the query optimizer.

Figure 2.2: The architecture of a conventional disk-based database engine. Data is assumed to be
disk-resident and cached in a buffer pool. If the requested data is not cached in the buffer pool, the worker
threads typically blocks on the I/O request that fetches the desired data from disk. Worker threads must
coordinate with centralized lock and log managers for concurrency control and durability, respectively.

The left side of Figure 2.2 shows the data structures that reside in volatile DRAM. These data
structures are inherently centralized. The lock manager maintains lock tables which are indexed by each
record in the database. Traditional disk-based databases usually employ pessimistic locking, and 2PL is
the dominant concurrency control method for serializability. Under 2PL, each transaction must first lock
the record it wants to access, through the centralized lock manager. The lock table is usually built as a
hash table with locks for records as keys, and each entry is a list of lock requesters [143]. A requester must wait for the current lock holder to release the lock if their requests are incompatible.

A conventional OLTP engine also employs a global ARIES-style log to ensure durability and correct recovery. The log buffer accumulates multiple log records before they are flushed to disk in large chunks, utilizing the relatively good sequential access performance of spinning disks and SSDs. Note that the log buffer is also a centralized component, i.e., the entire system has only one log buffer. Hence, synchronization is usually needed for concurrent accesses from multiple threads/transactions. A transaction trying to insert log records must first acquire the latch\(^1\) that protects the log buffer. It must also release the latch after it has finished log insertion so that another transaction could be allowed to append to the log buffer. To achieve correct recovery and data consistency, a transaction commit result can only be returned to the client when all of its log records are durable, i.e., after the log buffer is flushed. Flushing upon every transaction commit will significantly degrade performance. Therefore, most systems employ group commit\(^2\) to hold transaction commit results until the log buffer is flushed. At the same time, the underlying execution threads can continue to process new incoming transactions.

### 2.2.2 Modern Main-Memory OLTP Engines

The significant price drop in DRAM has allowed database servers to be equipped with hundreds of gigabytes or even terabytes of main memory. Such spacious main memory allows the working set—or even the whole database—to fit in memory. I/O operations, which were a major source of bottleneck, are completely out of the critical path [40]. As a result, worker threads can usually finish executing each transaction without any interruption. Optimizations for multicore and multi-socket hardware also allow much higher parallelism compared to traditional disk-based systems.

Compared to the architecture of conventional engines, data structures in main-memory database engines must be memory-friendly. In disk-based systems, it is important to keep the processor busy as I/O operations happen. The high cost of centralized data structures (e.g., centralized lock tables) are often hidden behind the high I/O cost. In main-memory engines, however, there is not much room to hide such cost. As a result, data structures in main-memory systems are usually lock/latch-free, decentralized and never block. Almost all main-memory engines avoid centralized lock tables. Instead, they co-locate locks with tuples, typically in each tuple’s header [146], saving the cost of coordinating with centralized lock tables. Thanks to the large memory space, memory becomes the major home of data, voiding the need for a buffer pool. Access methods and transactions directly manipulate virtual memory pointers, reducing the cost of indirection required by a dedicated buffer pool.

Main-memory systems usually employ lightweight concurrency control mechanisms, such as decentralized OCC [90, 169] and optimistic MVCC [18, 42]. One of the major reasons for employing these concurrency control methods (instead of 2PL) is that reads can be made very cheap with few or even no writes to shared memory. With 2PL, every read must be preceded by acquiring a lock in reader mode, which essentially turns a read operation into a write. Modern main-memory OLTP engines run

---

1 In database literature, a synchronization primitive that protects physical states, such as the log buffer, is referred to as a “latch”; synchronization mechanisms for scheduling logical tuple access are referred to as “locks”.

2 Traditionally, group commit only refers to the batching of log records to utilize faster sequential I/O speed; upon commit transaction threads normally block until log records are flushed. More recently, “group commit” becomes more or less a synonym to “pipelined commit” which is the same as traditional group commit but decouples transactions and threads, i.e., upon commit a thread leaves the committing transaction on the commit queue and continues to work on the next request, and the client only gets notified when the committing transaction’s log records are persisted [77]. Throughout this thesis, we use both terms interchangeably to refer to pipelined commit unless otherwise specified.
on hardware with massively parallel processors and consequently more severe NUMA effects. Frequent writes to shared memory—even when they do not contend logically—could saturate the interconnect and become a bottleneck.

Durability in main-memory engines is usually guaranteed in a similar way as in conventional engines: log records are still buffered and flushed periodically to durable storage. Optimizations such as pipelined group commit [77] still apply.

2.3 Synchronization

The making of high-performance database engines relies heavily on judicious use of synchronization primitives, in particular atomic instructions and latches. In this section we give background on related atomic instructions and latches. More detailed discussions on this topic can be found elsewhere [152].

2.3.1 Atomic Instructions

Modern processors provide a rich set of atomic instructions that are capable of changing individual memory words (e.g., 8-byte words) atomically in a single step regardless of the execution of other threads. The guarantees provided by these atomic instructions are quite similar to what transactions provide: an operation either succeeds or fails. Much use of these atomic instructions goes to building more complex but easier-to-use latches, as well as devising latch/lock-free data structures widely used by main-memory database engines. Techniques discussed in this thesis make use of the following atomic instructions found in 64-bit x86 architectures [72].

Compare-and-Swap

Compare-and-swap (CAS) operates on individual 8-byte words and changes the target word from one “expected” value to a “desired” value. The interface is as follows:

\[ \text{val} = \text{compare_and_swap}(*\text{address}, \text{expected}, \text{desired}) \]

To use CAS, the calling thread gives (1) the address of the memory word to change, (2) an expected value, and (3) a desired new value. The following pseudocode snippet shows how CAS proceeds on a logical level. Note, however, that this snippet cannot be implemented as a normal function to realize CAS; rather, it is implemented by the hardware.

```
function compare_and_swap(*address, expected, desired)
    val = *address
    if val == expected then
        *address = desired
    return val
```

The operation will only succeed if the memory word contains the expected value. It also returns the value (val) that was stored in the memory word when the atomic operation was conducted. So if the operation succeeded, val will be identical to expected; otherwise val could be any value that was stored in the memory word.
Fetch-and-Add

Unlike CAS, the atomic fetch-add-add (FAA) instruction always succeeds. As its name suggests, FAA atomically increments a specified memory word by a specified value. In other words, it performs the operation of $a + b$ in a single step and no concurrent thread will see any intermediate results. It exhibits the following interface:

$$val = \text{fetch}_\text{add}_\text{add}(\ast \text{address}, \text{addendum})$$

The logical level pseudocode is as follows:

```python
function fetch_add_add(*address, addendum)
    val = *address
    *address = val + addendum
    return val
```

To use FAA the calling thread gives an address of the memory word to change, and the addendum to be added. When finished, FAA returns the value stored in the memory word before the addition.

Exchange (Swap)

The atomic exchange (also called “atomic swap”) instruction stores in a single step a specified value to the memory word of interest and returns the previous value stored there:

$$\text{val exchange}(\ast \text{address}, \text{new\_value})$$

The interface and logical level functionality are almost identical to that of FAA’s, as shown below. The only difference is that it performs a blind store, rather than an addition.

```python
function exchange(*address, new_value)
    val = *address
    *address = new_value
    return val
```

2.3.2 Latches

Latches are synchronization primitives used to protect data structures shared among multiple threads or processes. There is a rich literature on the design and implementation of these synchronization primitives. Here we focus on the most related ones, i.e., spinlocks, including the (test-and-)test-and-set lock, and the MCS lock [119].

(Test-and-)Test-and-Set Lock

The test-and-set (TAS) lock consists of a single lock word, which switches between LOCKED or UNLOCKED states. Each lock requester tries to change the lock word from UNLOCKED to LOCKED. The requester who succeeds in making this change enters the critical section (i.e., lock acquired). Others must retry to acquire the lock.

Lock acquire can be implemented in two ways: using the atomic exchange or CAS instruction. In the former case, the lock requester issues an atomic exchange instruction on the lock word to change it to LOCKED, and checks the return value. If the return value is UNLOCKED, the requester has acquired the lock;
otherwise it must retry this process. This exchange-check process can be replaced by a single atomic CAS instruction: the requester issues a CAS instruction with expected and desired values being \texttt{UNLOCKED} and \texttt{LOCKED}, respectively. Again by checking the CAS's return value, the requester knows whether it has acquired the lock. Releasing a TAS lock is as simple as using a normal \texttt{store} instruction to reset the lock word back to \texttt{UNLOCKED}.

A popular variant of TAS is the test-and-test-and-set (TATAS) lock. As its name suggests, the only difference between TATAS and TAS is that TATAS reads the lock word first, and only attempts the atomic exchange or CAS instruction if the lock word contains \texttt{UNLOCKED}. TATAS reduces overhead as atomic exchange and CAS instructions are more expensive than normal \texttt{load} and \texttt{store} instructions.

The (TA)TAS lock is easy to implement and thus widely used. But it exhibits poor scalability due to its centralized design. Under high contention, failed lock acquire attempts are common, and the retry process could flood the interconnect (even with backoff strategies that only retry after pausing for a specified number of CPU cycles), leaving little bandwidth for useful work. Queue-based locks, such as the MCS lock [119], relieve this problem.

\section*{MCS Lock}

Mellor-Crummey and Scott [119] invented the MCS lock. In an MCS lock, the lock word represents a tail pointer to a linked list of lock requesters. Each lock requester arrives with its own queue node and swaps the tail pointer to its own queue node using the atomic exchange instruction. Thus, the tail pointer always points to the last requester, or \texttt{NULL} if none.

The requester-private queue node has two cache-line aligned fields: (1) a status \texttt{flag} and (2) a pointer (\texttt{next}) to a successor node. Swapping the tail pointer informs each requester of its predecessor. If there is none, then such a requester immediately enters the critical section. If there is a predecessor, then such a requester sets the \texttt{flag} field in its queue node to \texttt{WAITING}, installs a reference/pointer to its own node in its predecessor’s \texttt{next} field, and spins on its \texttt{flag} field until the \texttt{flag} is toggled to \texttt{GRANTED}. The release protocol involves setting a successor’s \texttt{flag} field to \texttt{GRANTED}, if present. If there is no successor, then the releaser issues a CAS instruction on the tail pointer to change it to \texttt{NULL} from a pointer to its own queue node. If the CAS fails due to some successor modifying the tail pointer (using an atomic swap instruction), then the releaser waits until the successor installs the \texttt{next} pointer and then toggles the successor’s \texttt{flag} field. In the MCS lock, the queue node is brought by the requester during acquisition and reclaimed by the requester after releasing the lock.

The MCS lock exhibits much better scalability (especially under high contention) because it spins locally (inside the queue node), instead of on a centralized location similar to what a (TA)TAS lock does. This saves much coherence traffic and leaves more interconnect bandwidth for useful work.

\section*{2.4 Emerging Hardware}

There are several hardware trends that are completely changing the way database engines are built. Next we discuss these trends. Designing database engines with and for such hardware is the focus of this thesis.
2.4.1 Byte-addressable, Non-Volatile Memory

Several NVRAM technologies—such as PCM [180], memristor [157] and STT-RAM [66]—are being actively investigated. Other efforts focus on NVDIMMs, or DRAM backed by flash/supercapacitors [2,171]. Though based on different technologies, these NVRAM products are all byte-addressable and can be placed side by side with DRAM on the memory bus. Software can use ordinary load and store instructions to access them. Data are immediately persistent upon write, no constant voltage is needed to maintain data.

Flash and supercapacitor backed NVDIMMs are already in mass production [2, 171]. During runtime, they exhibit exactly the same performance characteristics as DRAM, and the supercapacitor can provide enough energy to drain data from its DRAM component to flash, allowing data to be loaded back to DRAM once power is restored. Also, some data centers employ large battery arrays that essentially make all the memory and storage units in the entire data center persistent [86].

The Intel 3D XPoint SSD [39] is available today (as of August 2017), but only offered in PCIe interface. It still exhibits slower performance compared to DRAM. Although it can be configured to be byte-addressable through add-ons, it bears more similarity to a cached storage device, rather than memory. In this thesis our focus is NVRAM in DIMM form factor.

2.4.2 Parallel Processors and Deep Memory Hierarchies

Modern servers usually have tens of cores, and future servers will feature even more cores, e.g., up to 1000 [7]. Future servers will also be equipped with large (hundreds of gigabytes or terabytes of) DRAM. Some vendors have demonstrated precursors of such machines, e.g., The Machine [65] from Hewlett Packard Enterprise and Intel Rack Scale Architecture [73]. Although dark silicon is coming, and it is unclear at this point how many cores a single chip is able to power up at the same time, servers with more than 200 physical cores are already in the market [64,154], making high parallelism a fact that software must be able to utilize.

Large core count brings ultra-high degrees of parallelism. It also takes non-uniform memory access (NUMA) to the extreme: a single server will have many more sockets than today’s counterparts and maintaining cache coherence becomes equally harder. Accessing remote memory will become much more expensive, which has already been demonstrated by existing 16-socket servers [64,65,90]. As a result—if cache-coherent designs are still a norm—software must judiciously place data and avoid unnecessary remote memory access at all costs. It is also possible that future processors will offer non-cache-coherent options; software must be adapted if that is the case. Based on the designs of the precursors of such architectures [64], cache-coherent machines will still dominate for a significant amount of time. We therefore focus on cache-coherent machines in this thesis.

2.4.3 Fast Network Interconnects

High-speed network interconnects (e.g., InfiniBand) were mostly used by high-performance computing systems, but are now becoming cost effective and being adopted by database systems. Despite the differences in their underlying technologies, these interconnects all provide high bandwidth and low latency, and support fast remote direct memory access (RDMA), changing the common belief that network is the slowest part of a distributed system [22].

Two major kinds of fast networks are InfiniBand and Converged Ethernet. InfiniBand is a switched fabric network that can be used both within and among nodes. Today’s InfiniBand already provides
aggregate bandwidth that is close to that of a multi-socket single server [190]. The data rate of the upcoming HDR 4× is 200Gbps [68]; we expect the trend to continue.

Converged Ethernet is an IEEE standard that converges several standards [160] and allows the layering of InfiniBand’s low latency features to implement RDMA over Converged Ethernet (RoCE). RoCE provides competitive performance to InfiniBand and can be implemented in software or hardware. For example, some Mellanox RoCE products support up to 100Gbps link speed [118]. Software RoCE is compatible with any standard Ethernet, but is not as performant as the hardware-accelerated ones.

The high bandwidth of these fast network interconnects enables fast RDMA. RDMA allows nodes in a cluster to access each other’s designated memory areas, without having to go through multiple layers in the OS kernel, avoiding unnecessary data copying and context switches between the user and kernel spaces which are typically unavoidable using the TCP/IP stack. Thus, RDMA utilizes the high bandwidth and low latency of next-generation networks well. With RDMA over InfiniBand, it takes around only 1µs to deliver 1KB of data [22]. Note that it takes around 0.08µs for the CPU to fetch the same amount of data from DRAM. With larger data sizes, the slowdown because of longer latency will almost disappear: bandwidth will become the dominant factor.

2.5 Experimental Environment

2.5.1 System Model

Throughout this thesis, we follow an “embedded” database approach to evaluate different systems and algorithms. Using this approach, the application developer writes transactions and queries directly using the system-provided APIs in C/C++, without the overhead to parse, optimize, and send/receive SQL queries via TCP/IP sockets and alike. This approach is widely accepted and used in many research proposals [76,90,145,169]. Without proper optimizations the performance of full-feature systems with these overheads could be orders of magnitude slower. Since our focus is the backend database engine, we follow this embedded database model for fair comparison.

2.5.2 Benchmarks

We rely on standard benchmarks and custom microbenchmarks to measure performance. Microbenchmarks are often tailor-made for specific systems and designs. Thus, they differ from each other significantly. We introduce them in the corresponding chapters when needed. Occasionally, we tweak the default design of standard benchmarks (e.g., by adding new transactions) in case the benchmarks are inadequate for testing certain workloads. Such tweaks help us stress test our proposed designs. These changes are also described in the corresponding chapters that require them. Here we summarize the standard benchmarks used in the rest of this thesis.

Each benchmark has its own targeted workloads and scenarios, so the experiments in each individual chapter may not necessarily cover all the benchmarks described in this section. Instead, we only run the relevant benchmarks that will stress the particular workload or system components we target. For example, for high contention workloads, we run custom YCSB microbenchmarks rather than stock TPC-C benchmarks because the latter by default does not exhibit high contention and thus will not be able to stress test the underlying concurrency control mechanism, defeating the purpose of the experiments.
On the other hand, TPC-C can always be used to test how the system behaves under low-contention workloads.

TPC-B

TPC-B [166] models an update-intensive banking workload that aims to stress database engines, without any network and terminal handling modeled in TPC’s other benchmarks. TPC-B consists of a single transaction called Account-Update, which manipulates four tables: Branch, Teller, Account, and History. The Account Update transaction simulates the process of customers’ withdrawals and deposits from different tellers.

TPC-C

TPC-C [167] simulates an order-entry environment of a wholesale supplier. It is the dominant benchmark for traditional OLTP systems. It is a write-intensive and low-contention workload that involves six tables and five transactions, including New-Order (45%), Payment (43%), Delivery (4%), Order-Status (4%) and Stock-Level (4%). Among these transactions, Order-Status and Stock-Level are read-only, while the others are read-write.

The database is easily partitionable. For instance, one could partition the database by warehouse. Each thread can be assigned a home warehouse; by default 15% and 1% of the Payment and New-Order transactions are cross-partition, respectively. The benchmark by default specifies about 10% of transactions to access remote warehouses.

The benchmark’s original specification reports the number of New-Order transactions processed per minute as the major metric. In this thesis, unless otherwise specified we use a different approach and report the aggregate throughput per second of all transactions in the mix, as well as transaction throughput breakdowns. This approach is widely used in the literature [89, 90, 169, 179], as it gives a more complete picture of system performance, especially when we modify the original mix to focus on certain transactions.

TPC-E

TPC-E [168] is a more recent OLTP benchmark that features more sophisticated and realistic tasks that are performed by brokerage firms. The database includes 33 tables of varying structure. The benchmark consists of 11 transactions that manipulate data related to user accounts and one cleanup transaction called DataMaintenance that simulates administrative updates, detailed below:

- **Trade-Order (10.1%):** Order the trade, purchase, or sale of a security;
- **Trade-Result (10.0%):** Represent the completion of a stock market trade;
- **Trade-Lookup (8.0%):** Retrieve information regarding specified customer accounts, trade transactions, or securities;
- **Trade-Update (2.0%):** Retrieve and possibly modify information regarding specified customer accounts, trade transactions, or securities;
- **Trade-Status (19.0%):** Return the status of 50 trades;
Chapter 2. Background and Environment

- Customer-Position (13.0%): Calculate the current market value of each of a specified customer’s accounts;
- Broker-Volume (4.9%): Generate a list of total potential volume generated by a list of brokers for all trades in a given sector;
- Security-Detail (14.0%): Retrieve all information of a given security;
- Market-Feed (1.0%): Retrieve the latest prices for securities;
- Market-Watch (18.0%): Calculate the percentage change in value of the market capitalization of a collection of securities;
- Data-Maintenance (once every minute): Modify a table that is used by other transactions in the mix;
- Trade-Cleanup (one time): Cancel pending or submitted trades.

TPC-E has a significantly higher read-to-write ratio (∼10:1 vs. ∼3:1 of TPC-C) [31]. Among these transactions, Broker-Volume, Customer-Position, Market-Watch, Security-Detail, Trade-Status, and Trade-Lookup are read-only. Data-Maintenance and Trade-Cleanup are run only once every minute and once per benchmark run, respectively.

Similar to the TPC-C benchmark, the original TPC-E specification requires the database engine report the number of Trade-Result transactions processed per second. Instead, we report the aggregate of the full mix (excluding Trade-Cleanup and Data-Maintenance).

Yahoo! Cloud Serving Benchmark (YCSB)

YCSB [35] is a newer and simpler benchmark that has only one table and simple, short transactions. YCSB has no locality, thus no partitioning makes sense. It parameterizes the number of records and the number of reads/writes in each transaction for simulating various settings. In this thesis, we focus on read-modify-write (RMW) and read-only operations. Unless otherwise specified, each YCSB transaction consists of a certain number of operations (e.g., 10), where some of them are RMW operations and the rest are read-only operations. Such flexibility gives us the ability to evaluate and stress CC schemes for different workloads.

Telecom Application Transaction Processing Benchmark (TATP)

TATP (also known as “Telecom One”) [130] models a cell phone service database. The data is stored in four tables: Subscriber, Access_Info, Special_Facility, and Call_Forewarding. The full mix of TATP consists of three read-only transactions and four read-write transactions:

- Get-Subscriber-Data (35%): Retrieve details of a subscriber;
- Get-New-Destination(10%): Retrieve call forwarding destination;
- Get-Access-Data (35%): Retrieve access validation data;
- Update-Subscriber-Data (2%): Update data for a random subscriber;
• Update-Location (14%), Update subscriber location;

• Insert-Call-Forwarding (2%): Add a piece of new call forwarding information;

• Delete-Call-Forwarding (2%): Remove a piece of new call forwarding information.

According to the mix, 80% of these transactions are read-only, including those that start with “Get”; the rest are read-write transactions. TATP transactions are very small, each touching only 1–4 records, but stress the logging and concurrency control components.

* * *

Now we have laid out the necessary background for the techniques we are about to propose and discuss, beginning with the next chapter on removing the centralized logging bottleneck.
Chapter 3

Scalable Logging through NVRAM

NVRAM is fundamentally changing the design principle of transaction logging. It potentially invalidates the need for flush-before-commit as log records are persistent immediately upon write. Distributed logging—a once prohibitive technique for single-node systems in the DRAM era—becomes a promising solution to easing the logging bottleneck because of the non-volatility and high performance of NVRAM.

We advocate NVRAM and distributed logging on multicore and multi-socket hardware. We identify the challenges brought by distributed logging and discuss solutions. To protect committed work in NVRAM-based systems, we propose passive group commit, a lightweight, practical approach that leverages existing hardware and group commit. We expect that durable processor cache is the ultimate solution to protecting committed work and building reliable, scalable NVRAM-based systems in general. We evaluate distributed logging with logging-intensive workloads and show that distributed logging can achieve as much as $\sim 3\times$ speedup over centralized logging in a modern DBMS and that passive group commit only induces minuscule overhead.¹

3.1 Introduction

Since its debut in the early 90s, ARIES [126] has been the de facto standard of transaction logging. Despite the prevalence of multicore hardware and large main memories, most systems still use a centralized ARIES-style log that buffers log records in DRAM until a commit request forces them to stable storage. Given the volatile, fast DRAM and non-volatile, slow disk, such flush-before-commit principle improves performance, without risk of lost work, by replacing the random write-back of all dirty pages in a transaction’s footprint with (ideally) a single sequential I/O to harden the log. However, centralized logging has become a significant bottleneck on today’s massively parallel hardware [77, 78].

Emerging NVRAM technologies, such as Intel 3D XPoint [39], PCM [180], memristor [157] and STT-RAM [66], are fundamentally changing the design principle of logging. As we have discussed in Section 2.4.1, these NVRAM products promise high performance, good scalability and low power consumption [97]. They can be manufactured as DIMMs and accessed through ordinary load and store instructions. The combination of DRAM-like performance and non-volatility makes NVRAM an attractive candidate for logging: log records persist immediately upon write, so transactions need not force log records to disk at commit. Various simplifications can then be admitted to improve logging.

¹ This chapter is based on materials that appeared in VLDB 2014 [174].
Figure 3.1: Time breakdown of running the Update Location transaction of TATP on Shore-MT, a modern OLTP engine designed for multicores. Log contention is a major source of overhead, especially so under high parallelism.

performance and reliability, such as removing asynchronous and group commit [52,143], and unifying the log buffer with backend log storage [48].

Although NVRAM-based logging brings improvements in latency and single-thread performance, contention for the log head still exists because of the centralized design. Figure 3.1 shows the time breakdown of Shore-MT [76], a prototype database system optimized for multicore and multi-socket hardware, when running a logging-intensive workload on a 24-core quad-socket server (48 hardware threads with hyper-threading). As system load increases, contention for the centralized log becomes a major overhead. Though rarely used in single-node systems, we observe that a distributed log has the potential to remedy this situation. In this chapter, we show that NVRAM-enhanced distributed logging can eliminate the logging bottleneck for modern multicore and multi-socket hardware. We equip each log with an NVRAM-based buffer, accessed via load and store instructions. Transactions commit immediately after buffering their commit records, and log records are only de-staged to disk when needed (e.g., when the log buffer is full).

Neither adopting distributed logging nor NVRAM is trivial. Distributed logging poses two main challenges: how to assign log records to logs (log space partitioning), and how to prevent holes in the distributed log, without imposing excessive overhead at commit time. ARIES-style recovery favors partitioning the log either by page or transaction, i.e., assigning all log records involving a given page or transaction to the same log, respectively. As we will show in later sections, adopting either approach involves different design challenges and performance implications.

NVRAM is no panacea, either. Modern processors heavily rely on multiple levels of caches to improve performance. Though log records become persistent immediately after reaching NVRAM, by default they are first buffered in volatile processor caches (“write-back” caching). Data remains cached until some event evicts the it from the caches (e.g., a cache conflict or the OS explicitly requests a cache flush), and a power failure in the meantime would lose log records buffered in the SRAM cache. Spreading log records over multiple logs worsens the situation: a transaction cannot commit until all records from all logs it accessed have reached persistent storage. In case of transaction-level partitioning, the transaction must also ensure that all previous log records for pages in its footprint have become persistent, even if they reside in a log the transaction never accessed. Tracking these dependencies explicitly is prohibitively complex [77], and flushing log records to NVRAM at every commit imposes high overheads.

Based on existing hardware, we propose passive group commit, a lightweight group commit protocol for NVRAM-based distributed logging, to protect committed work upon failure. Unlike most NVRAM logging
proposals, passive group commit does not rely on NVRAM-specific hardware support for customized memory barriers [34], and does not issue a barrier after buffering each log write to NVRAM [48, 140]. Instead, passive group commit builds on commodity processor features—write combining and memory barriers—and leverages group commit to ensure all necessary log records are persistent before a transaction finishes committing.

Passive group commit is very lightweight and offers significant performance benefits for existing hardware equipped with NVRAM, but is actually a stop-gap solution. For future NVRAM-based systems, we argue that the ultimate solution is not a new type of memory barrier, but rather a durable processor cache [158, 193]. A durable cache ensures that writes are effectively persistent as soon as they leave the CPU’s store buffers, and can be implemented either with on-chip NVRAM or a supercapacitor-based solution that drains caches whenever power is lost. Durable caching is largely transparent to software. The only requirement, on some processors, is to employ memory fences that prevent reordering of stores (x86 does not reorder stores, but other architectures like ARM are more aggressive). Durable caches are also helpful in extending the lifetime of NVRAM by allowing the cache hierarchy to absorb the vast majority of writes. Although not yet widely available, they could be developed rapidly using existing technology: prototypes have already been built [128, 132]. Durable caches would improve and simplify our proposed design even further, by removing the need for passive group commit and write combining.

In summary, we make the following contributions:

- We show that NVRAM allows for practical distributed logging, thereby eliminating the scalability challenges posed by a single centralized log.

- We propose passive group commit, a lightweight group commit protocol that leverages existing hardware support for write combining to protect committed work upon failures.

- We show that durable cache architectures promise to both simplify distributed logging and improve its performance in NVRAM-based systems.

Note that the focus of this work is not to create another NVRAM-based log that removes I/O latencies. Rather, we propose to leverage NVRAM in support of distributed logging, with the goal to alleviate the contention bottleneck that all centralized logging schemes suffer (NVRAM-based or otherwise).

In the rest of this chapter, we discuss design challenges brought by distributed logging and propose solutions in Section 3.2. Section 3.3 details approaches to protecting committed work and expected support for NVRAM from hardware and the OS. We present evaluation results and conclude this chapter in Sections 3.4 and 3.5, respectively.

### 3.2 Distributed Logging

Distributed logging eases the logging bottleneck by spreading log insertions over multiple physical logs. Historically, distributed logging has been prohibitive for single-node systems due to dependency tracking and I/O overheads. Most single-node systems avoid using a distributed log, and even distributed systems have used a centralized log, hosted on a dedicated node [107]. NVRAM makes it possible to use distributed logging in single-node systems, as it potentially eliminates the need for flush-before-commit. Log de-staging is only required when log buffers are full, and uses large sequential writes to maximize disk bandwidth utilization.
Logging is tightly coupled with other core components in database systems (e.g., the transaction manager) and is strongly affected by recovery: ARIES uses physical logging for redo and logical logging for undo (physiological logging). Both redo and undo have their own parallelism models: during redo, modifications to different pages can be replayed in parallel, while undo is parallelized at transaction-level. Resource managers that generated the log records select a log either by page or transaction, and the competing recovery parallelism modes add to the importance of selecting the right partitioning scheme. The rest of this section discusses the design challenges and trade-offs posed by distributed logging, in terms of both forward processing and recovery.

### 3.2.1 Forward Processing

#### Uniqueness of Log Records

In a centralized log, the log sequence number (LSN) uniquely identifies each log record; records written by the same transaction are chained together using a `prevLSN` field, which identifies the previous log record written by the same transaction. When a transaction aborts, log records are undone by following the chain of `prevLSN` pointers, most-recent first. However, LSNs are not unique identifiers in distributed logging, because they only indicate a log record’s position within the log that holds it, saying nothing about the relative ordering of records from different logs. Under page-level log space partitioning, a transaction could touch any page and write to any log, and a `prevLSN` no longer allows a transaction to trace its log records backward. For transaction-level log space partitioning, a transaction only inserts log records to a single log, but page LSNs are no longer unique in the transaction because updates generated by different transactions go to different logs. The lack of ordering among records from different logs potentially causes the recovery manager to skip applying newer log records, or even to apply older log records on top of newer log records that happen to have smaller LSNs.

To uniquely identify log records, we propose a *global sequence number* (GSN) based on logical clock [95]. At runtime, a GSN is maintained in each page, transaction and log. Pages and log records also store a GSN with them permanently. Pinning a page in the buffer pool sets both page and transaction GSNs to \(\max(tx\ GSN,\ page\ GSN) + 1\) if the transaction intends to modify the page; otherwise only the transaction GSN is set to \(\max(tx\ GSN,\ page\ GSN)\). Inserting a log record will set both transaction and page GSNs to \(\max(tx\ GSN,\ page\ GSN,\ log\ GSN) + 1\). The same GSN is also stored in the log record that is being inserted. In addition to the GSN, each log record also stores an LSN, which indicates the offset into an individual log that holds it. Although GSNs are only partially ordered, they uniquely identify log records and provide a total order for log records belonging to any one page, log, or transaction.

GSN solves the uniqueness problem posed by LSN in distributed logging. The recovery manager can follow GSNs to determine if a record should be applied to its targeted page. Note that GSN might not be mandatory for page-level log space partitioning because log records for the same page are always stored in the same log. To allow transaction rollback, each transaction should record not only the LSN, but also the log in which each log record is stored. Though not required, using GSN can avoid such complex bookkeeping and simplify the design of page-level partitioning. Transaction-level partitioning relies heavily on GSN to function properly during forward processing as log records for the same page could be stored in arbitrary logs by different transactions.
Cross-Log Aborts

GSN only partially solves the problem of tracking log records in a distributed log. Compared to LSN, GSN is also monotonically increasing, however, it does not identify the log a particular log record came from, nor give the record’s byte offset within that log. A naïve implementation will scan multiple logs during forward processing to find the correct log record with a specified GSN. To avoid expensive log scans, we maintain a private DRAM undo buffer which records log writes in each transaction. Similar approaches have already been implemented in both commercial and open source systems to simplify transaction rollback [156]. The buffer is managed as a stack (with abort repeatedly popping records), and discarded after the transaction is ended. Log records generated during rollback are never undone, and recorded in the NVRAM log buffer only.

Processor Affinity

The shift to multicore and multi-socket systems brings non-uniform memory access (NUMA), where memory is either “local” or “remote”, with the latter being much more expensive to access. Transaction threads could be scheduled on arbitrary processors, spreading insertions over any log, which in turn could be allocated in any NUMA node (or striped over all of them). If the log buffer and the transaction thread are not within the same NUMA node, as shown in Figure 3.2(a), log insertion will involve remote memory accesses. In the figure, each NUMA node has an NVRAM log buffer allocated from local memory. Two transaction threads (Tx 1 and Tx 2) run on two processors. We assume all pages with odd page numbers are covered by Log 1, and all even numbered pages are covered by the other log. Under page-level partitioning, both transactions access remote memory for log insertions, while transaction-level partitioning generates only local writes. Since page-level log space partitioning allows transactions to write to any log, accessing remote memory is inevitable, unless transactions are scheduled in a way that restricts threads to particular sets of pages (e.g. using physiological partitioning [138]). Transaction-level partitioning, on the contrary, fits directly with multi-socket hardware as each transaction’s writes go to one log. As Figure 3.2(b) shows, a log could be allocated in each NUMA node, with transactions assigned to logs based on the CPU they run on. This approach eliminates remote memory accesses when inserting log records, improving logging performance and freeing up interconnect bandwidth for other uses.

Checkpointing

To accelerate recovery, ARIES mandates periodic checkpoints of all dirty pages and in-flight transactions during forward processing. The recovery manager can load these tables directly rather than scanning
the entire log to reconstruct them. A distributed log requires the checkpointing thread to be aware of individual logs, especially for page-level log space partitioning: either page GSN must be recorded, or a dirty pages table should be generated for each log. The analysis pass during recovery can then process log records from different logs in parallel. As the active transactions table is global and records only transaction status, it could be stored in a predefined log (e.g., the first one), and recovered with a pre-analysis step. Another approach is to generate multiple active transactions tables, one for each log, to remove the pre-analysis pass.

Log Space Utilization and Skew

Under page-level partitioning, transactions leave log records in multiple logs during forward processing, but all records for a given page are stored in the same log. If certain pages are especially hot, the corresponding logs could become over utilized, while leaving other logs under utilized. In contrast, transaction-level partitioning stores each transaction’s log records in a single log, so log accesses should be distributed uniformly as long as the OS spreads load evenly across cores. Transactions are usually short enough, and migrations rare enough, that we can assign each transaction a log based on the core where it first runs, with little impact on performance.

3.2.2 Recovery

ARIES separates recovery into three passes: analysis, redo and undo. The analysis pass extracts information (e.g., lists of dirty pages and active transactions) logged before the system failed, to determine how redo and undo should proceed. With a distributed log, the analysis pass could be parallelized at both page or transaction levels, though a pre-analysis step might be needed first, to retrieve the dirty page and/or active transactions tables (as discussed previously). Once analysis completes, ARIES naturally supports parallel redo and undo: redo of different pages can proceed in parallel, as can undo of different transactions. Ideally, a distributed log should recover fully in parallel, but neither log partitioning choice aligns perfectly with both redo and undo. We discuss how distributed logging impacts recovery under page and transaction level partitioning in this section.

Page-Level Log Space Partitioning

A page-partitioned distributed log suits parallel redo nicely. The only difference is that each redo thread should work with pages from a single log, preferably residing on the same socket. However, the undo pass requires analysis to build up the same per-transaction log buffers that would have existed during forward processing, which is potentially expensive given that we expect most transactions to have committed with only a few losers left to undo. Further, if the per-transaction buffers store pointers rather than copies of log records, a parallel undo pass would have to randomly access multiple logs, which is expensive for disk-based systems (recall that log records are de-staged to disk as the NVRAM buffer fills). The overheads can be mitigated by copying log records to the per-transaction buffers, or by using efficient checkpointing to ensure that most log records to be undone are in NVRAM.

Tracking dummy CLRs written by nested top actions also complicates undo for page-level partitioning. ARIES uses nested top actions and dummy CLRs to denote (usually physical) operations that will never be undone once complete, even if the requesting transaction eventually aborts. For example, B-tree page splits only affect physical representations (not logical contents in the database) and can safely persist
across transaction rollback [126]. A dummy CLR is written after the nested top action (e.g., B-tree split) completes, with a \texttt{prevLSN} field pointing directly to the log record before the nested top action started. Since a dummy CLR could cover multiple log records which are spread over multiple logs under page-level partitioning, tracking back to the log record before the nested top action involves jumping across different logs. Even though the log records written by nested top actions are never undone, they still have to be examined (and potentially cached) during the analysis pass.

To simplify undo, we replace nested top actions with system transactions [56]. A system transaction only modifies physical representations of the database, and will not change any logical contents visible to the user. Therefore, system transactions acquire no locks, and can inherit latches from the invoking transaction. The user transaction can start a system transaction when physical-only changes are required, and resumes after it finishes. System transactions also generate log records like user transactions do, and the rollback of an interrupted system transaction is identical to that of a user transaction: it can safely go through its log records in the private DRAM undo buffer to undo them one by one.

**Transaction-Level Log Space Partitioning**

Parallel undo is straightforward for a transaction-partitioned log as all the log records to undo a transaction reside in the same log. However, redo becomes more complex because dependencies among pages can reside in any log. A naive replay of all logs in parallel results in log records for the same page being applied in arbitrary order, potentially skipping certain records. For example, consider a transaction \texttt{Tx 1} that modified page \texttt{P1} and logged it on \texttt{Log 1} with GSN 50. Another transaction \texttt{Tx 2} modified the same page but logged it on \texttt{Log 2} with GSN 60. Though the log record generated by \texttt{Tx 2} had a larger GSN, its relative position (byte offset) in \texttt{Log 2} could be well before the log record generated by \texttt{Tx 1} in \texttt{Log 1}, causing the log record generated by \texttt{Tx 2} to be applied first (assuming we start redoing both logs at the same time and they proceed at the same speed). As the redo pass will not apply a log record with a GSN less than the page GSN, the record generated by \texttt{Tx 1} will be skipped.

A two-step redo pipeline, reminiscent of map/reduce, can solve this problem efficiently: the first stage uses \(N\) threads to scan the logs up to some target GSN, partitioning log records into \(M\) buckets by page ID. \(M\) threads can then sort each bucket and apply the changes in parallel, without risk of skipped records. Full parallelism can be achieved using this approach, with the available parallelism balanced between the \(N\) partitions and \(M\) redo threads.

### 3.2.3 Page vs. Transaction-Level Partitioning

Table 3.1 compares the two approaches in terms of design challenges. In general, a transaction-level partitioned distributed log features straightforward undo and tracking of log records. However, it needs more complex redo because of dependencies among pages and transactions. Page-level log space partitioning is implicitly supported by ARIES. It allows easy parallel redo, but complicates with transaction-level parallel undo and generates more remote accesses. As we will show in our evaluation results, transaction-level partitioning achieves higher performance during forward processing for multi-socket hardware as it avoids crossing NUMA boundaries to insert log records. For both designs, checkpointing records could be kept in a predefined log or generated for specific logs to ease recovery: page-

\[\text{The system transaction can use the resources (e.g., latches) of its invoking thread, which therefore is essentially “waiting” for a function call to return, instead of another thread.}\]
Table 3.1: Comparing page-level and transaction-level log space partitioning.

<table>
<thead>
<tr>
<th></th>
<th>Page-level partitioning</th>
<th>Transaction-level partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Record uniqueness</strong></td>
<td>LSN is adequate, but GSN is more convenient</td>
<td>GSN is required</td>
</tr>
<tr>
<td><strong>Checkpointing</strong></td>
<td>Per-log dirty page tables without GSN</td>
<td>Per-log transaction tables for simpler recovery</td>
</tr>
<tr>
<td><strong>Recovery</strong></td>
<td>Easy parallel redo, but parallel undo of transactions expensive (need to hop among multiple logs)</td>
<td>Log synchronization needed for parallel redo; cheap parallel undo by transactions</td>
</tr>
<tr>
<td><strong>CPU affinity</strong></td>
<td>Cannot utilize CPU affinity since a transaction could potentially insert to any log</td>
<td>Logs can be pinned to specified NUMA nodes to serve requests exclusively for the node</td>
</tr>
<tr>
<td><strong>Log space utilization</strong></td>
<td>May be unbalanced, depending on workload</td>
<td>Balanced if transactions are scheduled properly</td>
</tr>
</tbody>
</table>

Page-level partitioning could use individual dirty pages tables for each log, while transaction-level partitioning could utilize per-log active transactions tables.

### 3.3 NVRAM-based Logging

Modern processors rely heavily on caching for good performance. When accessing NVRAM through the memory interface, log records are first cached by the CPU. Log records that are not yet persistent in NVRAM could be lost upon failures, which risks losing committed work. ARIES truncates the log at the first hole, and transaction-level partitioning could leave the database unrecoverable if earlier updates to a page in one log are lost and the system attempts to apply later updates to the same page from a different log. Worse, the transactions generating such log records may not have committed before a crash. So it is not sufficient to track which logs a committed transaction touched. Instead, we must ensure that no transaction can commit until every log is durable up to the transaction’s commit GSN. The overhead of flushing every log at every commit (or every time a page moves to a new log) is a principal reason why implementations have avoided distributed logging when possible.

It is straightforward to store log records directly in NVRAM by completely disabling caching for the NVRAM address range. However, this approach adds significant latency to the critical path (log insertion) and threatens to cancel out any benefits brought by a distributed log. Prior NVRAM research has proposed augmenting the processor with a new memory epoch barrier [34]. Although such a barrier would be complex to implement, and would have high runtime overhead, some NVRAM logging proposals rely on this primitive. Fang et al. [48] propose to issue an epoch barrier after writing each log record to persist the record. To tolerate holes in the log, an epoch barrier is issued as soon as the log header is written, leaving transactions to populate log records at their leisure. Pellely et al. [140] proposed NVRAM Group Commit to mitigate the overhead of the epoch barrier. All these approaches are tailored for centralized logging, however. With a distributed log, the issuing processor would have to not only flush its own records to NVRAM, but also ensure that other processors have done the same (in case any dependent log records are not yet durable). The proposed epoch barrier introduces more complexity in both processor cache and the memory controller, and it is not clear whether this approach can scale in multicore environments. Instead of relying on hypothetical hardware primitives, we develop a group commit protocol that protects committed work by leveraging existing hardware support.
3.3.1 Passive Group Commit

This section presents passive group commit, a lightweight group commit protocol for reliable and high performance NVRAM-based distributed logging. Passive group commit leverages existing hardware primitives to protect committed work while still providing good performance.

Leveraging Existing Hardware Primitives

Modern processors allow the OS to control how the cache interacts with accesses to a given physical address range. Addresses can be marked as uncacheable (important for memory mapped I/O), write-through (serving reads from the cache but propagating all writes immediately), write-back (the normal caching mode), or write-combining (WC). WC is a compromise for uncacheable memory in cases where write coalescing is tolerable. Accesses to WC memory bypass caches for both reads and writes, but can buffer writes inside the CPU for a short time. For example, the Intel Core series processors have eight WC buffers per core [70]. WC buffers can coalesce repeated/adjacent writes (replacing several small memory transfers with a larger one) and allow store instructions to complete before the write actually completes (to be drained lazily); with judicious use of normal memory fences, write combining provides comparable performance to cached writes, while guaranteeing that those writes still reach the memory system in a timely manner [21,69]. Events such as interrupts and context switches also drain WC buffers, avoiding the need to worry about thread migrations. Though WC only provides uncached (slow) reads, it has minuscule effect on logging, which is write-mostly. In particular, our distributed log does not read from NVRAM during forward processing (abort is supported by the private DRAM undo buffer). During recovery, we map the NVRAM in write-back mode before the undo pass for fast reads, because no log records are generated during the analysis or redo passes. Thus, write combining is a natural fit for NVRAM-based logging buffers. Whenever writing sensitive log records (e.g., commit), a memory barrier (e.g., mfence) is issued to empty the WC buffers and persist log records in NVRAM. When the user receives a return value from the log manager, the records are guaranteed to reside in NVRAM. However, simply issuing memory barriers when committing the transaction does not solve the dependency problem, as discussed in the beginning of this section.

Protecting Committed Work

Log records are spread over multiple logs by transaction threads running on different processors. Before a transaction can safely commit, all of its log records, along with all the log records that logically precede them, must become persistent. For transaction-level partitioning, log records affecting pages touched by a committing transaction could have been created by other, as-yet-uncommitted transactions running on other processors. The committing transaction’s own memory barrier only empties the worker thread’s own WC buffers, potentially leaving (parts of) other dependent log records in other cores’ WC buffers; a memory barrier from the running processor will only make its writes globally visible from a cache coherence perspective, and does not guarantee that other WC buffers have drained. We thus introduce a variant of group commit to coordinate draining of WC buffers on other cores.

Group commit was originally proposed to avoid excessive random writes to the disk. It gathers and writes multiple commit records in a single disk I/O. Passive group commit relies on a similar mechanism to ensure that all log records written by the committing transactions are persistent in NVRAM before notifying the client that commit succeeded. In each thread, we maintain a thread-local variable called...
Figure 3.3: Protecting committed work with passive group commit. The passive group commit daemon keeps dequeuing committed transactions with GSN ≤ the minimum dgsn and signaling stragglers to make progress.

To record the GSN of the last log record the thread can guarantee to be persistent. We reserve one bit as a dirty flag, to indicate whether the thread has written any log records that might not be persistent yet. Whenever a thread writes a log record, it sets the dirty flag. When a transaction commits, it will issue a memory fence, clear the dirty bit and store the record’s GSN in dgsn. The transaction is then added to a group commit queue. The group commit daemon periodically examines the dgsn of each worker thread and records the smallest dgsn it sees (min_dgsn), as the guaranteed upper bound of persistent log records. Log records written by all transactions on any processor with a GSN ≤ min_dgsn are guaranteed to be persistent. The passive group commit daemon dequeues transactions once their commit record precedes min_dgsn and notifies the client. If any thread takes too long to update its dgsn (perhaps due to a long read-only query), the daemon sends a signal to the straggler. Signal delivery implicitly drains the WC buffers of the processor the thread is running on, and the thread’s signal handler then clears the dirty bit so the commit daemon can proceed. Note that operations on dgsn need not be atomic, because the owning thread makes all writes and the daemon thread reads a monotonically increasing GSN.

Figure 3.3 shows an example of passive group commit in action. Here, each transaction thread runs on a physical processor. Transactions Tx 3 and Tx 4 update pages P1 and P2 before commit. At steps 3 and 4, both transactions issue an mfence, clear their dirty bit, and join the commit queue. Another transaction Tx 5 sets its dirty bit and inserts a log record for updating P1 (right of the figure). The daemon reads dgsn of all threads and obtains the minimum, which is 8 from Tx 5. Note that Tx 5 has not committed yet, therefore it still has the dirty bit set. The daemon then dequeues all transactions with dgsn ≤ 8 in the queue and notifies the clients. After step 7, Tx 3 and Tx 4 are left in the queue since they have dgsn > 8. Normally the daemon will start another round of checking for the smallest dgsn and repeat steps 1–7. However, if there is any straggler which is not updating its dgsn, the commit queue will become full gradually and no more transactions could be removed from it due to low min_gsn. In such cases, the daemon will signal through pthread_kill all threads whose dirty bit is still set. Upon receiving the signal, the thread’s signal handler will set the correct dgsn and clear the dirty bit, as shown in steps 8–10a. Since the signal delivery is asynchronous, the call to pthread_kill may return before the signal is delivered. Therefore the daemon will return to step 6 and start checking again, as shown in step 10b.

Note that it is unnecessary to issue a barrier for every processor. We only need to make sure that all the processors that have written log records issue a barrier to empty their WC buffers. This observation makes passive group commit lightweight: signaling is only required if stragglers arise.
3.3.2 Durable Processor Cache

Recent advances in ferroelectric RAM (FeRAM) [23] and STT-RAM [158] have made it possible to build processors with durable caches. For instance, real prototypes [132] have been built using FeRAM to realize “normally off, instant on” computing. Kiln [193] is a recent proposal which adopts both non-volatile cache and main memory to build a persistent memory system. Another approach is augmenting the processor with a supercapacitor [34] that can drain the SRAM cache to NVRAM if power is lost. As a beneficial side effect, durable caches improve NVRAM lifetime by allowing large L2 and L3 caches to absorb the vast majority of writes without risking data loss. When a durable cache is available, programming NVRAM becomes much simpler: NVRAM can be treated like normal memory by the OS and caches, and software need only issue memory fences when it is necessary to force ordering of writes. Based on these observations, we expect that durable processor cache (built with capacitor-backed SRAM or NVRAM such as FeRAM) is the ultimate solution to reliable and high performance NVRAM-based systems. Though durable cache is still in its infancy, we believe it will become the most convenient drop-in solution in the future. In the context of a distributed log, passive group commit is no longer necessary since all writes, on all processors, are immediately durable as long as each log insertion includes a (now inexpensive) memory fence to ensure write ordering on certain architectures that are aggressive on reordering store operations (e.g., ARM and SPARC). Therefore, in later sections we also measure the performance that could be achieved with a durable cache, in addition to evaluating passive group commit.

3.3.3 Hardware/OS Support

To use NVRAM in existing systems, the memory controller should distinguish DRAM and NVRAM, so that the OS can manage both of them and enable applications to make requests for both types of memory explicitly. Moreover, transaction level log space partitioning requires directly allocating NVRAM from specified NUMA nodes. It is therefore desirable to have a certain amount of NVRAM on each NUMA node. We use NVRAM for buffering log records, and use DRAM as main memory for the buffer pool.

At the software level, the memory allocator should provide interfaces for explicitly allocating NVRAM from a specified NUMA node for transaction-level log space partitioning. For passive group commit to work, the OS should expose an interface for user space to specify the caching mode. However, currently most systems expose such interfaces in kernel space or only allow very limited control in user space (e.g., via MTRR registers). NVRAM-based systems would benefit strongly if the OS could expose more interfaces for user space to directly control the behavior of NVRAM. These changes are straightforward and easy to make in the OS. For example, caching-mode-related functions are already available for Linux device drivers. Even without current OS support, our kernel module for NUMA-aware write combining memory allocation has only $\sim 250$ LoC. After changes in the OS, such functionality can be exposed to user space as system calls easily. Given historical trends, however, we suspect that durable caches will become widely available before operating systems incorporate such support.

3.4 Evaluation

We implement both flavors of distributed logging and passive group commit based on Shore-MT [76], an open source database system designed specifically to remove bottlenecks on multicore hardware\(^3\). Since

\(^3\) Downloaded from https://bitbucket.org/shoremt.
different NVRAM products have varied latencies and most of them provide near-DRAM (or faster, such as STT-RAM) performance, we first use DRAM in our experiments, and then add delays to emulate the latencies for worst-case analysis. The rest of this section presents experimental setup and results.

3.4.1 Experimental Setup

We conduct experiments on a quad-socket Linux server with 64GB main memory and four Intel Xeon E7-4807 processors, each with six physical cores. With hyper-threading, the OS sees a total of 48 “CPUs”. The data used in our experiments are completely memory-resident using a tmpfs file system, to eliminate I/O effects unrelated to logging. Buffer flushing and checkpointing are still enabled, however.

Experimental Variants

We evaluate distributed logging based on two dimensions: (1) log space partitioning: transaction (TX) or page (PG) level, and (2) processor cache type: durable (D) or volatile (V). Variants that use volatile caches rely on passive group commit to protect committed work. Variants that emulate a machine with durable caches disable passive group commit as it is not needed. Based on existing work on non-volatile caches, similar performance to SRAM is expected, giving us the opportunity to measure the performance on existing hardware. We use the default write-back mode for durable cache variants, and write-combining for volatile cache variants. To focus on logging performance, we minimize other bottlenecks by enabling speculative lock inheritance (SLI) [75] and early lock release (ELR) [77] in Shore-MT for all distributed logging variants. Both flavors of distributed logging are compared to Aether [77], a state-of-the-art centralized logging scheme (denoted as “Aether”). For Aether, we enable SLI and ELR. Results from another Aether variant without SLI and ELR (denoted as “Baseline”) are also provided for comparison. For both Aether and Baseline, group commit is enabled and the log buffer size is set to 80MB. For all distributed logging variants we use 20 logs, each with a 64MB NVRAM-based log buffer. TX variants have five logs per socket and insert log records only to local logs. PG variants insert log records into the log with logID = pageID modulo 20. Records carrying no page ID are inserted into a predefined log. These configurations are completely adjustable, however, despite the differences in configurations such as log buffer size and the number of log buffers in Aether and distributed log variants, we did not observe significant differences by varying them as other contention becomes the major bottleneck.

NVRAM Performance

Some NVRAM technologies (e.g., flash backed NV-DIMMs) already promise at least comparable speed to DRAM. For such candidates, we use DRAM in our experiments. Our scheme is general enough that better performance is expected with faster-than-DRAM NVRAM. Since PCM is not widely available, we add a conservative delay of 1.2$\mu$s, longer than the maximum ($\sim$1$\mu$s) reported in most existing literature [32,97], on top of DRAM latency for emulation. The final, mature PCM DIMMs are yet to reach the enterprise market and they are constantly changing anyway. Therefore, by using DRAM and artificial delays, we only provide a lower bound for NVRAM latency based on existing hardware. The extra latency is added through a loop that busy-waits for a specified number of CPU cycles. We first convert clock time latency to CPU cycles according to our CPU’s frequency. Then for each log insertion, we embed an RDTSC instruction in the loop to read the number of cycles passed since we entered the loop, so it can break
after the specified number of cycles have passed. Since our NVRAM log buffers are write-only—a private DRAM undo buffer is maintained in each transaction—we do not introduce read delays. Recent research has shown that carefully architected DDR PCM modules could provide a write bandwidth comparable to at least 50% of that of DRAM [29]. Pelley et al. [140] also showed that bandwidth is not a concern for OLTP. Thus we do not specifically model NVRAM bandwidth in our experiments.

**Log Buffer Allocation**

For Baseline, Aether and PG-D (page-level log space partitioning with durable cache) variants, we use the default `malloc` to allocate log buffers and use write-back caching. We built a Linux device driver for PG-V variants to allocate write combining memory. We export it to user space via the `mmap` interface by reserving memory at the end of the physical address space (i.e., the last NUMA node in our experimental system). This allows us to quantify the impact of NUMA effects for page-level log space partitioning, detailed in Section 3.4.3.

For TX variants, we implement another driver which is capable of allocating memory from a specified NUMA node and marking it as write-back or write combining as requested. Libraries such as `libnuma` also provide similar functionality (e.g., `numa_alloc_onnode`). However, Linux does not provide convenient ways to mark specific ranges of virtual memory as write combining from user space. We therefore build this dedicated driver to change caching modes directly and then export the memory to user space through the `mmap` interface. Using the driver, we allocate write-back memory for TX-D variants, and write combining memory for TX-V variants.

**Benchmarks**

We run the TATP [130], TPC-B [166] and TPC-C [167] benchmarks to test the scalability of different variants. For TPC-B, we run its only transaction “Account Update”. For TATP and TPC-C, we run both the full mix and write-intensive transactions (“Update Location” in TATP and “Payment” in TPC-C) to show the general and stressed performances.

For each benchmark and variant, we report the average throughput (in thousand transactions per second, or kTPS) of five 30-second runs under different system loads (by varying the number of transaction threads). All workloads have a scaling factor of 48 to saturate the system, so internal storage engine bottlenecks are responsible for any observed performance differences. We use `perf` to collect stack traces and turn them into time breakdowns to identify the CPU cycles consumed by different components.

**3.4.2 Distributed vs. Centralized Logging**

As we have shown in Figure 3.1 on page 28, logging gradually becomes a bottleneck as system load increases, even with a state-of-the-art centralized log. Figure 3.4 compares the scalability of centralized and distributed logging when running write-intensive workloads. The y-axis shows the throughput as a function of core count on the x-axis. Due to heavy log contention, neither Baseline nor Aether scales well as the system approaches saturation.

The other lines in Figure 3.4 plot the throughput of both distributed logging variants, assuming a durable cache system. We use durable cache variants to highlight the scalability improvement brought by distributed logging (Section 3.4.3 analyzes the performance impacts of passive group commit for existing

---

volatile-cache systems). For comparison, we also include the variants with NVRAM delays (labeled as “1.2µs”, while those without these delays are labelled as “ideal”). However, in future real durable cache systems, we do not expect any such delays, even if the NVRAM is slower than DRAM, as the non-volatile CPU cache (in write-back mode) will be fast enough to hide the latency and provide at least near-DRAM performance. The performance numbers for these variants with extra delays show a lower bound and are for reference only. For TATP, Figure 3.4 shows that both TX-D and PG-D (with and without the delay) achieve near-linear scalability up to 24 threads, the number of physical cores in our system. TX-D (ideal) achieves a speedup from 1.6–3.2× over Aether. The numbers for PG-D (ideal) are slightly lower, ranging from 1.3–2.7×. Figure 3.4 shows a similar trend for TPC-B and TPC-C. As system load increases, throughput scales up more slowly due to bottlenecks outside the log manager, especially after 24 threads. The corresponding time breakdowns for TX-D (ideal) in Figure 3.5, clearly show that logging continues to perform well, but other sources of contention (e.g., contention in the buffer manager) gradually become larger for TPC-B and TPC-C. The growth of throughput in Figure 3.4 also becomes slower or even stopped. The same breakdowns for TATP do not show significant increase in contention as system load increases, explaining the superior scalability of the microbenchmark vs. TPC-B and TPC-C. In most cases, the imposed NVRAM delay shows an overhead of up to 15% for all the three workloads. However, in other cases the “1.2µs” variants actually achieved higher throughput (e.g., TPC-C with 24 threads) because the delay slowed down the system and thus reduced contention in components outside the log manager. Threads will spend more time in the log manager, resulting in fewer threads contending for other components at the same time. For example, running TPC-C’s Payment transaction under TX-D (1.2µs) with 24 threads increased 4.09% of log work, and reduced contention for the log and other components by 0.99% and 1.16%, respectively.
To show the performance of distributed logging and compare it with centralized counterparts in more ordinary cases, we repeat the same experiment with the full TATP and TPC-C benchmark mix (TPC-B only has one transaction, which is also write-intensive). Figure 3.6 shows the result for running these workloads which have less contention and more read operations. For clarity, we only show the “ideal” variants. With fewer writes, Aether scales almost equally well as both distributed logging variants for TATP, though the throughput is slightly lower. Baseline keeps a similar trend as before, and cannot scale as system load increases. For TPC-C, both Baseline and Aether scaled up to 16 threads, after which only distributed logging variants can scale. Similar to the write-intensive case we discussed earlier in this section, after 32 threads, throughput numbers stop growing or even start to drop as the system approaches full saturation. Clearly passive group commit has an effect on this phenomenon as both TX-V and PG-V have lower throughput after 32 threads. However, PG-D also sees a drop at 40 threads. As we will show in Section 3.4.3, passive group commit is not the major cause. We plot the time breakdown graph for TX-V in Figure 3.7, in which “other work” increased by \(\sim 4\%\) with 44 threads compared to with 32 threads, despite the decreased contention. More time is spent on other work which is responsible for the slowdown, but log related contention is low and does not change much as system load increases.

Compared to TX-D (ideal), PG-D (ideal) achieved lower throughput in all benchmarks for all experiments we have conducted so far, because PG-D accesses log buffers in remote memories. Profiling both TX-D and PG-D, when running the Update Location transaction from TATP with 24 threads, we observe nearly 9\% more off-core requests with PG-D (by comparison, Baseline produces 18\% more requests). Based on these results, we conclude that a distributed log can improve performance on multi-socket under both high and low contentions, and transaction-level partitioning can avoid NUMA effects for even higher throughput.

### 3.4.3 Impact of Passive Group Commit

We explore the performance impact of passive group commit, which is required to protect committed work in today’s volatile-cache machines. Passive group commit imposes overhead mostly when it signals stragglers, though we expect the signaling should be rare.\(^5\) We show the throughput in Figure 3.8 when running distributed logging variants with passive group commit on write-intensive workloads. Similar to durable cache variants, TX variants scale well as core count increases. When running TATP’s Update Location transaction, both TX-V and PG-V lose throughput when compared to the durable cache variants

---

\(^5\) In our experiments, fewer than one in 20,000 commits requires a signal, though this could vary with workload.
Figure 3.8: Scalability of distributed logging with passive group commit. PG-V cannot scale after 32 threads due to single-node log buffer allocation, but PG-V with malloc shows similar scalability to TX-P.

In Figure 3.4. The overhead ranges from 4% to 10% for TX-V (ideal). A similar trend is observed for PG-V (ideal) until the core count reaches 32, after which the performance could drop as much as 50%. TPC-B and TPC-C results show a similar trend. Compared to the TX-V variant in Figure 3.6, when running TPC-C, TX-V in Figure 3.8 does not show a performance drop at 40 threads. Note that in Figure 3.8 TX-V runs a more write-intensive workload than in Figure 3.6. Thus the extra other work incurred by the transaction mix workload (Figure 3.6) is the real major cause of the performance drop at 40 threads in Figure 3.6. Passive group commit does not incur significant performance overhead. PG-V (ideal) and PG-V (1.2µs) use our custom driver to allocate WC memory from a single NUMA node, concentrating log traffic to a single-node and taxing the machine’s interconnect. The impact becomes more severe when we impose NVRAM delays. A more sophisticated strategy, such as the one used by malloc, would stripe the allocation over all nodes to distribute memory traffic more evenly. In Figure 3.8 we also show a PG-V variant using malloc, (“PG-V (malloc)”) to estimate the real performance that passive group commit could achieve if the driver were improved to stripe the log across all nodes.

Compared to PG-V (ideal), PG-V (malloc) in Figure 3.8 shows similar scalability to TX-V (ideal), except that the numbers are slightly lower, which is in line with our results on TX-V (ideal) and PG-V (ideal). Note that as the system approaches full saturation (e.g., with 42 threads), daemon threads including the passive group commit daemon and buffer flusher start to affect performance. Therefore performance numbers after 42 threads are only for completeness. Results from PG-V (malloc) also indicate that the performance difference between write-back and write combining on write-only access is minuscule as the throughput numbers are very close before 32 threads. After 32 threads, PG-V (malloc) maintained the same trend as TX-V (ideal) for both workloads. Since we also use malloc to allocate log buffers for PG-D variants, the scalability of PG-D is similar to that of TX-D, as shown in Figure 3.4.

Based on the above results and analysis, we have shown the real overhead of passive group commit, and conclude that passive group commit is very lightweight and is suitable for current multi-socket systems. We also verify that write combining imposes little overhead when compared to write-back in terms of writes [21,69]. The performance of write combining is not a major concern in distributed logging.

### 3.4.4 Log Space Partitioning

We summarize the differences between and implications of page and transaction level partitioning. As discussed in Sections 3.4.2 and 3.4.3, transaction-level partitioning generally has higher throughput on multi-socket machines. With the high performance interconnect, OS scheduling and NUMA-aware
memory allocation, page-level partitioning only induces a small amount of overhead. However, without NUMA-aware memory allocation, page-level partitioning cannot scale as core count reaches the system limit due to the NUMA effect. Compared to transaction-level, page-level partitioning is more suited for systems which also partition data by processor locations [137, 138]. Several logs could be allocated for each data partition, turning log insert into operations local to the socket on which the thread is running.

Page-level partitioning is more naturally supported by ARIES. The implementation is also simpler. For example, GSN is not required unless passive group commit is needed. In terms of individual components, page-level partitioning requires more changes in the transaction manager, as a transaction will touch multiple logs. On the contrary, transaction-level partitioning requires more changes in the buffer manager, while changes in the transaction manager are more straightforward with the help of GSN. Based on Aether in Shore-MT, we added, modified and removed 9644 LoC in total to implement page-level partitioning, while the number for transaction-level is 10280. These numbers include GSN and passive group commit implementations. The net changes required without GSN will be fewer. The majority of code changes were simply refactoring to replace “lsn” with “gsn” in variable names; the actual implementation of distributed logging only occupied about 2kLoC (much of which was deletions from the removal of Aether).

Based on the above analysis, we argue that on multicore and multi-socket hardware, transaction-level is more preferred over page-level log space partitioning. Though the implementation is somewhat invasive, due to refactoring involved, it does not require significantly more design effort.

3.5 Summary and Discussion

NVRAM is fundamentally changing the landscape of transaction logging. Its non-volatility and byte-addressability potentially invalidate the assumption of flush-before-commit and thus enable distributed logging to ease the logging bottleneck. However, the volatile nature of existing processor caches vetoes the option of blindly replacing the centralized log with a distributed one. The increasing popularity of multi-socket hardware also poses challenges in adopting a distributed log. We have discussed and proposed solutions to these challenges, in particular log space partitioning. We have shown that a distributed log could provide near-linear scalability and up to more than 3× speedup over state-of-the-art centralized logging on multi-socket hardware. In particular, transaction-level partitioning is more favorable by allowing us to allocate log buffers from local memory, avoiding cross-socket log buffer access. Leveraging existing hardware, we have proposed passive group commit which is very lightweight to protect committed work. Finally, we expect that durable processor caches will be the ultimate solution to building reliable and high performance NVRAM-based systems.

* * *

With the logging bottleneck removed, modern OLTP engines are able to achieve high throughput and scalability. In particular, large DRAM and massive parallel processors enable one to build fast main-memory databases that are capable of processing millions of transactions per second. Despite the high performance, such systems often sacrifice functionality, i.e., the ability to efficiently ensure serializability for heterogeneous and high-contention workloads. The next chapter tackles this problem in the context of modern multi-socket hardware by proposing the Serial Safety Net (SSN), while Chapter 5 discusses solutions for future hardware with even larger scales (e.g., up to 1000 cores).
Chapter 4

The Serial Safety Net

Concurrency control (CC) algorithms must trade off strictness for performance. In particular, serializable CC schemes generally incur higher cost to prevent anomalies, both in runtime overhead such as the maintenance of lock tables, and in efforts wasted by aborting transactions. We propose the serial safety net (SSN), a serializability-enforcing certifier which can be applied on top of various CC schemes that offer higher performance but admit anomalies, such as snapshot isolation and read committed. The underlying CC mechanism retains control of scheduling and transactional accesses, while SSN tracks the resulting dependencies. At commit time, SSN performs a validation test by examining only direct dependencies of the committing transaction to determine whether it can commit safely or must abort to avoid a potential dependency cycle.

SSN performs robustly for a variety of workloads. It maintains the characteristics of the underlying CC without biasing toward a certain type of transactions, though the underlying CC scheme might. Besides traditional OLTP workloads, SSN also efficiently handles heterogeneous workloads which include a significant portion of long, read-mostly transactions. SSN can avoid tracking the vast majority of reads (thus reducing the overhead of serializability certification) and still produce serializable executions with little overhead. The dependency tracking and validation tests can be done efficiently, fully parallel and latch-free, for multi-version systems on modern hardware with substantial core count and large main memory.

We demonstrate the efficiency, accuracy and robustness of SSN using extensive simulations and an implementation that overlays snapshot isolation in ERMIA, a memory-optimized OLTP engine that supports multiple CC schemes. Evaluation results confirm that SSN is a promising approach to serializability with robust performance and low overhead for various workloads.

4.1 Introduction

Concurrency control (CC) algorithms interleave read and write requests from multiple users simultaneously, while giving the (perhaps imperfect) illusion that each transaction has exclusive access to the data. Serializable CC mechanisms generate concurrent transaction executions that are equivalent to some serial ones. This is desirable for users, because serializable executions never have anomalies (e.g., lost update and write skew) and can preserve integrity constraints over the data. Enforcing a cycle-free transaction

---

1 This chapter is based on materials that appeared in The VLDB Journal [176] and DaMoN 2015 [175].
dependency graph is a necessary condition to achieve serializability, and is the focus of this work.\footnote{Phantom protection, as we will discuss later in Section 4.6, is another necessary, but largely orthogonal issue.} Some CC schemes—such as two-phase locking (2PL) \cite{2PL} and serializable snapshot isolation (SSI) \cite{SSI}—forbid all dependency cycles to guarantee serializability, but in doing so they also forbid many valid serializable schedules.

Traditional serializable CC schemes have been either pessimistic or optimistic. In today’s environment of massively parallel, large main memory hardware, it is common for the working set—or even the whole database—to fit in main memory. I/O operations are completely out of the critical path. Existing pessimistic scheme implementations often scale poorly in this situation, due to physical contention (e.g., on centralized lock tables \cite{locktables,locktables}). Lightweight optimistic concurrency control (OCC) \cite{OCC} is favored in many recent memory-optimized systems \cite{memory-optimized, memory-optimized, memory-optimized}, but OCC is known to be unfriendly to heterogeneous workloads that have a significant amount of analytical operations and read-mostly transactions \cite{read-mostly, read-mostly}. Considering the performance impact of both kinds of serializable CC, many designs have non-serializable execution as the common case. For example, although serializable SI (SSI) \cite{SSI} has been implemented in PostgreSQL to ensure full serializability \cite{postgres-SSI}, Read Committed (RC) is still the default isolation level in PostgreSQL for performance reasons \cite{postgres-RC}, and a similar default is found in most widely-used database systems. Sometimes there is no available isolation level that guarantees serializability. Whenever an application uses transactions that may not be serializable, data corruption is a risk, so our focus is on guaranteeing serializability while reducing the performance degradation as much as possible.

Figure 4.1 illustrates the relative strictness vs. performance trade-off for several well-known CC schemes. At one extreme, strict 2PL ensures serializability but offers low concurrency because readers and writers block each other. At the other extreme, a system with no CC whatsoever (No CC) offers maximum concurrency but admits often intolerable anomalies (e.g., dirty reads and lost writes). With low performance cost, RC and its lock-based variant (RCL) offer much stronger semantics than No CC, and are often used in practice. Snapshot isolation (SI) makes a very attractive compromise, offering reasonably strict semantics and fairly high performance, while SSI offers full serializability but lowers concurrency. Fully precise serialization graph testing (SGT) \cite{SGT} allows all (and only) cycle-free executions, but is impractical as every commit requires an expensive search for cycles over an ever-changing dependency graph.
4.1.1 The Serial Safety Net in a Nutshell

The serial safety net (SSN) is an efficient, general-purpose certifier to enforce serializability on top of a variety of CC schemes, such as RC and SI. SSN does not dictate access patterns—the underlying CC scheme does that—but instead tracks dependencies and aborts transactions that might close a dependency cycle if allowed to commit. SSN admits false positives, but is much more accurate than prior practical serializable CC schemes (e.g., 2PL and SSI). As illustrated by Figure 4.1, SSN guarantees serializability with concurrency levels not drastically worse than the underlying CC scheme. In particular, SI+SSN, RC+SSN, and RCL+SSN all allow higher concurrency than 2PL and SSI. The SSN infrastructure can also be used to prevent phantoms, and so offers full protection for any CC mechanism that forbids dirty reads and lost writes. The majority of schemes meet these constraints; read uncommitted (ANSI SQL) and eventual consistency (NoSQL favorite) are perhaps the most notable exceptions, by allowing dirty reads and lost writes, respectively. SSN thus expands significantly the universe of CC schemes that can be made serializable.

SSN can be implemented in both traditional disk-based systems and recent main-memory databases. We focus on multi-version main-memory systems in this chapter. To facilitate dependency tracking, SSN requires global timestamps, which can be generated from a centralized source (e.g., a counter incremented by the atomic `fetch-and-add` instruction) or by augmenting unique thread-local counters to block-allocated timestamps from a centralized source.

The gist of SSN consists of two parts: (1) a low watermark $\pi(T)$ (formally defined in Section 4.3 on page 49) for transaction $T$ that is the commit timestamp of the earliest transaction committed before $T$ but which must be serialized after $T$, and (2) a conservative validation test that is applied when $T$ commits at time $c(T)$: if $U$ has already committed and had a conflict with $T$ (i.e., $U$ must be serialized before $T$), then $\pi(T) \leq c(U) < c(T)$ is forbidden because $U$ might also need to be serialized after $T$, forming a cycle in the dependency graph. We prove that maintaining this exclusion window suffices to prevent all cycles in the serial dependency graph, and then show how phantoms can be converted into dependency cycles so that SSN can enforce truly serializable executions in systems that otherwise lack phantom protection. We also show that $\pi(T)$ can be computed efficiently for multi-version systems.

One unique aspect of SSN is that it works in spite of bugs, omissions, or unanticipated behaviors in the underlying CC scheme, so long as the basic requirements still hold. This protection is important, because CC schemes tend to be complex to implement, and bugs can lead to subtle problems that are difficult to detect and reproduce. Unanticipated behaviors are even more problematic. For example, a read-only anomaly in SI arises only if a reader arrives at exactly the wrong moment [50]. This anomaly was not discovered until SI had been in use for many years. Assuming SSN is implemented correctly—hopefully achievable, given its simplicity—bugs or unexpected behaviors in the CC scheme that would confuse applications, will instead trigger extra transaction aborts caused by SSN. The application sees only serializable executions that preserve data integrity.

SSN is amenable to a variety of workloads and does not exaggerate the underlying CC’s favor for either reader or writer accesses. Moreover, SSN’s efficient dependency tracking and exclusion window test give the opportunity to optimize emerging heterogeneous workloads that contain a significant portion of long, read-mostly transactions: reads of stale records that are not updated recently do not have to be tracked in the transaction’s read set. This greatly reduces bookkeeping footprint and improves performance.
4.1.2 Contributions and Chapter Organization

SSN uses only local knowledge of each transaction and its direct conflicts to determine whether committing a transaction will close a potential dependency cycle. Besides the basic techniques of SSN, we further propose a generic approach to optimizing emerging heterogeneous workloads, and an efficient parallel commit protocol with minimum overhead for today’s memory-optimized, multi-version systems.

We evaluate SSN with a wide set of experiments, both in simulation and ERMIA [89], a recent OLTP system optimized for massively parallel processors and large main memory. Simulation results show that SSN works well under a wide variety of circumstances, including both lock-based and multi-version CC, mixed workloads and very high contention. Evaluation using ERMIA on a quad-socket, 60-core Xeon server shows that SSN scales as well as the underlying CC scheme. In particular, SSN’s optimization for read-mostly transactions can significantly reduce last level cache misses and perform more than 2× better than an efficient parallel SSN implementation without the optimization. Compared to SSI, SSN matches its performance for workloads with low and medium contention that do not stress the CC protocol. For high-contention workloads with retrying aborted transactions, SSN can provide more robust performance and better accuracy for both write-intensive and read-only transactions.

The rest of the chapter is organized as follows. In Section 4.2, we give background on serial dependency graphs that we use throughout the chapter to understand serializability properties. Section 4.3 discusses the design and presents a theoretical proof of the correctness of SSN. Section 4.4 gives an efficient and scalable implementation of SSN for multi-version systems, leveraging parallel programming techniques. In Section 4.5, we discuss ways of making SSN lightweight and efficient, including how we optimize read-only and heterogeneous workloads using SSN. Section 4.6 extends the SSN infrastructure to prevent phantoms for systems that are not otherwise phantom-free. We then present evaluation results of SSN in Section 4.7 and 4.8, using simulation and implementation respectively. We conclude this chapter in Section 4.9.

4.2 Serial Dependency Graphs

We model the database as a multi-version system that consists of a set of records [1]. Each transaction consists of a sequence of reads and writes, each dealing with a single record. In this model, each record is seen as a totally-ordered sequence of versions. A write always generates a new version at the end of the record’s sequence; a read returns a version in the record’s sequence that the underlying CC mechanism deems appropriate. In the model, each record exists forever, with a continually growing set of versions. In practice, obsolete versions that are no longer needed are periodically recycled to avoid wasting storage space. Insertions and deletions are represented using a special “invalid” value, for the initial version of a record that has not yet been inserted, and also for the last version of a deleted record. Insertions are updates that replace invalid versions. A deletion flags a record as invalid without physically deleting it, and the record can continue to participate in CC if needed. The physical deletion is performed in the background once the record is no longer reachable [55]. In this model, we do not explicitly model the case where a transaction reads a record it has previously written, because doing so does not add new edges to the dependency graph, i.e., no new cycles can arise. Many real systems ensure that a read will return the version that the transaction itself wrote. We note, however, that there are exceptions: certain OCC-based systems [90, 169] do not allow a transaction to read its own writes.

We first only consider the serial dependency cycles that may arise among individual records that are read and written. In the absence of insertions, preventing such cycles produces a serializable schedule.
In Section 4.6, we extend these concepts to include analogs of hierarchical locking, lock escalation, and predicate-based selection to prevent phantoms.

Accesses by transaction $T$ generate serial dependencies that constrain $T$’s place in the global partial order of transactions. Serial dependencies can take two forms:

1. $T_i \xleftarrow{w} T$ (read/write dependency): $T$ read ($T_i \xleftarrow{w} T$) or overwrote ($T_i \xleftarrow{w} T$) a version that $T_i$ created, so $T$ must be serialized after $T_i$.

2. $T \xleftarrow{r} T_j$ (read anti-dependency): $T$ read a version that $T_j$ overwrote, so $T$ must be serialized before $T_j$.

A read implies a dependency on the transaction that created the returned version, and an anti-dependency from the transaction that (eventually) produces the next version of the same record (overwriting the version that was read). A write implies a dependency on the transaction that generated the overwritten version. Accessing different versions of the same record (e.g., a non-repeatable read) within a transaction implies a serialization failure: $T_1 \xleftarrow{r} T_2 \xleftarrow{w} T_1$.

We use $T \rightarrow U$ to represent a serial dependency of either case: either $T \xleftarrow{w} U$ or $T \xleftarrow{r} U$, and we say that $T$ is a direct predecessor of $U$ (i.e., $U$ is a direct successor of $T$). Note that in the former case $x$ can be $r$ or $w$. The set of all serial dependencies between committed transactions forms the edges in a directed graph $G$, whose vertices are committed transactions and whose edges indicate required serialization ordering relationships. When a transaction commits, it is added to $G$, along with any edges involving previously committed transactions. $T$ may also have potential edges to uncommitted dependencies, which will be added to $G$ if/when those transactions commit.

Note that our notation puts the arrowhead of a dependency arrow near the transaction that must be serialized before the other. This is the reverse of the usual notation [1] but it makes the arrowhead look similar to the transitive effective ordering relation symbol we define next.

We define a relation $<$ for $G$, such that $T_i < T_j$ means $T_i$ is ordered before $T_j$ along some path through $G$ (i.e., $T_i \rightarrow \ldots \rightarrow T_j$). We say that $T_i$ is a predecessor of $T_j$ (or equivalently, that $T_j$ is a successor of $T_i$). When considering potential edges, we can also speak of potential successors and predecessors. These are transactions for which the potential edges (along with edges already in $G$) require them to be serialized after (or respectively before) $T$.

A cycle in $G$ produces $T_i < T_j < T_i$, and indicates a serialization failure because $G$ then admits no total ordering. The simplest cycles involve two transactions and two edges:

1. $T_1 \xleftarrow{w} T_2 \xleftarrow{w} T_1$. $T_1$ and $T_2$ saw each others’ writes (isolation failure).

2. $T_1 \xleftarrow{w} T_2 \xleftarrow{r} T_1$. $T_2$ saw some, but not all, of $T_1$’s writes (atomicity failure).

3. $T_1 \xleftarrow{r} T_2 \xleftarrow{r} T_1$. $T_1$ and $T_2$ each overwrote a value that the other read (write skew).

In our work, a central concept is the relationship between the partial order of transactions that $G$ defines, and the total order defined by their commit times. At the moment transaction $T$ enters pre-commit, we take a monotonically increasing timestamp, and call it $c(T)$. An edge in $G$ is a forward edge when the predecessor committed first in time, and a back edge when the successor committed first. A forward edge can be any type of dependency, but (for the types of CC algorithms we deal with, which enforce write isolation) back edges are always read anti-dependencies where the overwrite committed before the read. We denote forward and back edges as $T_1 \xleftarrow{f} T_2$ and $T_1 \xleftarrow{b} T_2$, respectively. Let us write
$T_0 \leftarrow^b T_k$ for the reflexive and transitive back edge situation where $T_0$ is reachable from $T_k$ without following any forward edges, e.g., $T_0 \leftarrow^b T_1 \leftarrow^b T_2 \leftarrow^b T_3 \ldots \leftarrow^b T_{k-1} \leftarrow^b T_k$. Note that $T \leftarrow^b T$ always holds.

We next describe a representative but not exhaustive sampling of CC mechanisms that will be used for both discussions and evaluations in the rest of the chapter:

- **Read Committed (RC).** Reads return the newest committed version of a record and never block. Writes add a new version that overwrites the latest one, blocking only if the latter is uncommitted. Allows dependency cycles but forbids isolation failures (dirty reads and lost writes).

- **Read Committed with Locking (RCL).** An RC variant (with the same types of cycles) that can be implemented with a single-version system using in-place updates. RCL is typically achieved by combining short-duration read locks with long-duration write locks. Readers and writers alike must block until the latest version is committed, but readers do not block writers.

- **Snapshot Isolation (SI).** Each transaction reads from a consistent snapshot, consisting of the newest version of each record that predates some timestamp (typically, the transaction’s start time). Writers must abort if they would overwrite a version created after their snapshot. Allows write skew anomalies, but forbids isolation failures and enforces write atomicity.

- **Serializable Snapshot Isolation (SSI).** Like SI, but forbids the “dangerous structure”: $T_1 \leftarrow^r T_2 \leftarrow^w T_3$ where $T_3$ committed first [26] (with some exceptions made for read-only transactions [141]). No cycles are possible and so all executions are serializable.

- **Strict Two-Phase Locking (2PL).** Used by many single-version systems with long-duration read and write locks. Reads return the newest version of a record, blocking if it has not committed yet. Writes replace the latest version, blocking if there are any in-flight reads or writes on the record by other transactions. No cycles are possible.

SSN can work with most realistic CC schemes that are at least as strong as RC (formal requirements are given in Section 4.3). We are especially interested in weaker CC schemes that allow atomicity failures, non-repeatable reads, write skew, and more complex cycles in $G$, including various forms of read skew (e.g., $T_1 \leftarrow T_3 \leftarrow^r T_2 \leftarrow T_4 \leftarrow^w T_1$).

### 4.3 SSN: The Serial Safety Net

In this section, we first describe how SSN prevents committing transactions that will close potential cycles in the dependency graph. We then formally prove the correctness of SSN and compare it with other serializable CC schemes.

#### 4.3.1 Preventing Dependency Cycles

Given a CC scheme that admits cycles in the serial dependency graph, SSN can be layered on top as a pre-commit protocol to abort transactions that might form potential cycles if committed. Although SSN can be overlaid on various CC schemes, we require the underlying CC scheme forbid lost writes and dirty reads (unless it is the transaction reading its own writes), which is effectively as strong as RC.
In addition to the commit timestamp $c(T)$ of transaction $T$, SSN associates $T$ with two other timestamps: $\pi(T)$ and $\eta(T)$, which are respectively the low and high watermarks used to detect conditions that might indicate a cycle in the dependency graph $G$ if $T$ is committed. We define $\pi(T)$ as the commit time of $T$’s oldest successor $U$ reached through a path of back edges:

$$\pi(T) = \min \left( c(U) : T \xrightarrow{b} U \right) = \min \left( \{ \pi(U) : T \xleftarrow{b} U \} \cup \{ c(T) \} \right)$$

The first equation captures the definition, in which $T$’s successor $U$ that overwrote versions read by $T$, committed first, forming a back edge that represents a read anti-dependency. The second, equivalent recursive equation, shows how this would be computed from only the immediate successors of a transaction in $G$, without traversing the whole graph. Note that in the recursive equation, we compute $\pi(T)$ with an initial value of $c(T)$. In systems that do not give duplicate timestamps to different transactions, there is no such $U$ that $c(U) = c(T)$. Therefore, in these systems $\pi(T) < c(T)$, and $\pi(T) = c(T)$ indicates that $T$ has no successor that committed earlier. In systems that do give duplicate timestamps, however, it is possible that $\pi(T) \leq c(T)$. In the rest of this chapter, we take the former model where timestamps are unique among transactions. Nevertheless, whether timestamps are unique does not impact the correctness of SSN; it only affects how the case $\pi(T) = c(T)$ is treated.

The values of $c(T)$ and $\pi(T)$ are fixed once $T$ has committed; $\pi$ will not change because committed $T$ only acquires new successors via forward edges, which do not influence $\pi(T)$.

The essence of SSN is a certification that prevents a transaction $T$ from committing if an exclusion window check fails for some direct predecessor $P$:

**Definition 4.1** A dependency edge $P \leftarrow T$ in $G$ (or alternatively, transaction $P$) violates the exclusion window of $T$ if $\pi(T) \leq c(P) < c(T)$.

The inequality checks whether $P$ (a predecessor of $T$ which committed first) might also be a successor of $T$ (because $P$ did not commit earlier than $T$’s oldest successor), indicating a potential cycle in $G$. When implementing exclusion window checks, we can use two observations to simplify the process. First, we need only consider predecessors that committed before $T$ (the second inequality), which means the check can be completed during pre-commit of $T$ (regardless of what happens later). Second, of those predecessors that committed before $T$, we only need to examine the most recently-committed one. Using the following definition of $\eta(T)$, an exclusion window violation occurs if $\pi(T) \leq \eta(T)$, so $T$ must abort:

$$\eta(T) = \max \left( \{ c(P) : P \xleftarrow{f} T \} \cup \{ -\infty \} \right)$$

We next illustrate visually why tracking $\pi(T)$ and enforcing exclusion windows might prevent cycles in $G$. Formal descriptions are provided later in Section 4.3.3.

Figure 4.2(a) gives a serial-temporal representation of a cycle in $G$. The horizontal axis gives the relative serial dependency order (as implied by the edges in $G$); the vertical axis gives the global commit order. In this figure, forward edges have positive slope (e.g., $T5 \leftarrow T1$), while back edges have negative slope (e.g., $T4 \leftarrow T3$). A transaction might appear more than once (connected by dashed lines, e.g., $T1$), if a cycle precludes a total ordering.

Visually, it is clear that $T1$ violates $T2$’s exclusion window because $\pi(T2) = c(T5) < c(T1) = \eta(T2)$. Figure 4.2(b) depicts information that is available to $T2$ as local knowledge. Without knowing $T1$’s predecessors, $T2$ must assume that $T1$ might also be a successor. Figure 4.2(c) demonstrates a case
where the exclusion window is satisfied: \( T_3 \) committed before \( \pi(T_x) \)—even earlier than \( T_x \)'s oldest successor—so \( T_3 \) could not be a successor and \( T_x \) will not close a cycle if committed; \( T_1 \) cannot have any predecessor newer than \( \pi(T_x) \) as that would violate its own exclusion window; any later transactions that links \( T_1 \) with \( T_x \) would suffer an exclusion window violation.

Finally, Figure 4.2(d) illustrates a false positive case, where \( T_3 \) aborts due to an exclusion window violation, even though no cycle exists. We note, however, that allowing \( T_3 \) to commit would be dangerous: some predecessor to \( T_1 \) might yet commit with a dependency on \( T_4 \), closing a cycle without triggering any additional exclusion window violations.

### 4.3.2 Safe Retry

Users submit transactions supposing they will commit, however, the underlying CC scheme might abort transactions due to various reasons, such as write-write conflicts. Ideally, the CC scheme should ensure that all transactions eventually commit (perhaps after some number of automatic retries), unless the user requests an abort.

SSN exhibits the safe retry property [141]. Suppose SSN aborts transaction \( T \) because \( U \) violates its exclusion window, and that the user retries immediately with \( T' \). Any back-edge successor \( T \) had, is a transaction \( S \) that committed before \( T \). Since \( T' \) started after \( S \) committed, \( T' \) will not read data that will be overwritten by \( S \). That is, \( T' \) will not have the same successor, and the same set of dependencies cannot form for \( T' \).

The importance of safe retry is often overlooked, and many serializable schemes do not provide this property, including 2PL (\( T' \) could deadlock with the winner of a previous deadlock) and OCC [42,169] that relies on read set validation (the overwriter could still be in progress, causing another failure). In Section 4.8, we empirically evaluate this property for SSN and other CC schemes.

### 4.3.3 Correctness

We now formally prove the correctness of SSN. Based on the database model we set up in Section 4.2, we first recall the key result of serialization theory:

**Theorem 4.1** Let an execution with schedule \( h \) have a serial dependency graph \( G(h) \) with no cycles. Then the execution is serializable\(^3\).

\(^3\) There are many formulations such as [15] and [139], the presentation with this form of dependency definition is in [1].
As mentioned in previous sections, SSN requires the underlying CC scheme forbid lost writes and dirty reads:

**Definition 4.2** Let a certifiable scheduler be any CC scheme that forbids lost writes and dirty reads (other than a transaction reading its own writes).

Definition 4.2 effectively allows any CC scheme at least as strong as RC. In particular, the underlying CC scheme is free to return any committed version from a read (not necessarily in a repeatable fashion), and can delay accesses arbitrarily.

Given a non-serializable schedule $h$ produced by a certifiable scheduler, we first identify the “dangerous” edges in its dependency graph $G(h)$ that SSN targets. We then prove that these edges exist in any dependency cycle that arises under a certifiable scheduler. We argue the correctness of SSN as follows:

**Theorem 4.2** Let $h$ be any non-serializable history produced by a certifiable scheduler. Then the dependency graph $G(h)$ contains at least one exclusion window violation.

**Proof** By the hypothesis that $h$ is non-serializable and Theorem 4.1, $G(h)$ must contain a cycle involving $n \geq 2$ transactions.\(^4\) We first name the transactions in that cycle, so that $T_n$ committed first in time: $T_n \leftarrow T_1 \leftarrow T_2 \leftarrow \ldots \leftarrow T_{n-1} \leftarrow T_n$. Because $T_n$ committed first in the cycle, its predecessor—which is also a successor—must be reached by a back edge. We can choose the lowest value of $k$ such that $T_k \leftarrow T_n$ holds. Then $\pi(T_k) \leq c(T_n)$. Further, the predecessor of $T_k$ ($T_{k-1}$, or $T_n$ if $k = 1$) must be reached by a forward edge. Combining the two facts reveals an exclusion window violation: $\pi(T_k) \leq c(T_n) \leq c(T_{k-1}) < c(T_k)$. Since we have shown that $T_k$ always exists and always has a predecessor that violates $T_k$’s exclusion window, we conclude that $G(h)$ always contains an exclusion window violation.

**Definition 4.3** A certifiable scheduler is said to apply SSN certification if it aborts any transaction $T$ that, by committing, would introduce an exclusion window violation into the dependency graph. That is, SSN forces $T$ to abort if there exists a potential edge $U \leftarrow T$ where $\pi(T) \leq c(U) < c(T)$.

**Theorem 4.3** All executions produced by a certifiable scheduler that applies SSN certification are serializable.

**Proof** By contradiction: If there is any execution of the scheduler that is non-serializable, Theorem 4.2 shows that there is an edge in the dependency graph that violates the exclusion window. However, the certification check in the scheduler does not allow any such edge to be introduced.

We next formally prove SSN’s safe retry property. Suppose SSN aborts transaction $T$ because $U$ violates its exclusion window, and that the user retries immediately with $T'$. Then the same dependencies cannot force $T$ to abort (though other newly arrived transactions could produce a new exclusion window violation for $T'$).

**Theorem 4.4** SSN provides the “safe retry” property, assuming the underlying CC scheme does not allow $T$ to see versions that were overwritten before $T$ began.

\(^4\) We ignore self loops, since our model excludes them. In reality transactions should be allowed to read their own writes.
Figure 4.3: SSN allows all schedules (a–e) that do not have “peaks,” and also “peaks” where no predecessor of $T$ violates the exclusion window. SSN rejects only case (f); other schemes tend to reject the “valleys” that arise frequently under MVCC.

**Proof** An exclusion window violation requires $U \leftarrow T \leftarrow S$, where $\pi(T) < c(U)$ and $T$ read a value which $S$ will eventually overwrite. By the definition of back edge, $S$ committed before $T$ tried to commit, therefore, before $T'$ starts. Therefore, $T'$ will read the version $S$ created (or a later one), so no anti-dependency between $T'$ and $S$ will be created. That is, the situation that caused $T$ to have an exclusion window violation will not occur for $T'$, although other dependencies might form for $T$, and another violation may occur.

### 4.3.4 Discussion

We now compare SSN with other cycle prevention schemes, and reason about their relative merits. Figure 4.3 highlights several “shapes” that transaction dependencies can take when plotted in the serial-temporal form. Of all the serializable schedules shown, SSN rejects only the last. In contrast, 2PL admits only the first (all others contain forbidden back edges). SSI always admits cases (a) and (b), always rejects (d), and often rejects (c) and (f).\(^5\) Case (e) cannot even arise under SI, let alone SSI. Thus, the improved cycle test in SSN allows it to tolerate a more diverse set of transaction profiles than existing schemes, including schedules forbidden by SI.

Figure 4.4 illustrates the accuracy of SSN in a different way using a simple schedule involving only three transactions, with time flowing downward (a). On the surface, all might appear reasonable: each transaction makes some number of reads before writing one record and committing. However, once the execution passes the horizontal dotted line, many serializable CC strategies are doomed to abort at least one transaction.

2PL will deadlock: $T_1$ reads $B$, blocking $T_2$ which in turn blocks $T_3$. Meanwhile, $T_1$ blocks attempting to write $A$, which $T_3$ has read-locked. With SI-based schemes, $T_3$’s read of $B$ will return the original version rather than the one produced in $T_2$, so there will be a back edge $T_3 \leftarrow T_2$ (as well as $T_1 \leftarrow T_2$). Thus SI-based certifiers that check for single back edges will abort at least one of the transactions, and SSI will also abort one due to the dangerous structure $T_3 \leftarrow T_1 \leftarrow T_2$ where $T_2$ committed first and $T_3$ is not read-only. In contrast, SI+SSN safely allows all three transactions to commit, with the dependency structure shown in Figure 4.4(b). SI+SSN is not a perfect certifier, as it sometimes aborts transactions unnecessarily: if $T_1$ tries to commit last (after $T_3$ commits), then SI+SSN would abort $T_1$ with a failed exclusion window test because $\pi(T_1) \leq c(T_2) < c(T_3) < c(T_1)$, even though the schedule is actually serializable. Figure 4.4(c) depicts the dependency structure for this case. Finally, suppose the concurrency control is RC+SSN and the schedule is that of Figure 4.4(a). Now, $T_3$’s read of $B$ sees

---

\(^5\) SSI allows (c) if the leftmost transaction is read-only and sufficiently old, but rejects (f) if a (harmless) forward anti-dependency edge joins $T$ with its predecessor.
4.4 SSN Protocols in Multi-Version Systems

In this section, we describe how SSN can be implemented for multi-version systems, including disk-based and main-memory optimized ones. We discuss how SSN processes read, write and commit requests. We assume that each version and transaction is associated with some storage to store the metadata SSN requires. To overlay SSN on top of a single-version CC scheme (e.g., Read Committed with Locking), one will need to store information in lock entries as proxies for the versions and keep some locks (in non-blocking modes) longer than the underlying CC would have done. This is similar to PostgreSQL’s SSI implementation [141]. We leave lock-based SSN as future work, and focus on multi-version systems.

The rest of this section first gives the basic SSN protocols that can be made parallel using latches. We then describe how the protocols can be parallelized in a latch-free manner for modern main-memory database systems.

Figure 4.4: A pathological scenario that (a) will deadlock under 2PL but will be serializable under SI; (b) the resulting serial-temporal plot when T3 commits last under SI and (c) when T1 commits last; (d) the serial-temporal plot that results under RC. The horizontal and vertical axes of (b)–(d) represent dependency and commit orders, respectively.

Despite this pathological case being simple, it gives an intuitive explanation for why SSN works well compared to other schemes: most schemes identify and reject the existence of back edges (either singly or in pairs) as a necessary condition to close cycles. However, we have seen that the “peaked” deadly structure identified earlier is a more precise cycle-closing pattern that allows SSN to ignore a large fraction of harmless back edges while still detecting all harmful ones.

As a final observation, we expect write-intensive workloads to perform better under RC+SSN than under SSI: A major source of transaction failures under SSI is temporal skew, where a transaction attempts to overwrite a version created after its snapshot. By allowing transactions to always access the latest version (except when forbidden by SSN), RC should lower the risk of encountering temporal skew in a short transaction. We show this effect in Section 4.7.4.
Table 4.1: Metadata required by SSN.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>t.cstamp</td>
<td>Transaction end time, (c(T))</td>
<td>0</td>
</tr>
<tr>
<td>t.status</td>
<td>In-flight/committed/aborted</td>
<td>in-flight</td>
</tr>
<tr>
<td>t.pstamp</td>
<td>Predecessor high watermark, (\eta(T))</td>
<td>0</td>
</tr>
<tr>
<td>t.sstamp</td>
<td>Successor low watermark, (\pi(T))</td>
<td>(\infty)</td>
</tr>
<tr>
<td>t.reads</td>
<td>Non-overwritten read set</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>t.writes</td>
<td>Write set</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>v.cstamp</td>
<td>Version creation stamp, (c(V))</td>
<td>“invalid” ((0))</td>
</tr>
<tr>
<td>v.pstamp</td>
<td>Version access stamp, (\eta(V))</td>
<td>0</td>
</tr>
<tr>
<td>v.sstamp</td>
<td>Version successor stamp, (\pi(V))</td>
<td>(\infty)</td>
</tr>
<tr>
<td>v.prev</td>
<td>Pointer to the overwritten version</td>
<td>NULL</td>
</tr>
</tbody>
</table>

4.4.1 Basic Protocols

The basic protocols of SSN require space and computation linearly proportional to the combined read/write footprints of all in-flight transactions, plus constant space per version. Each transaction should maintain its footprints using read and write sets, which contain all the versions read and written by the transaction, respectively. SSN summarizes dependencies between transactions using various timestamps that correspond to commit times. For in-flight and recently-committed transactions, these timestamps can be stored in the transaction’s context. For older transactions, the timestamps can be maintained in versions without a need to remember which committed transactions were involved. SSN supports early detection of exclusion window violations before entering pre-commit, aborting the transaction immediately if the arrival of a too-new potential predecessor (or a too-old potential successor) dooms it to failure.

Suppose transaction \(T\) created version \(V\), while transactions \(R\) and \(W\) respectively read and overwrote \(V\). Then we can define \(c(V) = c(T)\), \(\pi(V) = \pi(W)\), and \(\eta(V) = \max\left(\{c(R) : T \not< R\} \cup \{c(T)\}\right)\). These per-version timestamps maintained in each version record everything we need to implement SSN if transaction execution is single-threaded. Explicit dependency tracking (making transactions aware of other transactions) is only needed to avoid races between transactions that coexist in time, particularly those whose validation phases overlap.

Table 4.1 summarizes the metadata (along with the corresponding initial values) that SSN tracks for each transaction \(T\) and version \(V\). Version-related states persist for the life of the version, while transaction states are discarded soon after the transaction ends. Although SSN increases per-version space overhead, we note that many MVCC implementations already track some of these values.\(^6\)

Interactions with the Underlying CC

As we have discussed previously, it is the underlying CC that dictates which version a transaction should see. Therefore, SSN’s read and write protocols (\texttt{ssn_read} and \texttt{ssn_write} functions in Algorithm 1, respectively) receive a reference to the version returned by the underlying CC as a parameter. It is up to the underlying CC to employ the appropriate synchronization mechanism to guarantee correct interactions among threads. For example, in an SI implementation the worker thread could walk through

\(^6\) For example, PostgreSQL maintains the equivalent of \(v.cstamp\) and \(v.prev\). In each version, SSN takes an extra 16 bytes for \(sstamp\) and \(psstamp\), assuming 8-byte stamps. For one million versions, SSN needs in total less than 16MB of extra memory. This is likely tolerable in today’s systems with abundant memory and storage.
Chapter 4. The Serial Safety Net

Algorithm 1 SSN read and write protocols (for multi-version systems).

```python
1  def ssn_read(t, v):
2      if v not in t.writes:
3          # update \eta(t) with w:r edges
4          t.pstamp = max(t.pstamp, v.cstamp)
5
6          if v.sstamp is infinity:
7              # update \pi(t) with r:w edge
8              t.sstamp = min(t.sstamp, v.sstamp)
9          else:
10             verify_exclusion_or_abort(t)
11  
12  def ssn_write(t, v):
13      if v not in t.writes:
14          # update \eta(t) with w:r edge
15          t.pstamp = max(t.pstamp, v.prev.pstamp)
16          t.writes.add(v)
17          t.reads.discard(v) # avoid false positive
18      verify_exclusion_or_abort(t)
```

the version chain to find the latest committed version that is visible to the transaction, and then pass the desired version to `ssn_read`. The SI implementation could indicate that a version is not yet committed by storing the creator transaction ID (TID) in the version’s commit timestamp field. Readers who see a version with a TID in the commit stamp field will skip and continue to examine the next available version. Upon commit, the creator transaction will transform the TID to the real commit timestamp. Some recent systems [42,89] follow this paradigm. As a result, `ssn_read` itself needs no extra synchronization protocol. It always reads a version that is already committed and made immutable by the creator transaction.

Neither does `ssn_write` need to handle concurrent writes itself: the underlying CC determines whether a new version can be appended, possibly by latching the version chain and compare transaction/version timestamps. If the transaction successfully installs a new version, as part of the underlying CC’s write protocol, `v.prev` should point to the version that is overwritten. The transaction then proceeds to update the timestamps using `ssn_write`. The underlying CC ensures that the in-flight new version is invisible to concurrent reader transactions, e.g., by storing the creator’s TID in the version’s commit timestamp field as described earlier.

Different from the read and write protocols, SSN’s commit protocol needs proper synchronization among transactions with overlapping footprints. We discuss the details following SSN’s read and write protocols.

Read

Lines 1–11 of Algorithm 1 describe SSN’s read protocol. Besides the reading transaction `t`, it also receives a reference to the appropriate version returned by the underlying CC as a parameter. Transaction `T` will record in `t.pstamp` the largest `v.cstamp` it has seen to reflect `T`’s dependency on the version’s creator (line 4). To record the read anti-dependency from the transaction that overwrote `V` (if any), `T` records the smallest `v.sstamp` in `t.sstamp` (lines 9–10). As shown by line 7 of Algorithm 1, if the version has not yet been overwritten, it will be added to `T`’s read set and checked for late-arriving overwrites during pre-commit. The transaction then verifies the exclusion window and aborts if a violation is detected.
The transaction will transition from in-flight status to the aborted status. If the transaction is aborted, the safe retry property allows it to retry immediately, minimizing both wasted work and latency.

Note that $T$ does not track reads of versions it creates or overwrites, nor does it track reads if an overwrite has already committed (i.e., $v.sstamp$ is valid). The read and write sets are currently implemented as simple arrays/vectors in our prototype. Further, the read set does not need to be searchable in order to enforce repeatable reads: SSN automatically enforces repeatable reads because a non-repeatable read corresponds to the cycle $T \leftarrow w \rightarrow T$. While a practical implementation would be well-advised to enforce repeatable reads by less draconian means, this is a matter of performance optimization, not correctness.

Write

The write protocol is shown by lines 13–19 of Algorithm 1. Note that $v$ in $ssn_write$ refers to the new version generated by $T$. When updating a version, $T$ updates its predecessor timestamp $t.pstamp$ with $v.prev.pstamp$. We use $v.prev.pstamp$ rather than $v.prev.cstamp$ because a write will never cause inbound read anti-dependencies, but it can trigger outbound read anti-dependencies (i.e., $R \leftarrow w \rightarrow T$, in which $R$ read $V$ before $T$ overwrote it). $T$ then records $V$ in its write set for the final validation at pre-commit (line 17). If more reads came later, $T$ would update $t.pstamp$ with $v.prev.pstamp$, which were updated by readers that came after $T$ but installed the new version $v$ before $T$ entered pre-commit. Additionally, we must remove $V$ from $T$’s read set, if present: updating $\pi(T)$ using the edge $T \leftarrow w \rightarrow T$ would violate $T$’s own exclusion window and trigger an unnecessary abort. Section 4.4.2 describes how we efficiently “remove” $V$ from $T$’s read set by skipping processing $V$ when examining the read set, without having to make the read set searchable.
Commit

We divide the commit process into two phases: pre-commit and post-commit. During pre-commit we first finalize \( \pi(T) \) and \( \eta(T) \), and then test the exclusion window. If the exclusion window is not violated, \( T \) commits and the system propagates appropriate timestamps into affected versions during the post-commit phase. Pre-commit begins when \( T \) requests a commit timestamp \( c(T) \) in the in-flight status (set at transaction initialization), which determines its global commit order, as depicted by line 2 of Algorithm 2. After initializing \( c(T) \), \( T \) is no longer allowed to perform reads or writes. It then computes \( \pi(T) \), following the formula given in Section 4.3. The computation only considers \( \pi(V) \) of reads that were overwritten before \( c(T) \).

The transaction next computes \( \eta(T) \) using a similar strategy, but must account for more dependency edge types. Recall that \( T \) can acquire predecessors in two ways: reading or overwriting a version causes a dependency on the transaction that created it; overwriting a version also causes a dependency on all readers of the overwritten version. The read and write protocols account for the former by checking \( c(V) \), and pre-commit accounts for the latter using \( \eta(V) \).

Once \( \pi(T) \) and \( \eta(T) \) are both available, a simple check for \( \pi(T) \leq \eta(T) \) identifies exclusion window violations. As shown by line 14 of Algorithm 2, transactions having \( \eta(T) < \pi(T) \) are allowed to commit, transitioning to the committed status. Otherwise, the transaction would abort with an “aborted” status and remove any new versions it installed during forward processing. The \( \text{pstamp} \) and \( \text{sstamp} \) maintained during forward processing and pre-commit are discarded, as if the transaction was never processed. During post-commit, the transaction updates \( c(V) \) for each version it created, \( \pi(V) \) for each version it overwrote, and \( \eta(V) \) for each non-overwritten version it read. Post-commit is a clean-up and resource reclamation operation that does not cause extra transaction aborts. Versions created by a pre-committed transaction that is still in post-commit do not delay transactions. A concurrent transaction can access the versions and infer their timestamps by inspecting the corresponding pre-committed transactions’ metadata.

The commit protocol described above allows parallel transaction execution but itself executes serially. The caller should acquire a latch upon entering \( \text{ssn-commit} \) and release the latch before returning from the function. The restriction is due to a number of races that can arise between acquiring \( c(T) \), computing \( \pi \) and \( \eta \) for the transaction, and updating \( \text{sstamp} \) and \( \text{pstamp} \) for versions in the transaction’s read and write sets.

4.4.2 Latch-Free Parallel Commit

Latch-based serial validation imposes an unacceptable scalability penalty, as shown by recent research [81], and especially so for modern main-memory optimized systems [42, 89, 90, 102, 169]. In this section, we describe how SSN’s commit protocol can be parallelized in a latch-free manner for recent main-memory systems. Conventional disk-based systems can also benefit from our approach with appropriate adjustment to a few assumptions described later, although not as significant as in main-memory systems. In this chapter, we focus on main-memory systems.

Main-Memory OLTP Systems

The abundant amount of memory available in modern servers has lead to many recent main-memory OLTP systems [42, 89, 90, 102, 169]. These systems assume that at least the working set (if not the whole
database) resides in memory, thus allowing several important optimizations. First, a thread can execute a transaction from beginning to the end without any context switch. Heavyweight I/O operations are completely out of the critical path. In case the transaction needs to fetch data from storage, mechanisms such as Anti-Caching [40] will abort and restart the transaction when all the data needed is available in memory. Second, main-memory OLTP systems utilize massively parallel processors and large main memory more effectively, by using memory-friendly data structures and algorithms that utilize high parallelism. For example, most main-memory systems dispense with centralized locking and co-locate the locks with records [90,146]. Lock-free techniques are often used to obtain high CPU utilization [89,103].

We exploit atomic primitives provided by the hardware to devise our latch-free parallel commit protocol. In particular, we assume 8-byte atomic reads/writes and the availability of the CAS instruction; both are supported by most modern parallel processors. We also assume that during transaction execution, there will be no I/O operations on the critical path and a thread will not be re-purposed.

**Finalizing \( \pi \)**

As shown in Algorithm 2, a committing transaction \( T \) needs to calculate \( \pi(T) \) and \( \eta(T) \) in a critical section (lines 4–11). Recall that \( T \) needs to update \( t.sstamp \) with \( v.sstamp \), which is set by the transaction that overwrote \( V \). In the latch-based commit protocol (Algorithm 2), only one transaction can examine \( v.sstamp \) at the same time, and no concurrent write to \( v.sstamp \) is allowed. A latch-free protocol, however, must account for transactions that are concurrently committing: \( v.sstamp \) might be changed at any time by the overwriter \( U \), which might have acquired a commit timestamp earlier or later than \( T \)'s.

In case \( U \) acquired a commit timestamp that is earlier than \( t.cstamp \), \( T \) should update its \( t.sstamp \) with \( U \)'s successor low watermark (i.e., \( u.sstamp \)) if \( u.sstamp \) is smaller than \( t.sstamp \).

As we have discussed in Section 4.4.1, an SI implementation can indicate that a version is not yet available for reading by storing the owner transaction’s TID on the version’s cstamp field. This approach is applicable to solving the above problem as well: an updater \( U \) should install its TID in the overwritten version \( V \)'s sstamp after it successfully installs a new version (i.e., as of the write before entering pre-commit). Our implementation does this in ssn_write. \( U \) then changes \( v.sstamp \) to contain \( u.sstamp \) during \( U \)'s post-commit phase. Consequently, a concurrent transaction might read an sstamp field and see a TID when reading a version or committing. For the former case, the transaction simply treats it in the same way as if \( v.sstamp \) contained \( \infty \) in ssn_read (i.e., no overwrite yet) and adds the tuple to its read set. Lines 9–18 of Algorithm 3 show how the latter case is handled in our latch-free commit protocol. After \( T \) detects a TID in \( v.sstamp \) (line 9), it will obtain \( U \)'s transaction context (line 10) through a centralized transaction table indexed by TIDs. Note that since \( v.sstamp \) might be changed to contain the overwriter’s commit timestamp from its TID at any time, at line 8 we must read \( v.sstamp \) into \( v.sstamp \) and determine if we should proceed to line 10 using \( v.sstamp \). Recall that by Definition 4.1, \( \pi(T) < c(T) \). So if \( U \) has acquired a commit timestamp earlier than \( t.cstamp \), \( T \) has to wait for \( U \) to conclude so that \( U \)'s successor low-watermark is stable. As line 12 shows, if \( U \) has not entered pre-commit, then it will have a commit timestamp larger than \( T \)'s. Otherwise, we spin at line 13 until \( u.cstamp \) contains \( U \)'s commit timestamp. Note that such spinning is necessary, because

---

7 Storing TID (before the overwriter finalizes) in the overwritten version’s sstamp also eases the removal of updated versions from the read set (see the end of Section 4.4.1). When iterating the read set, the updater \( T \) simply skips versions whose sstamp points to \( T \)’s own TID.

8 There exist viable approaches other than spinning for \( U \)'s sstamp to become stable; e.g., \( T \) could deduce \( U \)'s low-watermark by helping with \( U \)'s pre-commit phase by iterating over \( U \)'s read set. In experiments, we use spinning due to its simplicity and lightweightness.
acquiring a commit timestamp (line 3) and storing it in u.cstamp are not done atomically by a single instruction. For example, next_timestamp itself might draw a timestamp from a centralized counter using an atomic fetch-and-add instruction. Therefore, transactions entering pre-commit must first transition to a “committing” status, and then obtain a commit timestamp (lines 2–3). As a result, upon detecting $U$ has entered pre-commit (i.e., status is not INFLIGHT), u.cstamp is guaranteed to contain a valid commit timestamp eventually. Note that this “committing” status is not needed in the latch-based commit protocol (Section 4.4.1): only a single thread can execute $\text{ssn.commit}$ at a time; a transaction’s status is not exposed to other concurrent transactions.

After obtaining u.cstamp, the protocol continues to check if u.cstamp is smaller than t.cstamp. If so, $T$ needs to find out $U$’s final result (line 16): if $U$ indeed pre-committed, $T$ then updates t.sstamp with u.sstamp to finalize its successor low-watermark (lines 17–18). If, however, $U$ acquired a commit timestamp later than t.cstamp or has not even started its pre-commit phase, $T$ can continue to process the next element in the read set without spinning on $U$ at line 16.

Finalizing $\eta$

We first note a fundamental difference between v.sstamp and v.pstamp that determines how their calculations can be parallelized: during the lifetime of $T$, $V$ could only have at most one successful overwriter, i.e., the transaction that installed a new version of $V$ and committed. Before $T$ enters pre-commit, however, multiple concurrent reader transactions (denoted as “readers” for simplicity) could have read $V$ during $T$’s lifetime. Each of these concurrent readers will need to update v.pstamp as shown by lines 16–17 of Algorithm 2 (the latch-free version is described later). In a latch-free commit protocol, as a result, $T$ will have to take into consideration all possible concurrent readers that have obtained a commit timestamp earlier than t.cstamp. Essentially v.pstamp becomes an array of the commit timestamps of all the readers of $V$.

A simple implementation might convert v.pstamp directly to an array of timestamps, one per predecessor, bloating the amount of metadata needed per version. To make the readers-tracking efficient, we use a bitmap to summarize all of $V$’s readers. Each bit corresponds to one reader transaction. As we have discussed in Section 4.4.2, in most main-memory databases, transactions are rarely delayed (e.g., by I/O operations). A worker thread processes only one transaction at a time, and there is roughly one worker thread per CPU core. We use this fact and correspond the $i$-th bit in the bitmap to thread/transaction $i$. Whenever a thread reads $V$ on behalf of a transaction $R$ for the first time, the thread registers itself by setting the corresponding bit in v.readers. $R$ clears the bit after it concludes.

Figure 4.5 shows an example of three threads, each executing on behalf of a different transaction. In this example, Thread 1 created tuple version v1 and has already committed. Thread 0 appended a new version, v2, after v1 but has not yet entered pre-commit. At the same time, Thread 2 read v1 and registered itself in v1.readers by setting the third least significant bit. As a result, when Thread 0 tries to commit, it will be able to locate the transaction being executed by Thread 2 by following v1.readers and locate its cstamp in the transaction-thread mapping table.

With a readers bitmap in each version, $T$ is able to examine all the concurrent readers of $V$ to finalize t.pstamp. The details are described by lines 23–32 of Algorithm 3. For brevity, in the algorithm we omit the details of extracting the thread ID from the readers bitmap. Our current implementation uses the bit scan reverse (BSR) instruction available on x86 platforms [72]. As shown by lines 24–30, for each overwritten version, $T$ examines and waits for each reader that acquired an earlier commit timestamp to
Algorithm 3 Latch-free SSN commit protocol (for multi-version systems).

```python
def ssn_parallel_commit(t):
    t.status = COMMITTING
    t.cstamp = next_timestamp() # begin pre-commit

    # finalize \pi(T)
    t.sstamp = min(t.sstamp, t.cstamp)
    for v in t.reads:
        v_sstamp = v.sstamp
        if is_TID(v_sstamp):
            u = get_transaction(v_sstamp)
            # obtain u.cstamp
            if u.status is not INFLIGHT:
                spin_while(u.cstamp == 0)
                if u.cstamp < t.cstamp:
                    # wait for U to finish pre-commit
                    spin_while(u.status == COMMITTING)
                    if u.status == COMMITTED:
                        t.sstamp = min(t.sstamp, u.sstamp)
            else:
                t.sstamp = min(t.sstamp, v.sstamp)
    # finalize \eta(T)
    for v in t.writes:
        for r in v.prev.readers
            if r.status is not INFLIGHT:
                spin_while(r.cstamp == 0)
                if r.cstamp < t.cstamp:
                    spin_while(r.status == INFLIGHT)
                if r.status == COMMITTED:
                    t.pstamp = max(t.pstamp, r.cstamp)
    # re-read pstamp in case we missed any reader
    t.pstamp = max(t.pstamp, v.prev.pstamp)
    verify_exclusion_or_abort(t)
    t.status = COMMITTED # post-commit begins

    for v in t.reads: # update \eta(V)
        pstamp = v.pstamp
        while pstamp < c.cstamp
            if CAS(v.pstamp, pstamp, t.cstamp):
                break
            pstamp = v.pstamp
    for v in t.writes:
        v.prev.sstamp = t.sstamp # update \pi(V)
        # initialize new version
        v.cstamp = v.pstamp = t.cstamp
```

finish pre-commit, using the same spinning machinery introduced earlier for finalizing \( \pi \). If the reader successfully committed, \( T \) will update \( t.pstamp \) using \( r.cstamp \).

If we track concurrent readers in an array, garbage collection is needed to remove unneeded readers metadata from \( V \) (after the overwriter committed). The list of readers serves as a history of all transactions that have read \( V \). It suffices for \( T \) to go though the list for finalizing \( \eta \).
Figure 4.5: The bit positions in the readers bitmap serve as indexes to the centralized transaction table which records details on the transaction that is being run by each thread (we discuss the use of the “last cstamp” field later in Section 4.5.3).

In the bitmap-based approach, however, \( T \) has to make sure the reader is indeed the predecessor that read \( V \). It is possible that the real reader \( R \) has already left and the same bit position now points to a totally different transaction, which may or may not have read \( V \). As a conservative estimate, \( T \) has to also consult \( v.pstamp \) to catch such cases after going through all the concurrent readers using \( v.readers \) (line 32). Theoretically, this approach could make \( \eta(T) \) larger (hence more false positives) because a newer reader might update \( v.pstamp \) with its commit timestamp. In practice, our evaluation in ERMIA reveals that the benefits of using a bitmap outweighs this drawback.

**Post-Commit**

After \( \pi \) and \( \eta \) are finalized, \( T \) tests the exclusion window and aborts if necessary (line 34 in Algorithm 3). \( T \) then starts the post-commit phase to finalize the creation of new versions it wrote and timestamps of existing versions it read. As lines 37–42 of Algorithm 3 show, \( T \) will have to compete with other readers to set \( v.pstamp \), so that \( v.pstamp \) is no less than \( t.cstamp \). Finalizing \( v.sstamp \) is straightforward: \( T \) simply updates it with \( t.cstamp \) and change its type from “TID” to “timestamp” (line 45). The initialization of new versions (line 47) is the same as the serial commit protocol described in Algorithm 2.

### 4.5 Reducing SSN Overheads

SSN requires space and time proportional to the transaction’s footprint. The metadata (e.g., \( \pi(T) \) and \( \eta(T) \)) associated with each version and transaction incurs more storage overhead, and post-processing requires time proportional to the amount of transaction-private state kept. SSN requires pre-commit work proportional to the combined size of the read and write sets. In particular, the work required for examining the read set will become a concern for long, read-only and read-mostly transactions.

In this section, we first discuss how SSN can leverage features that are available in most existing systems to reduce some of the above mentioned overheads. We then propose two optimizations specifically
designed to reduce the overhead of handling reads. We adapt the safe snapshot [141] to free read-only transactions from dependency tracking, and propose an optimization for read-mostly workloads that can avoid tracking reads of cold data. With these two optimizations, the vast majority of read-tracking is eliminated, while serializability is still guaranteed. The work required at commit time becomes much less for (usual) cases where the write set is much smaller than the read set, allowing a high-performance implementation of SSN.

### 4.5.1 Leveraging Existing Infrastructure

Out of the four machine words SSN maintains in each version, most MVCC implementations are already tracking two of them: the version creation timestamp \((v.cstamp)\) and a pointer to the overwritten version \((v.prev)\). These are respectively needed to control snapshot visibility and to allow transactions to retrieve the appropriate version. SSN can therefore utilize existing infrastructure, leaving only \(v.pstamp\) and \(v.sstamp\) as new overheads. However, we observe that SSN never needs both of these two values at the same time. The \(pstamp\) is set at version creation and updated by any reader that commits before the version is overwritten. No transaction will access \(v.pstamp\) once an overwrite of \(V\) commits. Meanwhile, the overwriting transaction sets \(v.sstamp\) when it commits, and all subsequent readers will use it to update their own successor timestamps. We can thus store both fields in a single machine word. If the remaining machine word is still objectionable, further space savings could be achieved in implementation-specific ways (such as storing a delta that occupies fewer bits), but we will not discuss such approaches further here.

As shown respectively by lines 14 and 28 in Algorithms 2 and 3, a transaction \(T\) is considered “committed” if it survived pre-commit. The versions \(T\) wrote immediately become visible to other transactions (depending on the underlying CC’s visibility policy). Before \(T\) finishes its post-commit phase, readers can use the TID stored in the version written by \(T\) to look up \(T\)’s status and complete the read if the underlying CC allows; the indirection is only used until post-commit converts the TID to a proper timestamp in each version (described in Section 4.4).

### 4.5.2 Safe Snapshots and Read-Only Queries

In systems that can provide read-only snapshots, including SI-based and some single-version systems [87, 90, 169], SSN supports a variant of the “safe snapshot” [141]: a transaction known to be read-only can avoid the overhead of SSN completely, by using a snapshot that the system guarantees will not participate in any serial anomalies.

The original safe snapshot design was a passive mechanism: a query takes a snapshot and then waits until all in-flight transactions have ended, while monitoring the system for unsafe accesses. If no unsafe accesses occurred before the last in-flight transaction committed, the snapshot is deemed “safe” and can be used without further concern. This approach requires tracking transaction footprints after commit, and can lead to long delays that make it most suitable for large read-only queries executing for tens of seconds or longer.

We instead propose an active mechanism: when requested, the system forcibly takes a safe snapshot, with its timestamp stored in a global variable. SSN treats the snapshot as a transaction that has read every record in the database, inflicting a read anti-dependency on all update transactions that were in-flight at snapshot creation time. Update transactions can still overwrite versions in the safe snapshot,
but will abort if they also take a read anti-dependency on a version created before the snapshot. Reads not using the safe snapshot are unaffected by it. Simulations suggest that active safe snapshots have minimal impact on abort rate, even under heavy contention, unless the time between safe snapshots is less than the expected duration of an update transaction (see Section 4.8 for details).

Ports and Grittner [141] also describe a read-only optimization for SSI that applies to SSN over SI: a transaction that enters pre-commit with an empty write set can set $c(T)$ to its snapshot time, thus keeping

\[ v.pstamp \]

smaller and reducing (sometimes significantly) the likelihood that a subsequent overwrite will trigger an exclusion window violation. Unlike with SSI, however, implementing this “read-only” optimization with SSN actually protects update transactions from read-only predecessors. Protecting writers is more helpful anyway, as read-only transactions would normally use the very lightweight safe snapshot mechanism we propose above.

### 4.5.3 Read-Mostly Transactions

SSN relies on version stamps to implicitly track (part of) the dependency graph to test exclusion window violations. As we have discussed previously, this mandates the tracking of full transactional footprints. Reads that are not overwritten upon access have to be kept in the transaction’s read set for verification later at pre-commit (line 7 of Algorithm 1); writes also have to be tracked and get finalized at post-commit.

On the one hand, tracking writes is usually not a concern for OLTP workloads: compared to the amount of reads a transaction performs, writes are usually the minority, unless the transaction is write-heavy (in which case, however, it is usually short as well). On the other hand, emerging heterogeneous workloads feature even more reads per transaction, i.e., read-mostly transactions [89]. It is not uncommon in these workloads that a much longer scan or series of reads are mixed with a small but non-empty write set. Tracking and validating a large read set dominates the cost of SSN because the write set is tiny by comparison. As we discuss in Section 4.8.4, examining each of these reads during pre-commit is a major potential source of last level cache misses that drags down the system’s performance.

It is worth noting that because these read-mostly transactions’ read set is much larger than their write set, it is unlikely for most reads to be overwritten by concurrent update transactions. We leverage this fact to optimize read-mostly transactions: versions that are not overwritten recently (governed by a threshold) are deemed “stale” and are not tracked in the read set.

Eliminating the tracking of (potentially long) reads reduces the burden on readers at pre-commit to verify their reads, however, this also brings challenges for SSN to finalize $\pi$ and $\eta$ for testing exclusion window violations. First, writers are unable to obtain up-to-date predecessor status because the readers that skip read tracking will not update the $pstamp$ of each read version. Second, it becomes impossible for readers to check whether the read versions are overwritten by concurrent writers at pre-commit, because they are not tracked at all. This fact has profound impact on the parallel commit protocol: the happens-before relationship we relied on (when finalizing $\eta$) will not hold anymore. In Algorithm 3 (line 27), an updater only needs to wait for the commit result of the readers who entered pre-commit earlier; those who entered pre-commit later than the updater will spin on the updater (successor), if the updater has not finished pre-commit yet. When certain reads are not tracked, there is no way for a reader to spin on its successor who entered pre-commit earlier: it does not even know such updates exist. Allowing a reader to commit without properly tracking its successors could lead to non-serializable execution.

To solve these problems, we again leverage the fact that main-memory systems execute a transaction on a single thread from beginning to end without migrating among threads. We allow read-mostly
transactions to commit without having to stamp each read version, but instead require each thread record the \texttt{cstamp} of the read-mostly transaction on whose behalf the thread has committed. As shown in Figure 4.5, the reader puts its \texttt{cstamp} in the \texttt{last.cstamp} field in its private entry in the translation table upon commit. The \texttt{last.cstamp} field essentially serves as a proxy that summarizes the commit stamps of all the read-mostly transactions on the same thread. Because of the lack of read tracking, readers will likely have larger $\pi$ values, leaving more back edges unaccounted for and becoming more unlikely to be aborted. Since readers might not update \texttt{pstamp}, it then becomes the updater’s responsibility to detect non-serializable schedules.

At pre-commit, the updater examines the readers bitmap and uses set bits to find the status of each thread. Note that a reader still indicates its existence in the readers bitmap upon version access, but without clearing it upon conclusion—the read is not tracked in the first place. The updater needs to consult \texttt{last.cstamp} to find out the last committed reader’s \texttt{cstamp} when calculating \texttt{t.pstamp}.

We note, however, \texttt{t.pstamp} calculated in this way is conservative and can admit false positives: a reader that set but did not clear the bit position in \texttt{v.readers} might lead the updater to inspect another completely irrelevant transaction—one that has non-overlapping footprints—and cause unnecessary aborts. Suppose transaction $T$ being executed by thread $t1$ read $V$ without tracking it. $T$ would have set the bit position in \texttt{v.readers} and committed by setting $t1$’s last \texttt{cstamp} to $t1$’s \texttt{cstamp} without clearing the bit in \texttt{v.readers}. After $T$ committed, transaction $U$ running on thread $t2$ overwrote $V$ and entered pre-commit. Suppose $t1$ now starts another transaction $R$ whose footprint does not overlap with $T$ or $R$. During pre-commit, $U$ would follow \texttt{v.readers} to find $R$, because it inherited the bit that was used by a previous reader $T$, although $R$’s footprint does not overlap with $U$’s. According to Algorithm 3, depending on the final $\pi$ and $\eta$ values calculated, $U$ might abort unnecessarily. We expect that such false positives are not a major concern for workloads with significant portions of long, read-mostly transactions where reads are the majority.

Using a thread-private \texttt{last.cstamp} as a proxy that accumulates \texttt{pstamp}s solves only half of the problem: an updater can account for read-mostly transactions that entered pre-commit earlier, but not those that entered pre-commit later. Recall that a read-mostly transaction might not spin on its successor (line 20 of Algorithm 3) with a smaller \texttt{cstamp} because of the lack of read tracking. Consequently, the updater (as a successor) will then have to figure out the read-mostly transaction’s state when it discovers that a concurrent reader exists through the readers bitmap. Otherwise, the updater would have to blindly abort, which will make it hard to commit write-intensive transactions that have overlapped footprint with read-mostly transactions, especially when the read-mostly transaction is much longer.

Therefore, it would be desirable for the writer to update the read-mostly transaction’s \texttt{sstamp} (using the updater’s \texttt{cstamp}) during pre-commit; the reader proceeds as usual and tests for exclusion window violation at the end of pre-commit. We employ a lightweight locking mechanism for the \texttt{sstamp} to guarantee correctness: the most significant bit (MSB) of \texttt{sstamp} serves as a lock; the \texttt{sstamp} value updated when its MSB is unset is guaranteed to be taken into account by the reader. The updater issues a \texttt{CAS} instruction to update the reader’s \texttt{sstamp} with its \texttt{cstamp}, expecting the MSB is 0. The reader should atomically set the MSB of \texttt{sstamp} to 1 (e.g., by using an atomic \texttt{fetch-and-or} instruction) right before it tests for exclusion window violation at the end of pre-commit. An updater that failed the \texttt{CAS} because the MSB is set will have to abort. In this way, we reduce unnecessary aborts of updaters, although a greater number of read-mostly transactions might be aborted than without this lightweight machinery. Our empirical evaluation in Section 4.8.4 reveals that this effect is minimal, and SSN can still
achieve a transaction breakdown that is close to the specification under a variant of the TPC-E [168] benchmark.

### 4.6 Locks and Phantom Avoidance

The description of SSN in the previous section works with per-transaction read sets and write sets, with the assumption that these sets contain versions of records. To ensure full serializability, e.g., repeatable counts, we need phantom protection, or the ability to prevent insertions that would change the results of an uncommitted query. Systems based on pessimistic concurrency control employ several useful concepts such as hierarchical locking and lock escalation to reduce tracking overheads, plus key and predicate locking approaches [46, 58, 125] for phantom prevention in ordered indexes such as B-trees. SSN is compatible with those mechanisms to prevent phantoms and so guarantee full serializability. To the extent that the underlying CC implementation is already phantom-free (as is the case for many recent systems [76, 89, 90, 169]), we will not need to redo them in SSN. Otherwise, the following subsections describe how to incorporate phantom detection into the SSN protocol.

#### 4.6.1 Hierarchical Dependency Tracking

We first adapt the idea of hierarchical locking to SSN’s dependency tracking needs. In a traditional lock-based system, the database is organized as a hierarchy: schemas, tables, pages, and records. Transactions acquire a concrete lock on the finest-grained object that suits their needs, and intention locks on the object’s parents in the hierarchy. With a hierarchical locking scheme in place, lock escalation also becomes possible: a transaction can choose to replace a large number of fine-grained locks with a single coarse-level lock, trading off reduced tracking overhead for an increased risk of conflicts.

We can adopt the same philosophy in SSN: a transaction that will read the majority of a table can acquire a single read (R) lock on the table, and only needs to update the table-level pstamp. Meanwhile, updating transactions acquire IW and W locks on the table and individual records, respectively. They update only v.sstamp but must check both table- and version-level pstamps to detect all conflicts. Table 4.2 summarizes the pre-commit checks and post-commit updates required when the system supports intention modes.

If update contention by readers is a concern, either the lock or the corresponding pseudo-version can be replicated, following the “super-latching” feature of SQL Server [121]. A reader (common case) can then update only one of many sub-versions, with the trade-off that a writer (infrequent) must examine all of them.
4.6.2 Predicates and Phantoms

In a database with no installed indices, the hierarchical lock system described above detects all phantoms: any scan—no matter how selective its predicates—must access the entire table, and the resulting table $R$ lock will conflict with the $IW$ locks of both inserts and updates. However, predicates involving a (partial) index key mean finer-grained range scans that access only a portion of the table. Phantom protection can be achieved in these cases by locking the gaps between keys that fall inside the range being scanned.

Several gap-locking schemes have been proposed [106, 125], and any of them could be adapted for use with SSN. We describe here a variant of the scheme due to Graefe [54], where each lock is a composite that can independently reference a particular key and/or the gap that follows that key. Both keys and gaps can be held in read and write mode, with conflicts tracked in piecewise fashion. For example, pairing $W/N$ (key-write, gap none) with $N/R$ (key none, gap read) does not imply any dependency edge, but $W/N$ and $R/R$ implies an edge because both transactions accessed the key (there is no conflict on the gap). The full action/mode table can be generated mechanically (component by component) using Table 4.2 as a starting point, so we do not reproduce it here.

With locks that cover key/gap pairs, SSN can prevent phantoms without abandoning the notions of read and write sets: when a transaction inserts into an index, its write set contains a version for the key (probably its index entry) associated with either a $W/N$ or $N/W$ lock, depending on whether the key was already present. Meanwhile, the read set of a range-scan transaction contains index entries it read, each associated with $R/N$, $N/R$, or $R/R$ locks, depending on whether key, gap, or both fall within the scan’s endpoints. From there, the normal SSN machinery will see these new “reads” and “writes”, and check for exclusion window violations.

4.7 SSN in Simulation

We implement the SSN protocol from Section 4.3 in a discrete event simulator\(^9\) to examine SSN’s accuracy and impact over a wide variety of transaction profiles, contention levels, and schedules. We are especially interested in the impact of contention, interference among readers and writers in a mixed workload, and the impact of active safe snapshots on writer abort rates. In the next section, we implement parallel, latch-free SSN in ERMIA [89] to measure actual commit and abort rates with variants of the TPC-C and TPC-E benchmarks.

4.7.1 Simulation Framework

We have implemented in Python a discrete event simulator designed specifically to evaluate CC schemes. We use it to compare the CC schemes listed in Section 4.2, with and without SSN. The simulator allows us to quantitatively compare supported concurrency levels and abort rates of different concurrency control models. It also exposes anomalies that would indicate potential design flaws. The simulator was invaluable not only in performing evaluations and isolating bugs in the various models, but also in driving the discovery and proof of SSN in the first place.

In all models, the simulator serializes write conflicts by blocking; 2PL and RCL also block readers that conflict with writers. To bound delays and avoid deadlocks, we apply a variant of wait depth limiting (WDL) [162]: the system aborts any transaction that attempts to block on a predecessor that has already

Chapter 4. The Serial Safety Net

blocked. Because all deadlocks necessarily involve transactions blocking on blocked transactions, using WDL also obviates the need for deadlock detection. Under this arrangement, performance degrades much more gracefully under contention than it would otherwise. This is especially important for 2PL, which tends to “freeze up” once the combined transactional footprint of blocked transactions encompasses a majority of the working set. Without WDL, the system can suddenly enter overload and the expected wait times spike upwards by several factors, increasing the aggregate transactional footprint even further in a vicious cycle. With WDL, 2PL achieves drastically better performance than is traditionally reported, both in terms of latency and completion rate. Meanwhile, the effect on non-locking schemes is minimal. Even under the most extreme contention, RC—whose failures are all due to WDL—has a commit rate better than 90%.

The simulation framework provides basic support for statistics and monitoring, automatic detection of serial anomalies, scheduling, and multi-versioned data access. Pluggable database models then implement the specific CC schemes, including 2PL and RCL (which simply choose not to return overwritten versions). The base simulator comprises ~1200 LoC, and most models require 200-300 additional LoC (SI and 2PL are extreme cases, at 80 and 400 LoC, respectively).

To ensure runs using different CC methods are comparable, we use an open queuing system: each client submits requests at predetermined times (at intervals roughly equal to expected transaction latency), independent of previous requests. This models the real world of connection concentrators and users who do not coordinate with each other before submitting requests. Thus, if two simulations are started with the same random seed, the same transactions will be offered at precisely the same times for both, independent of delays imposed by the CC model in use. Thus, any difference in throughput, relative latency or abort rates is due to the models themselves, not differences between transactions offered. Further, exact reproducibility allows standard test case reduction tools to isolate problems from a large simulation trace.

Finally, we point out one caveat: although the simulator models transaction execution times, the low-fidelity timing model does not account accurately for overheads and bottlenecks that would arise in a real system implementing these CC schemes. We present the results only to show the relative timing and concurrency characteristics of different CC schemes. A later section presents results for one implementation of SSN in ERMIA, but an exhaustive performance study of optimized implementations in multiple database engines is outside the scope of this thesis.

4.7.2 Microbenchmark Description

Our simulated evaluations use an enhanced version of the SIBENCH [26] microbenchmark. The database consists of a single table and a fixed number of records. Each record contains a single attribute that stores the TID of the transaction that last wrote to it. Each transaction makes a random number of accesses, selected uniformly at random from a tunable range of valid footprint sizes. The last \( m \) of those accesses are writes (with \( m \) being a workload parameter). Repeated reads, repeated overwrites, and blind writes are all allowed. Mixed workloads can be emulated by instantiating multiple client groups with differing parameters (e.g. to mingle large read-only queries with short write-intensive transactions).

The benchmark logs the w:w or w:r dependencies implied by each access, using the TID stored in each record to identify the predecessor. After the run completes, a post-processing step reconstructs the r:w anti-dependency edges and tests the resulting graph for strongly connected components (SCC). To avoid blaming one cycle on multiple transactions (and to allow blaming multiple cycles on one transaction), we
only report transactions in an SCC as serialization failures if they also fail the exclusion window test (every SCC is guaranteed to contain at least one such failure). This pruning strategy is quite effective in practice, flagging only 1–2 transactions from a typical SCC involving 2-20 transactions, or up to a dozen in an SCC with hundreds of transactions. Nevertheless, we recognize that this strategy overestimates the true number of serialization failures, because SSN admits false positives.

Offline cycle testing is important for two reasons. First, it is completely independent of the concurrency control mechanism used; the simulator does not trust any CC scheme to be correct (with or without SSN). Second, long chains of r:w anti-dependency edges can produce serialization failures that reach arbitrarily far back in time, and a test with a limited horizon would fail to detect such cycles.\textsuperscript{10}

### 4.7.3 SSN and Other Schemes Under Contention

Our first experiment, shown in Figure 4.6, calibrates expectations. Each transaction makes between 8 and 12 accesses, with 25% of them being writes. We fix the number of clients at 30. The figure shows completion rates of single-version (left) and multi-version (right) CC schemes as the database size varies along the log-scale horizontal axis. Contention decreases as the database size increases. For reference only, we show the (non-serializable) commit rates of SI, RC and RCL. We also show the effective commit rates of those schemes if we subtract off the number of serialization failures the simulator reports, given as SI/SER, etc. Note that the numbers for RCL/SER and SI/SER are highly unrealistic—requiring an offline oracle to compute—and are for reference only. However, the difference between SI/SER and SI, etc., provides information on how many of the executions are committing with potential anomalies when a non-serializable scheme is used.

We make two observations: first, SI suffers a lower commit rate because transactions cannot overwrite versions outside their snapshot. This weakness extends to SSI and SI+SSN as well. Second, the number of actual serialization failures remains quite low until contention becomes severe, and gives a sense of the false positive rates the other schemes produce (quite high for SSI, very low for RCL+SSN). Finally, we note that protecting weaker CC schemes (RC and RCL) with SSN yields significantly higher completion rates than any other approach, across the full range of contention. RCL+SSN, in particular, sees 90% or better completion rates until the database size drops below 400 records. SSI passes that point at 1600 records. Note that this workload averages an aggregate transactional footprint of 300 records at any given

\textsuperscript{10} For example, consider $T_1 \xrightarrow{w} T_n \xrightarrow{r} \cdots T_i \xrightarrow{r} T_1$, where each $T_i$ begins just before $T_{i-1}$ commits.
moment. For a 100-record database, RCL+SSN has a completion rate above 50%, even though three or more transactions compete for each record. A completion rate of above 50% is relatively high for this setting, considering that every record a transaction reads will have an active writer with high probability.

Overall, these results indicate that, for these workload parameters, a database smaller than 500 records suffers severe contention while one larger than 5,000 records is nearly contention-free (though some schemes have non-negligible abort rates even then).

### 4.7.4 Transactions with Varying Write Intensity

One of the key benefits of multi-version schemes is that reads and writes need not block each other. A secondary benefit—for read-only queries at least—is the ability to access a stable snapshot. However, any transaction that makes at least one update suffers a temporal skew under SI, where all reads occur at the start of the transaction and all writes take effect at commit time. Figure 4.7 illustrates this vulnerability: we run 30 clients, each making 100 uniformly random accesses against a database containing 100k records. As the fraction of accesses which are writes increases to 100% along the horizontal axis, the SI and non-SI schemes are clearly differentiated, with the latter all converging to a completion rate nearly double that of the SI-based schemes. The SI schemes all converge to the same performance because temporal skew is the primary cause of transaction failure. In contrast, schemes that always read the latest committed value (2PL, RC, RCL) are much less vulnerable to temporal skew and consistently achieve better completion rates. Note that the workload should have low-contention: all clients together have an aggregate transactional footprint covering at most 3% of the database at any given time.

### 4.7.5 Interference between Readers and Writers

So far, all our simulations involve fairly update-intensive workloads, with transactions of uniform size and no read-only queries. Lock-based approaches tend to outperform more optimistic approaches under updates, in no small part because MVCC is of little use to a writer (who must always overwrite the newest version of a record). Indeed, we have seen that 2PL performs quite competitively in update-intensive workloads. However, 2PL interacts very poorly with large read-only transactions, as demonstrated in Figure 4.8. Here, we model a system with 10 update clients (denoted as class “Write” in the figure) and a varying number of read-only clients (denoted as class “Read”). Each update client writes between 8
and 12 records, so the aggregate footprint of the update clients is roughly 100 records. The database contains 3000 records. Thus, update clients collectively touch only 3% of the database at any given time. We vary along the horizontal axis the number of read-only clients and measure the resulting abort rate (note the logarithmic vertical axis). Each read-only client reads between 100 and 200 records (5% of the database, on average) before committing. We disable safe snapshots for both SSI and SSN in this experiment. This workload exhibits extreme contention under 2PL, with reader and writer abort rates both quickly approaching 100% as additional queries overload the system. RC+SSN and RCL+SSN also suffer high abort rates ranging from 53–55% for readers across all experiments with readers, because the (already-long) query suffers additional delays due to W-R conflicts that drastically increase the likelihood of a non-repeatable read that will be aborted by SSN. In contrast, SI-based models avoid non-repeatable reads, and so achieve completion rates that suggest low contention: SSI and SI+SSN achieve better than 97% completion rates for updates and—thanks to its read-only optimization—99.9% completion rates for readers.

### 4.7.6 Writer Abort Rate due to Safe Snapshots

Finally, we examine the performance impact of our safe snapshot mechanism. Safe snapshots forcibly aborts writers that would invalidate a snapshot, so we would expect higher abort rates in return for reduced latency vs. the passive safe snapshot described in prior work [141]. Figure 4.9 examines this trade-off, varying the frequency of safe snapshots along the horizontal axis and plotting the resulting abort rate suffered by two 30-client update workloads: the class “S” workload touches roughly 10 records...
that touch roughly 10 and 40 records per transaction, respectively.

per transaction (in a 1000-record database), while the class “L” workload touches 40 (in a 4000-record database). Both workloads have a 3:1 read/write ratio, and scaling the database size with transactional footprint produces similar contention levels in both. We compare abort rates of SSI, SI+SSN and RC+SSN, for each of two footprint sizes, differentiating between aborts due to safe snapshot conflicts vs. other causes. Both horizontal and vertical axes are log-scale.

Even though the workload is rather contentious (aggregate transactional footprint size is more than 30% of the database), in most cases abort rates are relatively low, 10% or less. The fraction of aborts due to safe snapshot conflicts drops exponentially as snapshots are taken less frequently. For both transaction sizes, the snapshot kill rate drops to below 1% once the delay between snapshots matches or exceeds the expected update transaction latency (note the 4× difference in snapshot interval, corresponding to the 4× difference in footprint size). Given that most read-only queries are far larger than any update transaction (the latter tend to finish in a few ms at most), fairly infrequent snapshots (every 10ms or so) will have little or no impact on writer abort rates or reader latency.

4.8 SSN in Action

We have incorporated SSN in ERMIA [89] to provide robust CC for heterogeneous workloads. ERMIA prevents phantoms at low cost using its index (Masstree [116]). The SSI implementation in ERMIA follows the parallel commit paradigm described in Section 4.4.2. In this section, we focus on evaluating the following:

- Performance of SSN in comparison to other CC schemes under traditional OLTP workloads (Section 4.8.3);
- Impact of the optimizations for read-mostly transactions on heterogeneous workloads (Section 4.8.4);
- Effectiveness SSN’s safe retry property and SSN’s accuracy under high contention (Section 4.8.5).
4.8.1 Benchmarks

We evaluate SSN and compare its performance with SI, SSI, and optimistic CC (OCC) [93] using variants of TPC-C and TPC-E. We first use TPC-C to explore how SSN performs for traditional OLTP workloads with low contention. We also compare different CC schemes using TPC-CC, a more contentious variant of TPC-C [89]. Finally, TPC-EH, a heterogeneous OLTP workload (detailed in [89]) that features long, read-mostly transactions is used to evaluate the effectiveness of SSN’s read optimization. Details about the stock TPC-C and TPC-E benchmarks are already given in Chapter 2. Here we describe their extensions used in our experiments, TPC-CC and TPC-EH.

TPC-CC

As we have discussed above, the stock TPC-C benchmark exhibits low contention. To evaluate SSN under high contention, we use TPC-CC, a variant of TPC-C implemented in ERMIA that uses a random warehouse for each transaction [89]. Instead of assigning each thread a home warehouse, a thread chooses a warehouse randomly as its home warehouse upon starting a transaction. The percentage of remote transactions for Payment and New-Order remain the same as in TPC-C.

TPC-EH

Compared to TPC-C, TPC-E [168] is a more recent OLTP benchmark that features more sophisticated and realistic tasks that are performed by brokerage firms. Although TPC-E models modern OLTP workloads more realistically, it lacks the support for emerging heterogeneous workloads, where the execution of long and read-mostly transactions are of paramount importance. TPC-EH [89] fills this gap by introducing an additional read-mostly transaction—Asset-Eval—to TPC-E, and extending the schema with an Asset-History table. Asset-Eval aggregates assets for a set of customers and inserts the results to Asset-History. For each customer account, Asset-Eval computes the total asset by joining the Holding-Summary and Last-Trade tables. As a result, Asset-Eval will contend mostly with the Market-Feed and Trade-Result transactions, which modify the Last-Trade and Holding-Summary tables, respectively. In our experiments, Asset-Eval scans 20% of all the records in the Customer-Account table.

The Asset-Eval transaction takes 20% of the total transaction mix in TPC-EH. Because our goal is to evaluate CC schemes under contention, Data-Maintenance and Trade-Cleanup are omitted from our TPC-EH implementation. The revised transaction mix therefore becomes: Broker-Volume (4.9%), Customer-Position (8%), Market-Feed (1%), Market-Watch (13%), Security-Detail (14%), Trade-Lookup (8%), Trade-Order (10.1%), Trade-Result (10%), Trade-Status (9%), Trade-Update (2%) and Asset-Eval (20%).

4.8.2 Experimental Setup

We apply SSN over SI (denoted as SI+SSN) and compare it with other CC schemes, including SI and SSI in ERMIA, and also with OCC. The OCC implementation used in our experiments is Silo [169], a single-version, main-memory optimized system that uses a decentralized architecture to avoid physical contention. There have been newer systems that follow a similar philosophy to achieve even better performance, such as FOEDUS [90]. However, ERMIA shares the same benchmark code and implementation paradigm with Silo (e.g., both use threads—instead of processes in FOEDUS—as transaction workers). Therefore,
for fair comparison, we use Silo in our experiments. The version of Silo we used is augmented with the same TPC-C, TPC-CC and TPC-EH benchmarks in ERMIA.

We run the benchmarks described in Section 4.8.1 in Silo and ERMIA under various CC schemes on a quad-socket Linux server with four Intel Xeon E7-4890 v2 processors clocked at 2.8GHz (60 physical cores in total) and 3TB of main memory. Each worker thread is pinned to a physical core. We keep all the data in memory and direct log writes to /dev/null.

The performance numbers we report are averages of three consecutive 10-second runs, each starting with a freshly loaded database. Unless explicitly stated, all transactions aborted due to CC reasons (e.g., phantoms, exclusion window violations and write-write conflicts) are dropped. In production environments, these aborted transactions should be retried until they successfully commit. We only avoid retrying to evaluate the fairness among transactions under different CC schemes. User-instructed aborts (such as those found in TPC-E) are never retried. For TPC-C and TPC-CC, the number of concurrent threads is fixed to the scale factor (i.e., number of warehouses) unless otherwise stated. We use ten working days and a scale factor of 500 for TPC-EH.

### 4.8.3 Traditional OLTP Workloads

We first explore how SSN and other CC schemes perform under traditional OLTP workloads, by comparing the throughput of TPC-C and TPC-CC under various CC schemes. Figure 4.10 shows the throughput of TPC-C (left) and TPC-CC (middle) with a varying number of concurrent threads. The number of warehouses is fixed to the number of concurrent threads. Note that with such a setup, neither TPC-C nor TPC-CC generates enough conflicts to stress the CC significantly. Therefore, the purpose of this experiment is to understand how different CC schemes perform for the most common and simple workloads. We explore how they behave under more contention in Section 4.8.5. SI outperforms SI+SSN and SSI in all cases, however, it is not serializable. OCC outperforms the other schemes under TPC-C, which has low contention. With random warehouse selection in TPC-CC, the gap shrinks and OCC starts to perform similarly to SI. SSI performs slightly worse than SI+SSN. OCC only marginally outperforms SI+SSN under TPC-CC, showing the minimal overhead of SSN on top of the underlying CC scheme.

To further understand how different types of transactions perform under SSN, the vertical axis of Figure 4.10 (right) presents the relative percentage of each transaction’s commit in the TPC-CC mix, for the different CC schemes in the horizontal axis, including the transaction mix specified by the
Figure 4.11: Commit throughput of the TPC-EH benchmark for the full mix (left) and the AssetEval transaction.

Figure 4.12: Throughput breakdown of the TPC-EH benchmark under 60 threads.

TPC-C specification [167] (“Ideal”) for comparison. The experiment was conducted with 60 threads and aborted transactions are dropped to show any bias a CC scheme might have toward certain types of transactions. Among all the schemes we evaluated, OCC has shown a bias toward the write-intensive Payment transaction, but the other multi-version schemes have shown similar profiles to Ideal. While this is expected as OCC is known to favor write-intensive transactions, we emphasize that SI+SSN provides fair scheduling and has kept a low abort rate, without deviating much from the workload specification. SSN does not aggravate the underlying CC’s bias. We further explore the behaviors of different CC schemes under high contention in Section 4.8.5.

4.8.4 Read-Mostly Transactions

We evaluate the performance of read-mostly transactions using TPC-EH. As shown by Figure 4.11 (left), OCC keeps up with the other multi-version schemes until 30 threads. With more threads, OCC’s performance drops sharply, achieving less than 20% of SI’s throughput at 60 threads. Because of its optimistic and single-version nature, OCC does not allow any back edges in the dependency graph. Long reads can be easily invalidated by concurrent, conflicting writers, leading to massive aborts of read-mostly transactions. We plot the throughput of the Asset-Eval transaction in Figure 4.11 (right). In the figure, OCC showed a declining trend after 15 threads. Although the aggregate throughput kept increasing from 15 to 30 threads as shown in Figure 4.11 (left), the corresponding throughput numbers for Asset-Eval transactions in Figure 4.11 (right) do not show a similar trend. In other words, OCC processed more
other transactions, and adding more workers does not help in committing more Asset-Eval transactions. Most of them are aborted by concurrent updaters to the same scanned region. The throughput breakdown shown in Figure 4.12 aligns with this observation: OCC commits many fewer Asset-Eval transactions than the other schemes. SI+SSN and SSI slightly deviates from Ideal by committing ∼1.76% more and ∼0.8% fewer of Asset-Eval transactions, respectively. Unlike single-version OCC, multi-versioning allows SI-based schemes to accept all (SI) or some (deemed not harmful by SSI or SI+SSN) back edges in the dependency graph, thus allowing more valid schedules.

As shown by Figure 4.11 (left), both SI+SSN and SSI scale well under TPC-EH full mix, but with a widening gap between them and SI as the number of worker threads increases. A similar trend is found for the Asset-Eval transaction in Figure 4.11 (right). Compared to SI, SI+SSN and SSI have to maintain a read set in each transaction for validation at pre-commit. With more concurrent threads, the tracking and checking of read sets imposes higher overhead. Specifically, as shown by Algorithm 3, SSN has to iterate over the whole read set during pre-commit (SSI does so, too), which is a major source of last level cache (LLC) misses. Our profiling results for a 20-second run of TPC-EH under 30 threads show that although SI+SSN’s parallel commit procedure only takes 12% of the total CPU cycles, the function alone incurs 36.25% and 16.23% of LLC load and store misses, respectively. In this experiment, we have added a new variant (SI+SSN-R) that employs the read-mostly optimizations (Section 4.5.3). SI+SSN-R skips tracking the majority of reads, thus avoiding most LLC misses during pre-commit. SI+SSN-R achieves ∼136% and ∼97% better performance compared to vanilla SSN, for the overall and Asset-Eval performance of TPC-EH, respectively.

When running at 60 threads, SI+SSN-R could even outperform SI in terms of total commit rate as shown in Figure 4.11. Table 4.3 lists the commit rates of individual TPC-EH transactions under different CC schemes running at 60 threads. Compared to SI, SI+SSN-R commits fewer heavy-weighted Asset-Eval and Market-Feed transactions, leaving more resources available for other transactions. We note that it is critical to set an appropriate threshold for SI+SSN-R, which governs whether a version is tracked in the read set. In the case of TPC-EH, our experiments show that using a low threshold tends to kill more Market-Feed transactions because of conflicts in the Last-Trade table (reads from Asset-Eval, updates from Market-Feed), leading to even higher commit rates for other transactions. In general, under SI+SSN-R an updater that overwrote a stale version needs to adjust the reader’s sstamp, and the longer

<table>
<thead>
<tr>
<th>Transaction</th>
<th>SI</th>
<th>SSI</th>
<th>SI+SSN</th>
<th>SI+SSN-R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asset-Eval</td>
<td>2232.50</td>
<td>907.60</td>
<td>970.67</td>
<td>1915.35</td>
</tr>
<tr>
<td>Broker-Volume</td>
<td>532.83</td>
<td>222.27</td>
<td>234.88</td>
<td>572.15</td>
</tr>
<tr>
<td>Customer-Position</td>
<td>899.02</td>
<td>376.98</td>
<td>399.32</td>
<td>971.44</td>
</tr>
<tr>
<td>Market-Feed</td>
<td>49.10</td>
<td>16.68</td>
<td>16.58</td>
<td>54.92</td>
</tr>
<tr>
<td>Market-Watch</td>
<td>1464.58</td>
<td>603.80</td>
<td>643.42</td>
<td>1588.25</td>
</tr>
<tr>
<td>Security-Detail</td>
<td>1552.43</td>
<td>661.19</td>
<td>702.25</td>
<td>1687.43</td>
</tr>
<tr>
<td>Trade-Lookup</td>
<td>885.77</td>
<td>359.61</td>
<td>381.04</td>
<td>959.91</td>
</tr>
<tr>
<td>Trade-Order</td>
<td>783.42</td>
<td>333.41</td>
<td>351.53</td>
<td>851.80</td>
</tr>
<tr>
<td>Trade-Result</td>
<td>563.76</td>
<td>211.33</td>
<td>220.02</td>
<td>607.78</td>
</tr>
<tr>
<td>Trade-Status</td>
<td>984.51</td>
<td>405.26</td>
<td>429.60</td>
<td>1068.06</td>
</tr>
<tr>
<td>Trade-Update</td>
<td>229.45</td>
<td>93.63</td>
<td>101.59</td>
<td>248.44</td>
</tr>
<tr>
<td><strong>Total TPS</strong></td>
<td><strong>10177.37</strong></td>
<td><strong>4191.75</strong></td>
<td><strong>4450.90</strong></td>
<td><strong>10525.52</strong></td>
</tr>
</tbody>
</table>

Table 4.3: Throughput (TPS) of individual TPC-EH transactions under multi-version CC schemes with 60 threads.
the reader is, the easier it is for an updater to catch the reader “alive” during the latter’s pre-commit phase. As a result, longer readers will have a higher chance of being issued a lower $s\text{stamp}$, making it easier to violate an exclusion window. For the experimental results reported in this chapter, we set the threshold to $0x\text{FFFFFF}$. This choice provides a balance between overall commit rate and fairness among individual transactions. With a threshold of $0x\text{FFFFFF}$, versions that are not updated during the past period in which $\sim1$MB of data are written in the database, are considered stale and consequently not tracked in the read set. For SI+SSN-R to provide both high aggregate throughput and fair scheduling, one must adjust the threshold depending on the workload.\footnote{\textit{Our current implementation accepts a user-defined, workload-specific threshold. Self-tuning it as workload changes is future work.}} In summary, as shown in Table 4.3, SI+SSN roughly follows SI’s breakdown but provides lower commit rates, while SI+SSN-R sacrifices a little fairness toward readers but also maintains high aggregate throughput (similar to SI’s) with a proper threshold.

### 4.8.5 Safe Retry and High-Contention Workloads

This section evaluates SSN’s safe retry property. We set both ERMIA and Silo to retry aborted transactions until they commit successfully (therefore the throughput breakdown—although not shown—will strictly follow the benchmark specification). Figure 4.13 shows the commit (left) and abort (right) rates of TPC-CC with a varying number of concurrent worker threads. Despite the extra effort needed to retry transactions, SSN matches the performance of SSI and performs similarly to the case where we drop aborted transactions (right side of Figure 4.10). OCC’s commit rate collapses as core count increases, especially after 15 threads. Our profiling results show that Silo spent the vast majority of CPU cycles (higher than 60\%) on retrying index insertions (mostly for New-Order), minimizing the available cycles for other transactions and getting little useful work done. Therefore, as shown by Figure 4.13 (right), OCC also kept a much lower abort rate than SSI/SSN did. Compared to Silo, ERMIA uses indirection arrays for multi-versioning \cite{89,103,151}. This design makes tuple insertion less reliant on index performance: Silo needs to first retry index insertion before finalizing the tuple write at commit time, putting tremendous pressure on the index, while ERMIA only needs to insert to the index after successfully appending an entry in the table’s indirection array, amortizing much contention on the index.

Although both SSI and SI+SSN commit and abort similar numbers of transactions, they do so because of different reasons: SSN exhibits higher accuracy and lower abort rate due to certification failures, i.e.,
it aborts fewer transactions due to serializability check than SSI does. Figure 4.14 shows this effect. As we increase the number of concurrent threads, SSI tends to abort more transactions due to certification failures, while SI+SSN tends to abort more due to other reasons including write-write conflicts and unsuccessful lookups. We also note that a random backoff strategy during retries can largely mitigate this problem in Silo. However, this will introduce a large variance in transaction latency, whereas SSN’s safe retry property can keep good performance and maintain stable latency.

In Figure 4.15 we further stress the CC schemes under high contention: the number of warehouses is fixed to 15 and we vary the number of concurrent threads. Like previous experiments, each thread chooses a random warehouse and retries a transaction until it is successfully committed. At 15 threads, all CC schemes exhibit exactly the same performance numbers as shown by Figure 4.13. As we increase the number of concurrent threads (i.e., more contention), as shown by Figure 4.15 (left), multi-version schemes can still benefit from the increased parallelism. OCC, however, almost completely collapses as it keeps failing index insertions. Figure 4.15 (right) exhibits a similar but more significant effect as in Figure 4.14, showing SSN’s accuracy and robustness under high contention. We also observed a similar trend under TPC-CC without retrying aborted transactions under moderate levels of contention (equal numbers of warehouses and concurrent threads). For example, when running at 60 threads, SSN exhibits overall $\sim60\%$ lower aborts due to serialization failures when compared to SSI.

Figure 4.16 shows the commit ratio for each transaction and CC scheme running at 60 threads with 15 warehouses. The vertical axis represents the percentage of a transaction committed out of all its
retries. Similar to the previous experiment, OCC has exhibited a bias toward the write-intensive Payment
transaction with a higher than 80% commit ratio, i.e., on average at most one in five transactions needs
to retry. But the percentage of committed NewOrder transactions is only 2.36%, due to its repetitive
failure of index insertion. For NewOrder, SI+SSN achieved the highest commit ratio, although SSI and
SI+SSN have similar aggregate throughput. Both SI and OCC have 100% commit ratio for the two
read-only transactions (StockLevel and OrderStatus); the former does not track and validate reads, while
the latter uses a read-only snapshot for read-only transactions. Finally, the commit ratios under SSI
and SI+SSN for StockLevel show the accuracy of SSN which achieves a 17% higher commit ratio when
compared to SSI.

4.9 Summary and Discussion

We have presented the serial safety net (SSN), a cheap certifier that can overlay a variety of concurrency
control schemes and make them serializable. We prove the correctness of SSN, we show how SSN can be
efficiently implemented for multi-version systems, and we have evaluated SSN in both simulation and
ERMIA, a recent main-memory multi-version database system designed for modern hardware.

SSN is robust against a variety of workloads, ranging from traditional OLTP applications to emerging
heterogeneous workloads. In particular, we have proposed specific optimizations for these heterogeneous
workloads where long, read-mostly transactions are of paramount importance. With the help of a carefully
designed lock-free, parallel commit protocol, SSN adds minimal overhead to the underlying CC scheme in
a multi-version system, in terms of read/write tracking and commit-time validation. Experiments using
TPC-C and TPC-E based benchmarks show that SSN is superior to prior state-of-the-art, in being more
accurate (fewer aborts and higher commit ratio), more general (not requiring SI), more robust against
retries and more friendly to emerging heterogeneous workloads that feature read-mostly transactions.

* * *

On modern high-end hardware with a few sockets, SSN performs well under a wide range of workloads.
Future machines will come with even larger scales and higher parallelism. However, SSN still relies
on a centralized timestamp for global ordering, which strongly limits its scalability on future servers.
Some of today’s high-end servers already exhibit properties of future hardware (e.g., more severe NUMA
effects), which requires a departure from CC methods that employ any form of centralized machinery
and unnecessary interthread communication. This requirement has lead to the wide adoption of OCC,
which is well-suited on large-scale machines. However, it is especially vulnerable to high-contention. The
next chapter solves this problem by proposing mostly-optimistic concurrency control (MOCC).
Chapter 5

Mostly-Optimistic Concurrency Control

Future servers will be equipped with thousands of CPU cores and deep memory hierarchies. Traditional concurrency control (CC) schemes—both optimistic and pessimistic—slow down by orders of magnitude in such environments for highly contended workloads. Optimistic CC (OCC) scales the best for workloads with few conflicts, but suffers from clobbered reads for high conflict workloads. Although pessimistic locking can protect reads, it floods cache-coherence backbones in deep memory hierarchies and can also cause numerous deadlock aborts.

We propose a new CC scheme, mostly-optimistic concurrency control (MOCC), to address these problems. MOCC achieves orders of magnitude higher performance for dynamic workloads on modern servers. The key objective of MOCC is to avoid clobbered reads for high conflict workloads, without any centralized mechanisms or heavyweight interthread communication. To satisfy such needs, we devise a native, cancellable reader-writer spinlock and a serializable protocol that can acquire, release and re-acquire locks in any order without expensive interthread communication. For low conflict workloads, MOCC maintains OCC’s high performance without taking read locks.

Our experiments with high conflict YCSB workloads on a 288-core server reveal that MOCC performs 8× and 23× faster than OCC and pessimistic locking, respectively. It achieves 17 million TPS for TPC-C and more than 110 million TPS for YCSB without conflicts, 170× faster than pessimistic methods.¹

5.1 Introduction

Core counts in modern database servers range into the hundreds today [64,154] and are expected to grow into the range of thousands soon [65]. Despite such advances, emerging software systems are struggling to utilize such modern hardware to generate, analyze, and store an exponentially growing amount of data. For example, GraphLab [110] parallelizes a graph query by internally splitting up the work into billions of ACID transactions that often contain read-write conflicts between cores. Other examples of growing concurrency demands include high-frequency trading and smart grid management.

Existing databases, however, cannot scale up to thousands of cores, especially when the workload is highly contended. Most database engines today employ either pessimistic two phase locking (2PL) or

¹ This chapter is based on materials that appeared in VLDB 2017 [179].
optimistic concurrency control (OCC) [93] to guarantee serializability. As we shall see, critical scalability bottlenecks slow down both 2PL and OCC by orders of magnitude in modern servers.

OCC has low overhead and scales up nicely on manycore servers for low-conflict/contention workloads. When the workload contains frequent read-write conflicts, however, OCC can exhibit high abort rates that strongly limit throughput. We have seen this effect in Section 4.8. 2PL exhibits two scalability issues. First, pessimistic read locks severely limit scalability due to expensive cache-coherence traffic on deep memory hierarchies. Second, somewhat ironically, 2PL does not necessarily reduce aborts. Especially when record accesses are highly contended and in a random order, even 2PL causes massive aborts due to deadlocks. Moreover, existing 2PL architectures necessitate unscaleable interthread communication.

We propose mostly-optimistic concurrency control (MOCC), a new concurrency control scheme that addresses these problems and scales up to thousands of cores. MOCC dramatically reduces the abort ratio and guarantees robust progress even when the workload has extremely frequent read-write conflicts. Yet, MOCC has extremely low overhead, performing orders of magnitude faster than 2PL and as fast as the state-of-the-art OCC even in its best case, low conflict workloads. Finally, MOCC dynamically and autonomously optimizes itself to retain its high performance in the most challenging yet common case, in which each transaction involves multiple data sets of opposite nature: records that have frequent read-write conflicts and records that are read-mostly.

MOCC is based on modern OCC that is highly decentralized. There are two key reasons why we design MOCC based on OCC. First, its decentralized architecture scales far better on modern servers when the workload does not have too many conflicts. It is adopted by state-of-the-art main-memory optimized systems, such as FOEDUS [90] and Silo [169]. MOCC behaves the same as FOEDUS for such workloads and retains its performance. Second, we found that OCC’s commit protocol fits well with the non-2PL protocol proposed here, which must acquire and release locks in arbitrary orders without violating serializability.

Like 2PL, MOCC incorporates pessimistic read locks to prevent writers from clobbering readers. Unlike 2PL, however, an MOCC transaction might release locks before commit, then re-acquire them in a different order during pre-commit. Doing so allows MOCC to effectively avoid aborts and deadlocks, without relying on any unscaleable component such as a centralized lock manager. Section 5.3 describes the MOCC protocol in detail, including how MOCC dynamically and selectively adds pessimistic locking onto OCC and how MOCC avoids deadlocks.

Another key technique to enable MOCC is the MOCC Queuing Lock (MQL), a reader-writer queue-based lock that allows parallel, asynchronous locking and cancelling. Compared to the lock algorithm in traditional databases, MQL scales far better on deep memory hierarchies and also supports modes that MOCC exploits: reader/writer and unconditional/try/asynchronous locking. Section 5.4 describes MQL and how MOCC exploits it.

The experiments in Section 5.5 show that MOCC achieves orders of magnitude higher performance than existing approaches on a modern 288-core machine serving high contention workloads.

5.2 Key Challenges and Principles

Scalability tradeoffs surrounding both optimistic and pessimistic concurrency control have been studied extensively. This section highlights the key challenges that we face and the principles that differentiate our work from prior research.
Chapter 5. Mostly-Optimistic Concurrency Control

5.2.1 Impact of Massive Parallelism

Compared with today’s hardware, emerging servers will contain far more CPU cores and a deeper, more complex memory hierarchy to interconnect them. Figures 5.1 and 5.2 illustrate the shocking consequences for performance.

This experiment uses FOEDUS [90], a manycore optimized database that uses modern decentralized OCC. We added a new 2PL variant to it and ran highly contended (50 records) YCSB benchmarks. We tested on various machines, ranging from a single-socket dual-core laptop to a modern server with 16 sockets and 288 cores. We present the details of this experiment in Section 5.5 and summarize the main observations here.

Extremely High Contention

Figure 5.1 shows a read-only workload where 2PL performs more slowly than OCC due to its overhead to take read locks, which is expected. Nonetheless, it is surprising to observe that the read lock overhead is not merely a nuisance, but an abyss [187], slowing down the seemingly easy read-only workload by \(170\times\) in a modern server. The slowdown quickly grows with the number of cores and the depth of the memory hierarchy due to physical lock contention even though all of the locks are logically compatible.
Prior literature has studied physical lock contention \cite{75,91}, but none has observed this much effect (e.g., Johnson et al. \cite{75} report up to 15% slowdown on 64 threads) or designed a system for such a high level of contention.

We also observed several other scalability issues in a far larger degree than prior studies, such as the ever growing cost of remote NUMA access and atomic operations. To the best of our knowledge, the only empirical performance studies at similar scales, except simulation-based \cite{187}, were found outside of the context of databases, such as those in the locking community \cite{43,173}.

At this scale, we found that strict 2PL is not a viable choice because it causes severe cacheline ping-pong in the already moribund cache-coherence backbone.

**Extremely High Conflict**

OCC scales well under low conflict, but it suffers a different drawback: high abort ratio under high conflict. Figure 5.2 varies the number of writes from zero (read-only) to 10 (read-modify-write only). The extremely high concurrency in hardware brings extremely high read-write conflicts that severely degrade performance. With just one write per transaction, more than 80% of transactions abort in OCC’s commit phase due to clobbered reads. With 10 writes, as many as 98% of transactions abort. Many transactional applications have some hot spot that receives high conflict accesses. Such a high abort ratio can completely stifle the performance of OCC. Furthermore, somewhat ironically, even pessimistic protocols have a high abort ratio due to deadlocks.

**5.2.2 Issues in Existing Databases**

Emerging server hardware demands a departure from previous database architectures that contain the following bottlenecks.

**Page Latches**

Most existing databases use page latches, or a lock to protect a physical page rather than logical records. Even some databases that use OCC for individual records take read latches for pages. Furthermore, read latches must be coupled to guarantee consistent page traversals \cite{114}. Although previous work at smaller scales did not find physical contention on page latches as a central bottleneck, it severely stifles performance on thousands of cores.

**Reads Become Writes**

Some prior systems (e.g., the BwTree \cite{103} in Hekaton \cite{42}) maintain a readers count to deter writers from clobbering reads. Although such schemes ameliorate aborts in OCC, a read operation must write to a contended location, limiting scalability and performance.

**Expensive Interthread Communication**

Traditional architectures, whether OCC, 2PL, or multi-versioned, require interthread communication that limits scalability in modern servers. One example is the tracking of anti-dependencies to guarantee serializability on top of snapshot isolation, where reads should leave notes in shared locations for other
threads to detect possible serializability violations [25]. Another instance is interthread communication to detect and resolve deadlocks, which we discuss next.

**Frequent Deadlocks and Aborts**

Aborts are the main scalability issue in OCC for highly contended workloads. However, even 2PL significantly slows down for these workloads due to aborts caused by deadlocks [145]. Using strict discipline, it is possible for a developer to design and implement a simple application where all record accesses strictly follow the same order to prevent deadlocks. However, this is impractical for database applications because of the complexity and hidden object accesses, such as secondary indexes and foreign key constraints. Hence, the high concurrency on modern servers often results in a large number of deadlocks and aborts. To make things even worse, due to the repetitive nature of transactional workloads, deadlocks will keep occurring in the same contended place, continually aggravating the issues described above.

**5.2.3 Key Principles of MOCC**

The above observations guide our key principles:

- MOCC must be based on an architecture without page latches for reads, similar to pure OCC.

- To scale better, the only mechanism that is needed for the majority of reads must be OCC without any writes to contended memory locations.

- On top of OCC, MOCC must selectively acquire read locks on records that would cause an abort without locks.

- MOCC must avoid deadlocks without any unscalable interthread coordination.

**5.3 Mostly Optimistic Concurrency Control**

Figure 5.3 gives a high level overview of MOCC. MOCC is based on a modern decentralized OCC without page latches for reads. Section 5.3.1 recaps the relevant aspects of OCC.

MOCC employs pessimistic read locks to overcome the drawback of OCC: aborts during read verification. However, if MOCC issues read locks for all reads, its scalability will degenerate to that of 2PL. MOCC thus maintains temperatures to selectively issue read locks only to reads that will likely cause read-write conflicts (Section 5.3.2).

Although it sounds trivial to acquire read locks, doing so naively could revive all the bottlenecks and complexities that OCC avoided to achieve high scalability. The crux of implementing MOCC lies in the locking and commit protocol to avoid deadlocks caused by the additional locks taken before the commit phase. Section 5.3.3 covers the details of the MOCC protocol.

**5.3.1 Keeping Decentralized OCC’s Benefits**

MOCC inherits several ideas from decentralized OCC-based systems, most notably the following.
No Page Latching for Reads

Same as FOEDUS [90], MOCC avoids acquiring page latches for reads. For example, FOEDUS employs a variant of Masstree [116] that is not only cache-friendly, but also has a performance advantage on manycore servers because it does not require page latches for read operations. Masstree employs RCU (read-copy-update) to create a new version of a page and atomically switches a pointer to it. Hence, a read operation does not need any page latch to safely access a page. In MOCC, other storage representations (e.g., lock-free hash tables) do not require page latches for reads, either.

Apply-after-Commit

One key principle in modern OCC is to decentralize its transactional processing as much as possible for higher scalability. Each worker thread maintains its own log buffer and merely appends its uncommitted write operations to the log buffer. The thread does not apply the writes to data pages until the transaction is guaranteed to commit. The log buffers are completely thread private, hence threads synchronize with each other only when there is a potential conflict in their data accesses or an epoch-based group commit happens. Another important aspect of this design is the redo-only protocol that has no uncommitted data in pages even when a transaction aborts. This enables the non-2PL yet serializable lock/unlock protocol MOCC employs.

Deadlock-Free Verification Protocol

All OCC variants, such as Silo and FOEDUS, verify reads at transaction commit time to guarantee serializability. Each transaction remembers the state of records as of reading and compares it with the current state at the commit procedure. If any of the reads observes a different state, the transaction aborts and either retries or returns an error to the client.
An OCC transaction finalizes the order in which it is serialized by locking all records it will modify (its write set) before it verifies the reads and determines the serialization order from its read/write set.

Most importantly, this protocol is completely deadlock-free. The locking happens during the commit phase when the transaction already knows all of its read/write set. As a result, we can lock the write set in any globally consistent order (e.g., by tuple virtual addresses), thus guaranteeing deadlock freedom and serializability without any interthread communication or relying on a centralized lock manager. Assuming this property, we can unconditionally take each lock, meaning the transaction indefinitely waits for lock acquisition.

The Missing Read Locks

The above property perfectly satisfies our goal if we do not take any read locks. However, a transaction must take read locks to prevent read-clobber aborts, and it must take them as of the read operations to be protected, at which point the full footprint is unknown. This breaks the deadlock-free property, and the existing solution is to revive heavyweight, unscalable interthread communication, falling back to the scalability of 2PL. This is exactly why adding read locks to OCC, albeit apparently straightforward, requires non-trivial efforts and is the crux of MOCC.

5.3.2 Temperature Statistics

MOCC maintains temperature statistics, illustrated at the top of Figure 5.3, to predict how likely a read on a record will be clobbered by concurrent transactions. MOCC takes read locks on records whose temperature is above some threshold. Such records are likely to be updated by concurrent transactions, causing aborts. The temperature statistics track the number of aborts due to verification failures. To reduce space overhead, we maintain the statistics at page level, instead of in each record. Verification failures will increase the temperature of affected pages containing the clobbered records.

Since all the records belonging to the same page share the same temperature statistic, frequent writes to it might cause physical contention, defeating our purpose. To solve this problem, we maintain page temperature (i.e., number of aborts) using a variant of approximate counters [127]. Each page $p$ maintains a temperature $\text{temp}_p$, which probabilistically implies that there were about $2^{\text{temp}_p}$ aborts in the page. Whenever a transaction is aborted due to a verification failure on a record in $p$, it checks the current value of $\text{temp}_p$ and increments it with probability $2^{-\text{temp}_p}$. Upon observing an abort, a page that has had frequent aborts (hot page) is likely to have a higher value in $\text{temp}_p$, significantly lowering the probability of cacheline invalidation due to frequent writes to a centralized location.

5.3.3 MOCC Protocols

Guided by the temperature statistics, MOCC takes pessimistic locks on hot records as described in Algorithm 4. Lines 9–10 and 15–16 show how we check the temperature statistics and trigger pessimistic locking. The OCC protocol is no longer deadlock-free as soon as we take locks during transaction execution in addition to taking locks at commit time. To avoid deadlocks and repeated aborts due to pessimistic locking, MOCC follows a non-2PL approach, described next.
Chapter 5. Mostly-Optimistic Concurrency Control

Algorithm 4 MOCC protocols: read, write, lock, and RLL construction.

```python
class MoccTransaction:
    const H # Temperature Threshold
    R := {} # Read Set
    W := {} # Write Set
    RLL := {} # Retrospective Lock List
    CLL := {} # Current Lock List

    def read(t: Record):
        if temp(t) >= H or t in RLL:
            lock(t, max(preferred mode in RLL, R-mode))
        R.add(t, t.TID)

    def read_write(t: Record):
        if temp(t) >= H or t in RLL:
            lock(t, W-mode)
        R.add(t, t.TID)
    # Blind-write, same as in OCC

    def lock(t: Record, m: Mode):
        if CLL already has t in mode m or stronger:
            return
        violations := {l ∈ CLL, l.mode ≠ null, l ≥ t}
        if too many violations:
            alternative_lock(t, m)
            return or abort
        elif violations not empty:
            # Not in canonical mode. Restore.
            CLL.unlock({violations})
        # Unconditional lock in canonical mode.
        CLL.unconditional_lock({l, m ∈ RLL, l < t})
        CLL.unconditional_lock(t, m)

    def construct_rll(): # Invoked on abort
        RLL := {}
        for w in W:
            RLL.add(w, W-mode)
        for r in R:
            if r not in RLL:
                if temp(r) >= H or r failed verification:
                    RLL.add(r, R-mode)
        RLL.sort()
```

Canonical Mode

Central to the MOCC protocol is the concept of canonical lock acquisition. The idea generalizes the consistent sorting of write sets in FOEDUS and Silo [169], such as the virtual address order and lock ID order. Let $l_m < l_n$ mean that the lock $l_m$ is ordered before the lock $l_n$ in some universally consistent order. Let $CLL$ be the list of locks a transaction $T$ has taken so far. Suppose $T$ now tries to acquire a new set of locks $NL$, then $T$ is said to be in canonical mode if and only if $l_c < l_n : \forall l_n \in NL, l_c \in CLL$. 

When a transaction is in canonical mode, there is no risk of deadlock. Thus, the transaction can unconditionally take the locks just like FOEDUS or Silo does. Unconditionally taking a lock is the most desirable way of acquiring locks especially in queue-based locks as Section 5.4 describes.

When MOCC does not take any read locks, it is always in canonical mode because the only locking happens in commit phase with an empty $CLL$ and $NL$ being the write set. Read locks, however, can cause the transaction to leave canonical mode. In short, the MOCC protocol is designed to: (1) keep transactions in canonical mode as much as possible, (2) restore canonical mode when not in canonical mode, and (3) try taking locks as efficiently as possible without risking deadlocks when canonical mode is not attainable.

**Acquiring and Releasing Locks**

In traditional 2PL architectures, there is not much one can do to reach the above goals because locks must be held in *two phases*: the growing phase takes locks and the shrinking phase releases locks. If a transaction releases a lock before commit and then takes another lock, the execution could be non-serializable.

However, MOCC is based on OCC, which gives far more flexibility in this regard. MOCC verifies reads at commit, hence serializability is guaranteed no matter whether it holds a read lock or not. MOCC, like the original OCC protocol, determines the serialization order and verifies/applies the read/write set only after it takes all write locks. Hence, until MOCC finalizes the transaction in commit phase, MOCC can safely acquire, release, or re-acquire arbitrary locks in an arbitrary order. MOCC fully exploits this flexibility, which is one reason why MOCC is based on OCC.

For example, in line 26 of Algorithm 4, suppose the transaction has a current lock list $CLL : \{l_1, l_2, l_4\}$ and intends to take a read lock $t = l_3$. Since the transaction is already holding $l_4$, taking a lock ordered before it will leave the transaction in non-canonical mode. MOCC can restore canonical mode by releasing $l_4$ first (lines 30–32), then unconditionally take $l_3$ (lines 35–36). We do not re-take any such released lock unless it is explicitly requested later. This does not violate serializability because MOCC verifies reads at commit time (same as in OCC). There is only a slightly higher risk of verification failure, but correctness is never compromised. The same technique applies to write locks during the commit phase, too. Whenever we take write locks in non-canonical mode, for example when the record is moved to a different page and the sort order of its write set we initially assumed has changed, we can safely release some of the locks to restore canonical mode.

The restoration of canonical mode is a powerful tool in MOCC to avoid deadlocks without any heavyweight modules or interthread communication. However, releasing too many locks is also costly. In the above example, if $CLL$ also contains $l_5, l_6, \ldots, l_{1000}$, the cost to release and re-acquire a large number of locks is fairly high. In such a case (lines 27–28), MOCC tries to take $l_3$ in non-canonical mode without releasing the affected locks.

Allowing transactions to violate the non-canonical mode requires a cancellable lock interface discussed in Section 5.4. When the lock cannot be granted immediately, the acquiring thread can choose to abort and retry because long lock acquire delays might indicate a deadlock (line 29 of Algorithm 4). If the lock is requested in read mode (thus not a mandatory acquisition), then one can ignore the lock request and move on. For write locks, however, a long delay might demand an immediate abort because acquiring the lock is mandatory for determining serialization order.
Retrospective Lock List

Whenever a transaction in MOCC aborts, due to either the aforementioned conservative aborts in non-canonical mode due to a long delay in acquiring a write lock or simply a read verification failure, MOCC instantiates a retrospective lock list (RLL) for the transaction. An RLL is a sorted list of locks with their preferred lock modes that will likely be required when the thread retries the aborted transaction. The RLL keeps the retried transaction in canonical mode for most cases.

Constructing RLL. Function `construct_rll()` in Algorithm 4 shows how MOCC constructs RLL from the read/write set of the aborted transaction and temperature statistics. All records in the write set are added to RLL in write mode (lines 40–41). Records in the read set that caused verification failures or in hot pages are added to RLL in read mode (lines 42–45). When a record is both in the read set and write set, which happens often, RLL maintains a single entry for the record in write mode. At the end of the construction, MOCC sorts entries in RLL for the next run.

Using RLL. Function `read()/read_write()` in Algorithm 4 is invoked whenever a transaction accesses a record. The transaction checks the temperature statistics and queries the RLL. When either of them implies that a pessimistic lock on the record is beneficial, we immediately take all locks in RLL ordered before the requested lock. The transaction also respects the recommended lock mode in the RLL. In other words, the preferred lock mode in RLL overrides the requested lock mode. It takes a write lock instead of read lock when the requested lock itself exists in RLL. Even if the transaction does not see a high temperature of the page, it takes a read lock on the record when RLL advises so, which means the previous run was aborted due to a conflict on the record despite low temperature.

RLL often keeps the retried transaction in canonical mode, making the transaction completely free of deadlocks and aborts because all the contended reads are protected with read locks. As we verify in Section 5.5, RLL achieves significantly more robust progress, even for the most contended transactions, than either OCC or 2PL.

However, there are a few cases where even RLL might not prevent aborts. First, the user code might have non-determinism that changes its behavior in another run and accesses a different set of records. Such applications exist, but removing non-determinism is relatively easy for most applications and will provide additional opportunities to improve performance [147]. Second, even when the user code is logically deterministic, it might access different physical records due to various structural modification operations (SMOs), such as page splits. When these cases occur, MOCC falls back to the behavior in non-canonical mode regarding the particular record that violates canonical mode. It might cause another abort, but it is unlikely for the same thread to keep observing an SMO for the same page.

Commit Protocol

An MOCC transaction invokes `commit()` and possibly `abort()` functions in Algorithm 5 when it commits. Like OCC, MOCC verifies reads after taking write locks. Moreover, we verify all reads even if they are protected by read locks. This is necessary because even if the record is already being protected by a read lock, it might not be acquired at the point this tuple was first read during the transaction because MOCC eagerly releases and re-acquires locks. However, note that once we have acquired a read lock, no other transactions can modify the record’s version number, which in turn should be cached by the CPU in most cases. This makes the verification process cheap.

When verification fails due to clobbered reads, MOCC probabilistically updates the temperature
Algorithm 5 MOCC Protocols: commit and abort.

```python
class MoccTransaction:
    const H # Temperature Threshold
    R := {} # Read Set
    W := {} # Write Set
    RLL := {} # Retrospective Lock List
    CLL := {} # Current Lock List

    def commit():
        W.sort()
        for w in W:
            lock(w, W-mode)
        for r in R:
            if r.observed_tid not equal r.tid:
                temp(r).hotter()
                abort
        # Committed
        Determine TID and publish W # Silo/FOEDUS protocol
        CLL.unlock_all()
        RLL, CLL, R, W := {}

    def abort():
        CLL.unlock_all()
        if user will retry the transaction:
            construct_rll()
        else
            RLL := {}
            CLL, R, W := {}
```

statistics of the corresponding page. MOCC then constructs an RLL from the now-complete read/write sets. MOCC uses RLL only when the previous run aborts and the client chooses to retry the same transaction.

5.3.4 Discussion

Temperature Statistics

Temperature statistics can be either per-page or per-record. Per-record statistics would be more accurate to predict verification failures, but it requires a larger footprint for the statistics. We chose per-page statistics for our current implementation because in many cases RLL can recommend read locks on individual records that caused aborts, which overrides what the (low) page temperature recommends.

The MOCC protocol is not sensitive to the threshold unless it is extremely low (e.g., 0) or high (e.g., 20) as shown in Section 5.5. Highly conflicting pages, by definition, quickly reach the threshold and trigger read locks. In most cases, we recommend setting the threshold to 10.

The counter must be either reset or decremented because the nature of the page might change over time. Our experiments in Section 5.5 occasionally reset the value of the counter to zero. The statistics will be inaccurate immediately after the reset, but the value quickly becomes large again if the page still is hot. MOCC can also use alternative approaches, such as occasionally decrementing the counter when the read lock turns out to be unnecessary by checking the state of lock queues at commit time.
Consistent Ordering

The ordering scheme can be anything so long as it is universally consistent among all threads as discussed in Tu et al. [169], and we follow the same principle. We note, however, that the order based on virtual addresses does not work in some situations. FOEDUS, like most major databases, runs multiple processes using multiple shared memory segments. The order of virtual address is not consistent among processes in such an environment. For instance, shared memory segments A and B might be ordered $A < B$ in process-1’s virtual address space while they might be ordered $B < A$ in process-2. Furthermore, even in a single process, memory-mapping APIs, such as `shmat` and `mmap`, give aliases to the same physical memory, leading to inconsistent ordering and deadlocks. We thus use a logical identifier of each lock, which consists of a shared memory segment ID and an offset of the lock object from the beginning of the memory segment.

5.4 MOCC Queuing Lock

In this section, we devise the MOCC Queuing Lock (MQL) to realize MOCC without the prohibitive overheads incurred by traditional locking. MQL is a scalable, queue-based reader-writer lock with flexible interfaces and cancellation support. This section focuses on describing:

- Why MOCC demands yet another lock (Section 5.4.1);
- Detailed MQL algorithm design and implementation (Sections 5.4.2 – 5.4.4);
- How MOCC uses MQL (Section 5.4.5).

5.4.1 The Need for Native, Queue-based Locks

Native Locks

MQL must be a native lock. Traditional databases use record locking that supports reader-writer modes and cancellation. But, they often use two levels of synchronization primitives: (1) an exclusive spinlock to protect the lock state itself, and (2) a logical lock to protect a record in various modes.

Traditional lock managers often use basic, centralized spinlocks (e.g., test-and-set and ticket locks) for (1) and sleep-wait (e.g., `pthread_mutex`) for (2). Lock managers typically employ a hash-partitioned lock table to manage locks [76]. Any access to lock state must be protected by a spinlock in the corresponding partition. A transaction must acquire the spinlock every time no matter whether it merely checks lock compatibility or inserts a new request into a lock queue. Because all accesses to the lock state are serialized, the database can easily provide various lock modes and support cancellation.

This approach works well enough in disk-based databases with a few CPU cores, but recent database engines optimized for main memory and manycore servers observed bottlenecks in two-tier locking. Thus, recent systems have started to employ low-level synchronization primitives directly as record locks. For example, FOEDUS places a spinlock in each record header. Transactions directly synchronize with each other using spinlocks. We call this approach “native locking”. MOCC demands a native lock to scale up to 1000 cores.
Queue-based Locks

With deep memory hierarchies, queue-based locks (e.g., MCS locks [119]) exhibit much better scalability than centralized locks. As Chapter 6 shows, on a 240-core server we observed that the MCS lock has two orders of magnitude better performance than centralized spinlocks. However, existing native queue-based locks have limited functionality and cannot fully replace logical locks found in traditional databases. For example, FOEDUS uses the MCS lock for records. The reason it could do so is because its commit protocol needs only exclusive and unconditional record locking. MOCC needs to issue read locks, which inherently demands reader-writer modes. MOCC also requires the cancellation (timeout) functionality to avoid deadlocks. Existing queue-based lock algorithms [82, 99, 120, 153, 173] can only partially satisfy MOCC’s requirements: they lack either the needed functionality or performance. For instance, exclusive-only locks would lower concurrency while non-cancellable locks would cause unrecoverable deadlocks. We thus develop MQL to support both cancellation and reader-writer modes.

5.4.2 MQL Algorithms

MQL combines the (fair) reader-writer [120] and timeout [153] variants of the MCS lock. However, combining these two variants is no trivial task. The key challenge is the increased number of lock states. Supporting reader-writer and cancellation features fundamentally necessitates more lock states. To achieve both, MQL maintains (1) three variables in the lock, (2) a doubly-linked list of requesters, and (3) five additional variables in each queue node (qnode) that represents a requester.

Figure 5.4 shows MQL’s lock word and qnode structure. The lock word is exactly the same as the reader-writer MCS lock’s and records the number of readers, a pointer to the writer lined up for the lock next, and a pointer to the latest requester. The requesters form a doubly-linked list and each requester, similar to requesters of the original MCS lock, carries a qnode. In each qnode, MQL employs the prev and granted fields to interact with the predecessor, and the busy, successor_type, status, next fields for interactions with the successor. In addition, we use special sentinel values on some of these fields for handshakes between threads during cancellation. Table 5.1 summarizes the possible and initial values of each field. The locks granted status is represented in multiple fields, this is needed to support reader/writer and cancellation. MQL uses a collection of lock-free protocols to carefully coordinate interactions and handshakes between requesters. Especially, the special sentinel values shown in Table 5.1 are used to support cancellation.

![Figure 5.4: MQL data structures. Requesters form a doubly-linked list of qnodes to handle cancellation.](image-url)
Table 5.1: Possible and initial values of MQL queue node fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Possible values and meaning</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Reader/Writer</td>
<td>Reader or Writer</td>
</tr>
<tr>
<td>prev</td>
<td>1. Pointer to predecessor</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>2. Acquired - lock acquired</td>
<td></td>
</tr>
<tr>
<td>granted</td>
<td>True/False - whether lock is granted</td>
<td>False</td>
</tr>
<tr>
<td>busy</td>
<td>True/False - whether lock state is changing</td>
<td>False</td>
</tr>
<tr>
<td>successor_type</td>
<td>None/Reader/Writer</td>
<td>None</td>
</tr>
<tr>
<td>status</td>
<td>1. “0” - waiting for lock</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2. “1” - lock granted</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. “2” - cancelling request</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. “3” - lock granted during cancellation</td>
<td></td>
</tr>
<tr>
<td>next</td>
<td>1. Pointer to successor</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>2. SuccessorLeaving - successor is cancelling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. SuccessorHandled - successor already handled</td>
<td></td>
</tr>
</tbody>
</table>

In the rest of this section, we first give details on how MQL handles unconditional lock requests, i.e., the requester will not cancel and will spin-wait until the lock is granted. We then extend our algorithm to support cancellation. Since the protocols for reader and writer requesters share many commonalities, we explain the algorithms for readers which are more complex in detail and present the algorithms for writers at the end of this section.

Supporting Readers and Writers

Without cancellation, MQL works similarly to the fair reader-writer MCS lock. A requester (reader or writer) S brings a qnode (shown in Figure 5.4) and joins the queue using the wait-free doorway [94]: S issues an atomic swap instruction (XCHG) to install a pointer to its qnode on lock.tail. The XCHG’s return value r will point to the predecessor P (if there is any).

Compatible requests. If S is a reader, it can enter the critical section if (1) the lock is free, or (2) P is also a reader and is holding the lock. Algorithm 6 details the steps for a reader to acquire the lock. We omit the details related to cancellation for now. Lines 3–6 show the first case in which the lock is free. Then S atomically increments lock.nreaders and toggles its granted field. S calls finish_reader_acquire to wake up its reader successor (if there is any) that came in before S executed line 2 of Algorithm 7. We discuss of finish_reader_acquire later when describing case (2) below.

Lines 11–18 show how the requester proceeds if it has a predecessor. In particular, if both the predecessor P and S are readers, S should handle one of the two principal cases: (a) the predecessor is

![Figure 5.5: The two principal cases a reader requester should handle if it has a reader predecessor.](image-url)
Algorithm 6 Acquiring a reader lock in MQL.

```python
1 def reader_acquire(lock, my_qnode, timeout):
2     pred = XCHG(lock.tail, my_qnode)
3     if pred == NULL:
4         FAA(lock.nreaders, 1)
5         my_qnode.granted = True
6         return finish_reader_acquire(lock, my_qnode) # Algorithm 7
7
8     retry:
9     # Make sure the previous cancelling requester has left
10    spin until pred.next == NULL and pred.successory_type == None
11    if pred.type == Reader:
12        expected = [ busy = False | successor_type = None | status = Waiting ]
13        desired = [ busy = False | successor_type = Reader | status = Waiting ]
14        ret = CAS(pred.busy | successor_type | status], expected, desired)
15        if ret == expected: # CAS succeeded, predecessor was also waiting for the lock
16           my_qnode.pred = my_qnode
17           pred.next = my_qnode
18           if my_qnode.granted becomes True before timing out:
19               return finish_reader_acquire(lock, node)
20           return cancel_reader_lock(lock, node)
21
22    if ret.status.Leaving == True:
23        # Predecessor is leaving and will give me a new predecessor in relink()
24        # which requires the link between my_qnode and pred is setup
25        pred.next = my_qnode
26        my_qnode.prev = pred
27    spin until my_qnode.prev != pred
28    # Now I have a new predecessor, or the predecessor got the lock during
29    # cancellation and passed the lock to be in reader_acquire()
30    pred = XCHG(my_qnode.prev, NULL)
31    if pred == Acquired:
32        spin until my_qnode.granted == True
33        return finish_reader_acquire(lock, my_qnode)
34    goto retry # pred must point to a valid predecessor
35    else # ret.status must be Granted
36        pred.next = NoSuccessor
37        FAA(lock.nreaders, 1)
38        my_qnode.granted = True
39        return finish_reader_acquire(lock, my_qnode)
40    else # predecessor is a writer, spin-wait with timeout
41        pred.successory_type = Reader
42        pred.next = my_qnode
43        if XCHG(my_qnode.prev, pred) == Acquired:
44            spin until my_qnode.granted becomes True
45        elif if my_qnode.granted == False after timeout:
46            return cancel_reader_lock(lock, my_qnode)
47        return finish_reader_acquire(lock, my_qnode)
```

also waiting for the lock (lines 15–19) or (b) the predecessor has got the lock (lines 36–39). Figure 5.5 visualizes these two cases (the busy field is omitted in the figure as it is not required without cancellation). In case (a), S needs to wait for P to pass it the lock; in case (b), S should be granted the lock immediately because the predecessor is also a reader and it is already granted the lock. Note that the predecessor might be going through the same process and have the Granted bit set in its status at any time.
Algorithm 7 Helper routine for reader.acquire.

```python
def finish_reader_acquire(lock, my_qnode):
    my_qnode.[ busy, status ] = [ True, Granted ]
    spin until my_qnode.next != SuccessorLeaving # Ensure to have a stable successor
    if lock.tail == my_qnode: # No successor if tail still points to my_qnode
        my_qnode.busy = False
        return LockAcquired
    # The successor will not be able to cancel now
    spin until my_qnode.next != NULL
    s = my_qnode.next
    if s == NoSuccessor or s.type == Writer or CAS(my_qnode.next, s, NoSuccessor) != s:
        my_qnode.busy = False
        return LockAcquired
    # Handled the writer successor case above, now the successor (if any) must be a reader
    if my_qnode.status == [ Leaving = True, Granted = True ] and my_qnode.successor_type == None:
        # The successor might be trying to leave too, wait for it to realize my status
        # (after its first CAS in reader.acquire)
        spin until s.prev == my_qnode
        if CAS(s.prev, my_qnode, Acquired) == my_qnode:
            FAA(lock.nreaders, 1)
            s.granted = True
            my_qnode.next = NoSuccessor
            break
    # The successor will set its own prev field to point to my_qnode, not Acquired
    spin until s.prev == my_qnode
    if CAS(s.prev, my_qnode, Acquired) == my_qnode:
        FAA(lock.nreaders, 1)
        s.granted = True
        my_qnode.next = NoSuccessor
        break
    my_qnode.busy = False
    return LockAcquired
```

Therefore, S must use a compare-and-swap instruction (CAS) to try “registering” itself in the predecessor’s successor_type field, by changing the composite of busy, successor_type and status from [ False, None, Waiting ] to [ False, Reader, Waiting ] (lines 12–14 in Algorithm 6). These three fields, as a result, must fit within the same memory word to be modified by a single CAS instruction. The busy field is used to handle cancellation, and we discuss its use later. If the CAS is successful, the requester S will set up its double-direction link with P (lines 16–17 of Algorithm 6) and continue to spin on its granted field until it is toggled by the predecessor (line 18, ignore the timeout for now). If the CAS is unsuccessful, it indicates that the predecessor is already holding the lock, therefore S should toggle its granted field by itself (line 38). In either case, when the granted field is toggled to True, the requester S at lines 19 and 39 calls finish_reader_acquire (Algorithm 7). This helper function sets the Granted bit in S.status, so that if an incoming reader successor T issues a CAS at line 14 of Algorithm 6, it will
Algorithm 8 MQL’s cancellation routines for readers.

```python
def reader_cancel(lock, my_qnode):
    pv = XCHG(my_qnode.prev, NULL) # After this the predecessor cannot cancel
    if pv == Acquired:
        spin until my_qnode.granted == True
        return finish_reader_acquire(lock, my_qnode)
    my_qnode.status.Leaving = True # Announce my leaving status
    spin until my_qnode.next != SuccessorLeaving # Ensure successor is stable
    if pv.type == Reader:
        return reader_cancel_with_reader_pred(lock, my_qnode, pv) # Algorithm 9
    else
        return reader_cancel_with_writer_pred(lock, my_qnode, pv) # Algorithm 11
```

fail and jump directly to line 22 (or 36 without considering cancellation) of Algorithm 6, thus obtaining
the lock immediately. If T successfully executed this CAS, however, S’s finish_reader_acquire then
needs to wake T up, because T would be spinning at line 18 of Algorithm 6. At lines 11 and 31–34 of
Algorithm 7, S follows its next pointer, then tries to set T.prev to a sentinel value Acquired and toggles
T.granted to break T out from the spinning. This is also the procedure P would follow if S’s CAS at line
14 of Algorithm 6 was successful in case (a).

Conflicting requests. If a requester S conflicts with its predecessor P (i.e., reader-writer, writer-
writer, and writer-reader), S must spin-wait for its predecessor to toggle its granted field. In the original
reader-writer MCS lock [120], this is as simple as setting the P.next and P.successor type, and then
spinning on S.granted until it is toggled to True. However, MQL needs to support cancellation, requiring
a doubly linked list of requesters, so that a cancelling requester can install a new predecessor and successor
for its successor and predecessor, respectively. MQL uses an additional handshake protocol to set up the
double-direction link between P and S. Here we describe how it works using an example where a reader is
queued up after a writer. Suppose R is a reader and has queued up after a writer W. As shown by lines
41–42 in Algorithm 6, R first registers itself in W’s successor_type and next fields, then sets its own prev
to point to R using an atomic swap instruction (as we will discuss later, the cancellation protocol requires
atomic swap, instead of a blind write). The writer W will pass R the lock upon lock release. Before R sets
W.next, W has to spin-wait until its next field becomes a valid pointer. After R registered itself in W’s
qnode, W will be able to follow its next pointer to find R’s qnode, and retries a CAS on R.prev, hoping to
change it from a pointer from W to Acquired. W then toggles the R.granted field to pass R the lock.

Cancellation

The gist of the cancellation algorithm is to (1) ensure the cancelling requester’s predecessor and successor
are stable and (2) link properly the predecessor and successor after cancellation. In case the cancelling
requester happens to have obtained the lock during cancellation, then it aborts the cancellation and
finishes whatever needed after acquiring the lock. For example, if the cancelling requester is a reader, it
needs to make sure its reader successor (if there is any) also gets the lock.

A cancelling requester uses additional fields in its qnode to handshake with its neighbors: the prev
field in the qnode as a channel to coordinate with the predecessor, and the next and status fields to
Algorithm 9 Cancelling a reader lock with a reader predecessor.

```
def reader_cancel_with_reader_pred(lock, my_qnode, pred):
    retry:
        if pred.type == Reader:
            spin until pred.successor_type == Reader and (pred.next == my_qnode or pred.next == NoSuccessor)
        des = exp = [next = my_qnode | busy = False | successor_type = Reader |
                     status.Waiting = True | status.Granted = False]
        # Set pred.next to SuccessorLeaving to tell predecessor about my leave
        des.next = SuccessorLeaving
        ret = CAS(pred.next | busy | successor_type | status], exp, des)
        if ret == expected: # Predecessor did not change and still waiting
            if my_qnode.successor_type == None and CAS(lock.tail, my_qnode, pred) == my_qnode:
                pred.successor_type = None
                pred.next = NULL
                return LockCancelled
        relink(pred, my_qnode) # Remove myself from the requesters list, defined below
    else
        # Predecessor not waiting any more, must reset my prev field so it can continue
        my_qnode.prev = pred
        if ret.status == Granted: # Predecessor got the lock, I should, too
            spin until my_qnode.granted == True
            return finish_reader_acquire(lock, my_qnode)
        else # Predecessor is also leaving, it will set my_qnode.prev to let me know the final result
            spin until my_qnode.prev != pred
            if pred == Acquired: # Turns out we have acquired the lock
                goto retry # Got a new predecessor, retry
            pred = XCHG(my_qnode.prev, NULL)
            return finish_reader_acquire(lock, my_qnode)
        goto retry # Got a new predecessor, retry
    return reader_cancel_with_writer_pred(lock, my_qnode, pred)
    return LockCancelled

def relink(pred, my_qnode): # Setup the links between (new) predecessor and successor
    # Link the predecessor and myself
    spin until my_qnode.next is not NULL
    while True:
        # Preserve pred's flags and update its successor type and next pointer, i.e.,
        # give my predecessor a new successor
        exp = des = pred.next | busy | successor_type | status
        des.next = my_qnode.next
        des.successor_type = my_qnode.type
        if CAS(pred.next | busy | successor_type | status], exp, des) == exp:
            break
        # Link the successor with its new predecessor
        retry until CAS(my_qnode.next.prev, my_qnode, pred) == my_qnode
```

coordinate with the successor. Now we explain how cancellation works for a reader-lock request; the writer case works similarly and is simpler.

Algorithm 8 shows the overall cancellation algorithm for readers. At line 2, the cancelling requester CR atomically puts NULL in prev using an XCHG instruction. If the predecessor has already passed the lock to CR (e.g., at line 32 of Algorithm 7), the XCHG will return Acquired; CR then continues at lines 4–5 and returns. Otherwise, the return value will point to the predecessor. The XCHG instruction ensures CR will
have a stable predecessor, which cannot leave the queue until CR is done, because CR.prev points to NULL and the predecessor can only leave by successfully executing a CAS that changes CR.prev from a pointer to itself to the new successor (e.g., at line 43 of Algorithm 9). CR then tries to make sure the successor pointed to by its next field is stable, by setting status to Leaving so that the successor will not be able to leave, either. For example, this ensures the CAS at line 9 of Algorithm 9, which “notifies” CR that the successor is cancelling with a sentinel value of SuccessorLeaving, to fail; however, the successor’s CAS might have already succeeded, we therefore spin at line 7 to make sure that next does not contain SuccessorLeaving, i.e., wait for it to point to a new successor or NULL if there is no successor.

After making sure the pointers to predecessor and successor are stable, the algorithm proceeds differently, depending on whether the predecessor is a reader or writer. We explain in detail the more complex case where the predecessor is also a reader. Algorithm 9 shows the detailed algorithm. At lines 5–9, CR uses a CAS to change the predecessor’s next from a pointer to CR to SuccessorLeaving. Note that we conduct this CAS against the whole composite field of next, busy, successor_type and status. If the CAS is successful, i.e., the predecessor is not cancelling or changing its state, we then check if there is any successor already lined up after CR, and if so, we make the predecessor’s next field point to this new successor. This is done by calling another helper function relink shown by lines 34–46 of Algorithm 9. If, however, there is no new successor (line 13 of Algorithm 9) the lock request is successfully canceled and CR will reset the predecessor’s successor_type and next to indicate that there is no successor.

Lines 18–29 of Algorithm 9 handles the case where the attempt of “telling” the predecessor P that CR is cancelling was unsuccessful. If P has acquired the lock (i.e., the Granted bit is set in P.status), CR should also be granted the lock because both P and CR are readers. Note that P is only able to wake up CR if CR.prev points to P. Therefore, at line 19 we reset my_node.prev to pred and spin-wait for P to toggle my_node.granted in CR’s qnode (line 21).

While CR is executing lines 4–9 of Algorithm 9, the predecessor P might also be cancelling, i.e., executed line 6 of Algorithm 8. The CAS at line 9 of Algorithm 9 will fail, and CR then expects the predecessor to give it a new predecessor or grant it the lock (lines 23–29). Note that at line 19 we have reset my_node.prev to point to P (denoted as pred in the Algorithm). CR then spins on the same field to expect for a new predecessor. In case the predecessor obtained the lock, it would issue an XCHG on CR.prev to change it to Acquired, passing the lock to CR. Otherwise, once CR noticed a new predecessor appeared in prev, it jumps back to line 2 of Algorithm 9 to retry.

Lock Release

With details on how lock acquire and cancellation work in MQL, now we turn to how lock release works. Again we use the reader’s lock release routine as an example to explain the algorithm. The protocol for writers works similarly and is less complex.

A reader that is just granted the lock will try to find out if it has a waiting reader successor and pass it the lock if so (Algorithm 7); otherwise the latter reader requester will automatically acquire the lock. Therefore, upon lock release, a reader lock holder only needs to check whether a successor exists, and pass the lock to it if it is a writer. We show the details in Algorithm 10. At lines 5–8, a releasing reader first checks whether it has a successor: if the qnode’s next field is not pointing to any qnode, the releaser will proceed to issue a CAS on the lock tail to change it from a pointer to its own qnode to NULL. If the CAS is not successful, the algorithm jumps back to line 6 and retries.

Once the releaser is sure that a successor exists, if the successor is a writer, the releaser will put a
Algorithm 10 Releasing a reader lock in MQL.

```python
def reader_release(lock, my_qnode):
    my_qnode.busy = True
    spin until node.next != SuccessorLeaving

    # See if we have a successor
    while my_qnode.next == NULL:
        if CAS(lock.tail, my_qnode, NULL) == my_qnode:
            goto finish

    # Prepare the writer successor for getting the lock
    successor = my_qnode.next
    if successor != NoSuccessor and my_qnode.successor_type == Writer:
        lock.next_writer = successor
        retry until CAS(successor.prev, my_qnode, NULL) == my_qnode

    finish:
    # Wake up the writer if I am the last reader
    if FAA(lock.nreaders, -1) == 1:
        next_writer = lock.next_writer
        if next_writer != NULL and lock.nreaders == 0 and
           CAS(lock.next_writer, next_writer, NULL) == next_writer:
            retry until CAS(next_writer.prev, NULL, Acquired) == NULL
        next_writer.granted = True
```

pointer to the writer’s `qnode` in `lock.next_writer`, so that the last reader that releases the lock will be able to pass the lock to the writer. As shown by lines 11–14 of Algorithm 10, the protocol is very similar to that employed by the reader-writer MCS lock, except that MQL needs to handle the special value `SuccessorHandled`, which indicates there is no need for the releaser to handle successor, which has already obtained the lock. For example, recall that a reader $P$ will wake up its reader successor $S$ if $S$ registered itself with $P$ when $P$ is still waiting for the lock. At lines 24 and 35 of Algorithm 7, the `next` field is set to `SuccessorHandled` after waking up the reader successor. Similar to the handshake protocols we have used before, the releaser retries to issue a `CAS` to change the successor’s `prev` field to `NULL` at line 14 of Algorithm 10.

Finally, the releaser checks whether it is the last reader. If this is case, it will try to wake up the writer pointed to by `lock.next_writer`. The process adds one more handshake step at line 23—the writer $W$ might be cancelling and the releaser then must make sure `Acquired` is set in $W.prev$.

Algorithm 11 shows the details on cancelling a reader lock with a writer predecessor. Algorithms 12–15 show the acquire/release/cancel routines for writers, which employ similar handshake protocols. Algorithms for the writers are in fact easier than the readers’ algorithms as they need to handle fewer cases (e.g., no compatible requests possible). We thus do not describe the details here verbosely.

### 5.4.3 Instant-Try Interfaces

Cancelling a lock request involves multiple atomic instructions on neighboring `qnodes` and potentially on the lock word. Under heavy contention, it is more costly to cancel a lock request than to wait unconditionally or to try to acquire the lock instantaneously. MQL supports a simple instant-try approach for writers without leaving a `qnode` in the requesters list if the request is unsuccessful: a writer
Algorithm 11 Cancelling a reader lock with a writer predecessor.

```python
def reader_cancel_with_writer_pred(lock, my_qnode, pred):
    retry:
    # Wait for the cancelling predecessor to finish setting up prev/next links
    spin until pred.next == my_qnode and pred.successor_type == Reader
    # Predecessor is a writer, so I can leave as long as it is not cancelling or releasing
    while True:
        pred_status = pred.status
        pred_busy = pred.busy
        if pred_status & Leaving:
            # Predecessor is also cancelling, wait for a new predecessor
            my_qnode.prev = pred  # Put the pred value back so the predecessor can succeed
            pred = XCHG(my_qnode.prev, NULL)  # Check for a new predecessor (optimize with backoff)
        if pred == NULL or pred == Acquired:
            spin until my_qnode.granted == True
        return finish_reader_acquire(lock, my_qnode)
    else
        # Got new predecessor, now make sure successor cannot leave
        my_qnode.status.Leaving = True
        spin until my_qnode.next != SuccessorLeaving
        if pred.type == Reader:
            return reader_cancel_with_reader_pred(lock, my_qnode, pred)
        goto retry  # Retry the cancellation with the new writer predecessor
    elif is_busy == True:
        # Predecessor is releasing the lock
        my_qnode.prev = pred
        spin until my_qnode.granted == True
        return finish_reader_acquire(lock, my_qnode)
    # Predecessor is waiting, try to tell it I am leaving
    expected = [next = my_qnode | eflags | my_qnode]
    if CAS(pred.(next | flags), expected, eflags | SuccessorLeaving) == expected:
        break
    # Predecessor now has SuccessorLeaving on its next, it won't try to wake me up during release
    # Link the new successor and predecessor
    if my_qnode.next == NULL and CAS(lock.tail, my_qnode, pred) == my_qnode:
        pred.successor_type = None
        pred.next = NULL
        return LockCancelled
    # Setup the links between (new) predecessor and successor (line 34 of Algorithm 9)
    relink(pred, my_qnode)
    return LockCancelled
```

requester with a pre-prepared `qnode` W issues a `CAS` against the whole lock word, including `tail`, `nreaders`, and `next_writer`. The `CAS` tries to change the lock word from `[ nreaders = 0, next_writer = NULL, tail = NULL ]` to `[ nreaders = 0, next_writer = W, tail = NULL ]`. The lock is acquired if this `CAS` succeeds.

It is tempting to “improve” this approach for readers by inspecting the `qnode` stored in `lock.tail` and issuing a `CAS` to attach if it is a reader holding the lock. We note that this inspection is not safe as it is possible that the reader might have just changed `lock.tail` from a pointer to its own `qnode` to `NULL` using a `CAS` and thus successfully released the lock. It is unsafe for the new requester to inspect the predecessor’s `qnode` as it might be recycled any time.
Algorithm 12 Acquiring and releasing a writer lock in MQL.

```python
def writer_acquire(lock, my_qnode, timeout):
    pred = XCHG(lock.tail, my_qnode)
    if pred == NULL:
        # Lock appears to be free, lock acquired if no reader is still holding the lock
        lock.next_writer = my_qnode
        if lock.nreaders == 0 and XCHG(lock.next_writer, NULL) == my_qnode:
            my_qnode.granted = True
            return Acquired
    else:
        # Make sure the old cancelling predecessor is gone
        spin until pred.successor_type == None and pred.next == NULL
        # Register with predecessor as a writer successor
        pred.successor_type = Writer
        pred.next = my_qnode
        if XCHG(my_qnode.pred, pred) == Acquired:
            timeout = Never
        if my_qnode.granted becomes True before timing out:
            my_qnode.status = Granted
            return Acquired
    return writer_cancel(lock, my_qnode)

def writer_release(lock, my_qnode):
    my_qnode.busy = True # Make sure successor cannot leave
    spin until my_qnode.next != SuccessorLeaving
    while my_qnode.next != NULL:
        if CAS(lock.tail, my_qnode, NULL) == my_qnode:
            return
    retry until CAS(my_qnode.next.prev, my_qnode, Acquired) == my_qnode
    if my_qnode.next.type == Reader:
        FAA(lock.nreaders, 1)
    my_qnode.next.granted = True
```

Therefore, this approach can be adapted for readers, but in a limited way: the reader can only issue a CAS against the entire lock word expecting no concurrent threads are holding the lock—as if it were a writer.

5.4.4 Implementation

At a first glance, MQL may require double-width CAS that is capable of conducting the CAS operation on 16-byte words (DWCAS, such as CMPXCHG16B [72]) for the instant-try APIs and to change the composite of [ next, busy, successor_type, status ] used in previous algorithms. Note, however, that MQL does not require double CAS which takes two arbitrary words and changes them to two desired values from two expected values. Instead, we layout the fields adjacently, so DWCAS that works with two contiguous pointers (16 bytes) suffices.

Nevertheless, our implementation only requires normal 8-byte CAS, taking advantage of FOEDUS's
Algorithm 13 Main routine for cancelling a writer lock.

```python
def writer_cancel(lock, my_qnode):
    start_cancel:
    pred = XCHG(my_qnode.prev, NULL)
    if pred == Acquired:
        spin until my_qnode.granted == True
        my_qnode.status = Granted
        return Acquired

    # Announce my intention to cancel and make sure the successor field is stable
    my_qnode.status = Leaving
    spin until my_qnode.next != SuccessorLeaving

    if pred == NULL:
        # Predecessor already left (cancelled or released lock).
        return writer_cancel_with_no_pred(lock, my_qnode)

    while True:
        # Wait for the cancelling predecessor to finish relink
        spin until pred.next == my_qnode and pred.successor_type == Writer
        pred_status = pred.status
        pred_busy = pred.busy
        if pred_status.Leaving == True:
            # Predecessor might be cancelling, reset my_qnode.pred for it to continue and retry.
            my_qnode.prev = pred
            goto start_cancel
        elif pred_busy == True:
            if pred.type == Writer: # Predecessor is a writer and is releasing the lock
                my_qnode.prev = pred
                spin until my_qnode.granted == True
                my_qnode.status = Granted
                return Acquired
            pred = XCHG(my_qnode.prev, NULL)
            if pred == 0:
                return writer_cancel_lock_no_pred(lock, my_qnode)
            elif pred == Acquired:
                spin until my_qnode.granted == True
                my_qnode.status = Granted
                return Acquired
                continue # retry

    # Proceed with cancellation and inform the predecessor
    expected = desired = [busy = False | status = pred_status | next = pred]
    expected.next = SuccessorLeaving
    if CAS(pred.[next | busy | status], expected, desired) == expected:
        return finish_writer_cancel(lock, my_qnode, pred)
```

shared memory design. FOEDUS places records (thus locks) and qnodes in shared memory segments (e.g., allocated using `shmget`). A transaction worker process/thread can access locks and qnodes after it has attached to the allocated shared memory space. Instead of using pointers, a thread accesses qnodes via integer offsets into the shared memory space. Each thread has a set of pre-allocated qnodes for better performance. The composite of thread ID and an index into the thread-local qnode array uniquely identifies a qnode. The thread ID and qnode index in our current implementation each occupies 16 bits.
def finish_writer_cancel(lock, my_qnode, pred):
    if my_qnode.next == NULL and CAS(lock.tail, my_qnode, pred) == my_qnode:
        pred.successor_type = None
        pred.next = NULL
        return LockCancelled
    spin until my_qnode.next != NULL
    successor = [my_qnode.next | my_qnode.successor_type | busy = False | status = 0 (waiting)]
    retry:
    expected = pred.[next | successor_type | busy | status]
    wakeup_reader = False
    if pred.type == Reader and my_qnode.next.type == Reader and pred.status.Granted == True:
        # There is a time window which starts after the predecessor finished its "acquired" block
        # and ends before it releases. During this period my relink is essentially invisible to
        # the predecessor. Wake up the successor if this is the case.
        successor.next = NoSuccessor
        wakeup_reader = True
    if pred.busy == True:
        successor.busy = True
    if CAS(pred.[next | successor_type | busy | status], expected, successor) != expected:
        goto retry
    # Wake up the reader successor if needed
    if wakeup_reader:
        FAA(lock.readers, 1)
    my_qnode.next.granted = True
    retry until CAS(my_qnode.next.pred, my_qnode, Acquired) == my_qnode
    else
        retry until CAS(my_qnode.next, my_qnode, pred) == my_qnode
    return Cancelled

(32 bits in total). Thus, the next field is a 4-byte integer in FOEDUS, leaving enough bits in an 8-byte word for busy, stype, and status. Similarly, the prev field is a 4-byte integer, co-located with the type and granted fields in an 8-byte word.

In the lock word, nreaders and tail occupy 2 and 4 bytes, respectively. For compatibility reasons [173], we keep the lock word within 8 bytes. This leaves 2 bytes for next_writer. We store only the writer’s thread ID in next_writer, and use a thread-local lock–qnode mapping to identify the qnode. The instant-try methods therefore also only rely on the normal 8-byte CAS instruction.

5.4.5 Using MQL in MOCC

Table 5.2 summarizes how MOCC uses MQL in various situations. MOCC requests either a read lock or a write lock depending on the type of the record access. The primary way to acquire locks in MOCC is the unconditional mode because it is the most performant code path; it simply waits on the qnode’s granted field.

When a transaction is not in canonical mode, however, we might need to use the cancellation functionality. Specifically, a transaction issues a lock request in either try or asynchronous mode. Both of them push the qnode into the requesters list. The try mode then instantaneously gives up and removes
Algorithm 15 Cancelling a writer lock without predecessor.

```python
def writer_cancel_with_no_pred(lock, my_qnode):
    spin until lock.next_writer != NULL or my_qnode.granted == True
    if my_qnode.granted == True or CAS(lock.next_writer, my_qnode, NULL) != my_qnode:
        spin until my_qnode.granted == True
        my_qnode.status = Granted
        return Acquired
    # lock.next_writer is NULL, try to fix the lock tail (similar to release with no successor)
    if my_qnode.next == NULL and CAS(lock.tail, my_qnode, NULL) == my_qnode:
        return Cancelled
    spin until my_qnode.next != NULL
    next = my_qnode.next
    # If the successor is a writer, put it in lock.next_writer
    if next.type == Writer:
        lock.next_writer = next
        retry until CAS(next.prev, my_qnode, NULL) == my_qnode
    # Cancelled myself but passed the lock to a successor
    if lock.nreaders == 0 and CAS(lock.next_writer, next, NULL) == next:
        retry until CAS(next.prev, NULL, Acquired) == NULL
    next.granted = True
    else:
        retry until CAS(next.prev, my_qnode, Acquired) == my_qnode
        FAA(lock.nreaders, 1)
    next.granted = True
    return LockCancelled
```

Table 5.2: Using MQL in MOCC.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Use in MOCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reader/Writer</td>
<td>Allows concurrent readers. Write is exclusive.</td>
<td>All cases</td>
</tr>
<tr>
<td>Unconditional</td>
<td>Indefinitely wait until acquisition.</td>
<td>Canonical mode.</td>
</tr>
<tr>
<td>Try</td>
<td>Instantaneously gives up. Remove qnode.</td>
<td>Non-canonical mode. Record access.</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Leaves qnode for later check. Allows multiple requests in parallel.</td>
<td>Non-canonical mode. Record access and pre-commit (write set).</td>
</tr>
</tbody>
</table>

the queue node when the lock is not immediately acquirable. The asynchronous mode, on the other hand, leaves the queue node in the requesters list so that we can later check whether we acquired the lock.

5.5 Evaluation

We have implemented MOCC with MQL in FOEDUS [90]. We empirically compare the performance of MOCC with other methods on a variety of workloads. In particular, we confirm that:

- MOCC keeps OCC’s low overhead in low contention workloads (Section 5.5.2);
- MOCC’s selective locking achieves high scalability in high contention, low conflict workloads (Section 5.5.3);
• MOCC with MQL achieves significantly lower abort ratio and higher performance than both OCC and pessimistic CC in high contention, high conflict workloads (Section 5.5.4);

• MOCC can autonomously and quickly adjust itself in more realistic, dynamically shifting workloads on multiple tables with different nature (Section 5.5.6).

• MOCC is especially beneficial for long-running transactions (e.g., scan) with high conflict operations (Section 5.5.7).

5.5.1 Setup

We run experiments on machines listed in Table 5.3. Most experiments use the largest server, Gryphon-Hawk, which is equipped with 16 Intel Xeon E7-8890 processors, each with 18 physical cores. In total the server has 288 physical cores and 12 TB of main memory. The processor has 256 KB of L2 cache per core and 45 MB of L3 cache shared among all cores in each socket. For all experiments, we fit the whole database in memory. Each run lasts for 10 seconds, and is repeated for 3 (YCSB) or 10 (TPC-C) times.

Table 5.3: Hardware for Experiments.

<table>
<thead>
<tr>
<th>HP Model</th>
<th>EB840</th>
<th>Z820</th>
<th>DL580</th>
<th>GryphonHawk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sockets</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Cores (w/HT)</td>
<td>2 (4)</td>
<td>16 (32)</td>
<td>60 (120)</td>
<td>288 (576)</td>
</tr>
<tr>
<td>CPU (GHz)</td>
<td>1.90</td>
<td>3.40</td>
<td>2.80</td>
<td>2.50</td>
</tr>
<tr>
<td>DRAM</td>
<td>4GB DDR3</td>
<td>16 GB DDR3</td>
<td>512 GB DDR3</td>
<td>12 TB DDR4</td>
</tr>
</tbody>
</table>

CC schemes and systems

We compare MOCC with a variety of CC schemes and systems. All the CC schemes used in our experiments are serializable. Throughout this section, FOEDUS denotes the original FOEDUS to represent the performance of pure OCC. MOCC denotes the modified version of FOEDUS that implements the MOCC protocol with MQL. We use per-page temperature and H=10 as the temperature threshold unless noted otherwise. To compare with pessimistic approaches and implementations, we evaluate a few pessimistic schemes, one in FOEDUS and others in Orthrus [145], a recent system that targets high contention workloads. We compare with the following schemes:

PCC. PCC is a 2PL variant we implemented in FOEDUS. It uses MQL’s try interface to opportunistically take locks. If the lock cannot be granted immediately, the transaction will continue executing without taking the lock, thus is deadlock-free. At commit time, PCC tries to acquire write locks for records in the write set, but aborts if the lock cannot be granted immediately.

Orthrus. Orthrus separates CC and transaction worker threads for high contention scenarios. It avoids deadlock by enforcing CC threads to acquire locks in a well-defined order. This is achieved by inspecting transaction logics and/or pre-execution [165]. We configured Orthrus on our hardware with the kind help from the authors and verified its performance results with them. We set the ratio of CC:executor threads to 1:4.

Dreadlock/WaitDie/BlindDie. denote 2PL variants in Orthrus. They take locks before every access to guarantee serializability. All of them avoid deadlocks by aborting threads that see themselves potentially risking deadlock.
Table 5.4: TPC-C throughput (low contention, low conflict) on GryphonHawk. MOCC behaves like OCC. PCC has a moderate overhead for read locks. ERMIA is slower due to frequent and costly interthread communication.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Throughput [MTPS+Stdev]</th>
<th>Abort Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOCC</td>
<td>16.9±0.13</td>
<td>0.12%</td>
</tr>
<tr>
<td>FOEDUS</td>
<td>16.9±0.14</td>
<td>0.12%</td>
</tr>
<tr>
<td>PCC</td>
<td>9.1±0.37</td>
<td>0.07%</td>
</tr>
<tr>
<td>ERMIA</td>
<td>3.9±0.4</td>
<td>0.01%</td>
</tr>
</tbody>
</table>

**ERMIA.** Serializable MVCC is a popular choice in recent main-memory systems because of its read-friendliness [42, 89, 102]. ERMIA [89] is based on snapshot isolation and uses SSN (Chapter 4) for serializability. We use ERMIA as a representative for certification and MVCC based systems.

**Workloads**

We use TPC-C and YCSB for our evaluation. We use the stock TPC-C benchmark described in Chapter 2. For YCSB, we focus on read-only and read-modify-write (RMW) operations. Unless otherwise specified, the YCSB transactions used in our experiments access ten records; we vary the number of reads and RMW operations, as well as the total number of records to control contention level.

5.5.2 **TPC-C: Low Contention, Low Conflict**

We run TPC-C under MOCC, FOEDUS, PCC, and ERMIA. We have not tested TPC-C in Orthrus because its current implementation lacks support for range queries. Table 5.4 shows the results.

TPC-C has low contention. It contains some read-write conflicts, such as remote-warehouse accesses in payment and neworder, but still there are on average a very small number of threads touching the same records concurrently. Hence, even FOEDUS experiences only one in a thousand aborts. MOCC behaves exactly the same as OCC because the temperature statistics of almost all data pages are below the threshold, rarely triggering read locks.

For such low contention, low conflict workloads, PCC adds unwanted overhead of read locks for only a slight reduction in abort ratio. However, PCC, unlike the experiments that follow, is only ∼2× slower than FOEDUS/MOCC. The reason is TPC-C does not have much physical contention, thus locks are not heavily contended.

ERMIA has the lowest throughput (∼3.9 MTPS) among the four systems we tested for TPC-C, due to its centralized design. It frequently issues atomic instructions such as CAS on centralized memory locations to manage resources (e.g., transaction IDs). To guarantee serializability, ERMIA needs to stamp most of the tuples read, making reads become writes. The experiments confirm that having frequent interthread communication prevents the system from scaling up to hundreds of cores, even for low-contention workloads.

In summary, avoiding frequent interthread communication allows scaling up low contention/conflict workloads easily. Pessimistic schemes might incur additional overheads of read locks, but this is a fixed overhead rather than a scalability bottleneck.

---

2 The table does not count deliberate 1% aborts in neworder defined in the TPC-C spec because they are not race aborts.
5.5.3 High Contention, No Conflict YCSB

Now we start to focus on high contention cases: a large number of concurrent threads touch a small number of records. There are two kinds of highly contended workloads: low read-write conflict and high read-write conflict. This section evaluates the performance under the former type of workloads.

To show the effect of contention, we vary the scale of the hardware ranging from a laptop (HP EB840) to GryphonHawk as listed in Table 5.3. Part of this experiment was shown in Figure 5.1. In this section, we also show additional throughput numbers for the other systems. Figure 5.6 plots the throughput of a read-only YCSB workload where each transaction reads 10 records randomly from 50 records. Both MOCC and FOEDUS scale up perfectly because they do not cause any interthread communication. Pessimistic approaches—PCC and Dreadlock—are again slower than MOCC/FOEDUS, but this time they are $170\times$ slower because a huge number of concurrent threads must take read locks on the same records, causing severe physical contention. Furthermore, Dreadlock adds one more scalability bottleneck: frequent interthread communication to check and disseminate thread fingerprints. It is not a significant overhead on machines with up to two sockets, but it causes expensive cache-coherence communication on deep memory hierarchies that interconnect hundreds of cores. In the remaining experiments, we observed orders of magnitude slower performance on Dreadlock for this reason. This result clearly shows the limitation of pessimistic approaches under highly contended workloads.

Orthrus scales better than PCC, but it still needs frequent interthread communication between the execution threads and CC threads. YCSB accesses a completely random set of records per transaction with no locality whatsoever, which means that the number of lock manager threads involved in each transaction will be approximately equal to the number of records accessed. This represents the worst case scenario in Orthrus, since the number of messages sent per transaction is linear to the number of lock manager threads involved in that transaction. For this workload without any read-write conflicts, execution threads can quickly saturate CC threads, limiting the throughput to 2 MTPS. In some sense, Orthrus shifts the issue of contentious communication to another place. We also varied the fraction of executor and CC threads, but the throughput is not comparable to that of FOEDUS/MOCC ($>100$ MTPS).

On smaller scales with at most two sockets, ERMIA is only slower than FOEDUS and MOCC. As
we move to larger machines, ERMIA’s throughput drops significantly: from 3.5 MTPS (2 sockets) to 0.23 MTPS (16 sockets). But it is still much faster than Dreadlock for all cases. On large machines such as GryphonHawk, ERMIA’s centralized thread registration mechanism used by SSN (for guaranteeing serializability) becomes a major bottleneck, although it is not necessary for this workload. Without SSN, ERMIA can achieve as high as \(~4.6\) MTPS for the same workload on GryphonHawk under snapshot isolation. Interthread communication required by the other centralized mechanisms prevented ERMIA from scaling further.

5.5.4 High Contention, High Conflict YCSB

The next experiment also uses YCSB under high contention, but also includes RMW operations that can cause frequent read-write conflicts. Again we use a table of 50 records and let each transaction access 10 records. We vary the amount of RMWs between 0 and 10; the rest of the operations are pure reads. We access records in random order to evaluate how pessimistic approaches handle deadlocks.

Figure 5.7 shows the throughput and abort ratio of each CC scheme. As expected, FOEDUS’s throughput significantly drops due to massive aborts: as many as 98% of transactions abort on the highest conflict level. Somewhat ironically, although the pessimistic variants (PCC, Dreadlock, WaitDie, and BlindDie) protect reads using read locks, their performance still drops due to aborts caused by deadlocks. The workload accesses records in a random order. Therefore, traditional pessimistic approaches also suffer. Further, although the abort ratio of PCC is slightly lower than FOEDUS, each retry in PCC takes significantly longer than in FOEDUS, mostly busy-waiting for the lock. Thus, PCC’s throughput is a few times lower than FOEDUS except with one RMW where pessimistic waiting significantly lowers aborts. Again, ERMIA’s centralized design does not work well on large machines. With more writes, its performance gradually converges to PCC’s and becomes worse than it as the workload becomes write dominant with more than 5 RMWs. Most aborts are caused by write-write conflicts (instead of serializability violations), which abort at least one of the participating transactions.

MOCC and Orthrus sustain high performance in this workload. MOCC dramatically reduces aborts without adding noticeable overhead, resulting in orders of magnitude higher throughput than FOEDUS and PCC. Although Orthrus has zero deadlocks, MOCC is an order of magnitude faster than Orthrus for transactions with up to one RMW because Orthrus needs frequent interthread communication between
Figure 5.8: Low contention YCSB (1M records) on GryphonHawk. MOCC/OCC/PCC all scale well and behave similarly. Orthrus is an order of magnitude slower due to the communication between CC and executor threads.

Figure 5.9: Varying-contention YCSB on GryphonHawk with 8 reads and 2 RMWs.

executor and CC threads. For transactions with two or more RMWs, the maximal concurrency of the workload becomes fundamentally low. This shrinks the difference between MOCC and Orthrus to less than $2 \times$, but still MOCC is consistently faster. Interestingly, the performance of Orthrus increases from 1 RMW to 2 RMWs because contention on the lock manager/CC queues decreases, leading to faster interthread communication.

5.5.5 Varying Contention YCSB

Figure 5.9 varies the number of records, the amount of contention, between 50 (high contention) and one million (low contention). Each transaction issues eight reads and two RMWs.

The results confirm our observations on the effects of contention: MOCC and Orthrus have similar performance under high contention whereas MOCC and OCC achieve the highest performance under low contention. However, there are two interesting cases: 100 records and 10,000 records. With 100 records, the throughput of Orthrus drops rather than increases compared to with 50 records. Our hypothesis is that fewer conflicts allow more executor threads in Orthrus to synchronize with different CC threads at the same time. Compared with the 50 records case where almost all workers are waiting for the same CC thread and they proceed one by one, it makes a larger number of workers to fetch the same cacheline
into their L1/L2/L3 caches, causing more frequent cacheline invalidations onto remote CPU sockets. Yet, the fundamental concurrency in this case is not high enough to justify the communication overhead.

With 10,000 records, on the other hand, the throughput of MOCC does not increase from 1,000 records. This is the only case we observed that Orthrus performs significantly faster than MOCC. Orthrus is a static concurrency control scheme without any aborts. This sweet-spot for Orthrus happens when contention is not too low and is not too high. In such a case, all dynamic concurrency control schemes (e.g., MOCC, PCC) abort at least half of the runs and waste CPU cycles whereas Orthrus completes all runs with higher CPU utilization.

Note that, however, Orthrus is a static approach on assigning concurrency control threads. To benefit from physical partitioning, the workload must have a partitionable locality, and the user must know it beforehand. For totally random workloads, Orthrus’ applicability is limited without oracles. Such totally random workloads (e.g., those represented by the YCSB workloads in previous experiments in this section) are the worst case for Orthrus.

5.5.6 Multi-table, Shifting Workloads

Multi-table environment. Real-world transactions often access many tables of varying sizes and read-write patterns. Most workloads access large tables without high contention or conflict. Nevertheless, they also involve highly contended tables, either small tables or tables that receive skewed accesses. A monolithic scheme that works well for one might not for another.

To test MOCC in more realistic situations, we devise a two-table experiment. The first table contains one record, while the other contains one million. In each transaction, we issue one operation (initially a read) to the small table and 20 reads to the large table.

Shifting workload. Real workloads are also often dynamic, i.e., the contention level and access patterns change over time. Hence, this experiment also dynamically switches the nature of the small table every 0.1 second, issuing an RMW instead of a read. During the first 0.1 second the small table is highly contended but has no conflict, hence the optimal behavior is expected to be similar to what we have shown in Section 5.5.3. During the next 0.1 second it receives frequent conflicts, hence the optimal behavior is like the one in Section 5.5.4. We keep switching between the two states.

Figure 5.10 shows the throughput and abort ratio of MOCC over time with different temperature
Table 5.5: Long Scan Workload on GryphonHawk.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Throughput [kTPS+Stdev]</th>
<th>Abort Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOCC</td>
<td>199.6±3.1</td>
<td>0.4%</td>
</tr>
<tr>
<td>FOEDUS</td>
<td>10.5±0.0</td>
<td>99.55%</td>
</tr>
<tr>
<td>PCC</td>
<td>25.7±1.7</td>
<td>0%</td>
</tr>
<tr>
<td>Thomasian</td>
<td>20.8±1.5</td>
<td>50.1%</td>
</tr>
</tbody>
</table>

thresholds (H) for MOCC to trigger pessimistic locking. We periodically reset the temperature counters. During the no-conflict period, all threshold values perform the same except \( H = 0 \). \( H = 0 \) is significantly (24×) slower because it always takes read locks like pessimistic schemes. We emphasize that the transaction issues most (20 out of 21) operations on the low-contention table. A monolithic, pessimistic scheme slows down for 24× due to the single operation on the high-contention table. This is very similar to our observations in TPC-E’s trade_type table, which contains only five records and is very frequently read. Real workloads often contain such tables.

After switching to high-conflict cases (at 0.3, 0.5, 0.7 seconds), all thresholds except \( H = 0 \) observed aborts. Lower thresholds (4, 8, 10) result in faster learning and the abort ratio quickly drops to zero. Extremely large thresholds keep causing aborts and unstable throughput. For example, \( H = 20 \) requires \( 2^{20} \approx 1,000,000 \) aborts to trigger read locks, which is virtually a monolithic OCC. A monolithic scheme, again, does not work well due to a single high conflict operation, which is not rare in real workloads.

We observe that as long as the threshold is reasonably small, all MOCC runs autonomously adjust their behaviors to converge to the same robust throughput. When the threshold is too small, however, MOCC can be confused by just a small number of sporadic aborts, leading to lower performance. Hence, we recommend setting \( H \) to 5–10, depending on the scheme to reset/decrement the counter and on whether the counter is per-page or per-record.

5.5.7 Long Scan Workloads

Combining OLAP and OLTP. Real workloads often contain both analytic accesses (e.g., a long cursor scan) and transactional updates. The last experiment evaluates MOCC’s benefits on such hybrid transactions. Every transaction reads one record in the small table of the previous experiment, scans 1,000 records in the larger table, then updates the record in the small table. This time, we also compare with a hybrid OCC method proposed by Thomasian [163], which switches from pure OCC to pure 2PL after an abort. We implemented the algorithm on FOEDUS’s codebase.

Table 5.5 shows the result. The long running transaction with high read-write conflict is especially challenging for pure OCC. FOEDUS aborts more than 99.5% of the time, being the worst among all schemes. PCC completely avoids aborts and performs faster, but it incurs unwanted locking overheads for scanning. Thomasian’s method [163] sits exactly in-between. It aborts the initial OCC run (thus exactly 50% abort ratio) and pessimistically locks all records in read/write sets in the second run. MOCC performs an order of magnitude better than all others because of its temperature statistics. Rather than statically switching from pure OCC to pure 2PL, all runs in MOCC are hybrid, taking beneficial read locks within an OCC transaction. The result verifies that MOCC is especially beneficial for long-running transactions in real workloads.
5.6 Summary and Discussion

We have proposed mostly-optimistic concurrency control (MOCC) to enhance OCC for highly contended dynamic workloads, by judiciously using pessimistic locking for suitable records. The main challenge is to add read locks as of the read with minimal overhead. Acquiring locks during forward processing breaks OCC’s deadlock-free property. To handle locking and deadlocks efficiently, we advocate native locking, which employs synchronization primitives directly as database locks. We have devised the MOCC queuing lock (MQL), a cancellable, queue-based reader-writer lock that works well on massively parallel machines, especially those with deep memory hierarchies. Using MQL, MOCC handles deadlocks without discarding OCC’s simplicity and scalability. MOCC incurs negligible overhead over OCC and achieves robust performance for a broad spectrum of workloads, including highly contended, dynamically shifting and read-mostly ones. MOCC does not require any new modules or controller threads (such as those in Orthrus), thus is simple and easy to incorporate in various OCC database engines. The core of our MOCC implementation adds fewer than 1000 LoC on top of the base system, which has more than 100k LoC in total.

* * *

Compared with SSN, MOCC is more suitable for future hardware that will potentially feature thousands of CPU cores and deep memory hierarchies, where remote memory access (cross-socket) is much more expensive than local memory access. It is imperative to avoid unnecessary writes to shared memory. In addition, such hardware and database systems require better synchronization primitives to realize their designs, for both performance and functionality. We have seen the requirements of MOCC and proposed MQL in this chapter, focusing more on the performance aspect. In the next chapter, we take a closer look at the functionality aspect of synchronization primitives and propose the MCSg lock to enable broader and easier adoptions of the MCS lock [119] in complex software systems, including but not limited to database engines.
Chapter 6

MCS Locks that Welcome Guests

The MCS lock is one of the most prevalent queuing locks. It provides fair scheduling and high performance on massively parallel systems. However, the MCS lock mandates a bring-your-own-context policy: each lock user must provide an additional context (i.e., a queue node) to interact with the lock. We propose MCSg, a variant of the MCS lock that relaxes this restriction.

Our key observation is that not all lock users are created equal. We analyzed how locks are used in massively-parallel modern systems, such as NUMA-aware operating systems and database engines. We found that such systems often have a small number of “regular” code paths that enter the lock very frequently. Such code paths are the primary beneficiary of the high scalability of MCS locks.

However, there are also many “guest” code paths that infrequently enter the lock and do not need the same degree of fairness to access the lock (e.g., background tasks that only run periodically with lower priority). These guest users, which are typically spread out in various modules of the software, prefer context-free locks, such as ticket locks.

MCSg provides these guests a context-free interface while regular users still enjoy the benefits provided by MCS. It can also be used as a drop-in replacement of MCS for more advanced locks, such as cohort locking. We also propose MCSg++, an extended version of MCSg, which avoids guest starvation and reduces non-FIFO behaviors that might happen with MCSg.

Our evaluation using microbenchmarks and the TPC-C database benchmark on a 16-socket, 240-core server shows that both MCSg and MCSg++ preserve the benefits of MCS for regular users while providing a context-free interface for guests.1

6.1 Introduction

Concurrent threads and processes must coordinate accesses to shared data. A synchronization mechanism for the coordination, typically locks, must scale up to the growing concurrency of today’s massively parallel processors.

Compared to centralized spinlocks, such as test-and-test-and-set (TATAS) and ticket (TKT) locks, software queuing locks such as MCS [119] and CLH [38, 112] locks scale better under high contention. In particular, the MCS lock has two advantages that make it well suited for massively parallel computers. First, the lock-entry (doorway [94]) protocol in the MCS lock is wait-free and FIFO because it uses

1 This chapter highlights our work published in PPoPP 2016 [173].
Chapter 6. MCS Locks that Welcome Guests

(a) Frequently invoked code paths causing scalability bottleneck.

TATAS/ticket lock: context-free, but non-scalable.

MCS: scales better, but requires a Context in all code paths.

(a) Regular users enjoy the same benefits as with MCS.

(b) Guest users do not need context or code modifications.

MCSg: allows Guest Users yet keeps MCS’s scalability.

Figure 6.1: Not all lock users are created equal. Frequent lock users need high scalability while occasional “guests” need context-free locking, rather than the best performance. MCSg satisfies both users.

an atomic swap (XCHG) instruction rather than a compare-and-swap (CAS) instruction for a thread to enqueue itself. Second, once a thread has enqueued its request, it spins locally. Local spinning reduces interconnect traffic among CPU cores.

These advantages led to the successful adoption of MCS locks in real production software. For instance, the Linux kernel has recently begun to replace some of its non-scalable locks with MCS locks. The result is a $\sim 3-5 \times$ performance improvement [24] in major benchmarks. Recent scalable database
systems designed for parallel hardware have also adopted MCS locks for synchronization of their internal data structures [76, 90].

6.1.1 MCS Adoption Challenges and Key Observations

Figure 6.1 illustrates challenges to adopting MCS locks in complex production systems. The high performance and scalability of MCS locks come with the price of bring-your-own-context—each lock user must provide an extra “context”, or a queue node (qnode), in addition to a reference to the lock itself. A qnode is typically pre-allocated for each lock user in a NUMA-aware fashion so that the lock user spins locally on its own qnode. The qnode is often allocated in a region of memory shared between processes so that users in other processes can access it. Whether we allocate qnodes on-demand or in advance, the added complexity of bring-your-own-context impedes adoption of MCS locks.

One prime example is the adoption of the MCS lock in the Linux kernel for better scalability [24] done by developers at Hewlett Packard Enterprise. Throughout the efforts, the developers have repeatedly observed the performance benefit of the MCS lock as well as the challenge to its adoption. Working with the developers, we made two key observations in real codebases that motivated and guided this work.

**Key observation 1: complex code paths necessitate a context-free lock interface**

TATAS and TKT locks need no additional contexts. They require only that the user hold a pointer to the lock for acquiring and releasing the lock. Much existing code in real systems assumes such a “context-free” interface. In fact, by far the most widely used locking interface in the Linux kernel consists of the spin_lock(lock*) and spin_unlock(lock*) macro families, which receive only a pointer to the lock object. Countless invocations of these macros involve infrequent code paths that do not require the same level of scalability or fairness as more frequently executed code paths. Re-writing all code paths that employ these macros to appropriately allocate, pass, and de-allocate conventional MCS lock qnodes would be a formidable task with meager benefits.

Therefore, the Linux kernel initially limited the adoption of MCS locks only to certain places. The developer who introduced MCS locks to the Linux kernel stated that:

“When trying to convert some of the existing Linux kernel spinlocks to MCS locks, it was especially complicated when the critical section spanned multiple functions. This required some functions to accept additional MCS node parameters, which was not practical.” [108]

We observed the same issue in the FOEDUS open-source database [90]. In this case, FOEDUS has already adopted MCS locks for higher scalability, pre-allocating qnodes for each transaction processing thread. The problem arises when the database needs to add new functions that manipulate shared data protected by the existing MCS locks. The new functions are occasionally invoked from various modules without pre-allocated qnodes. Moreover, it is difficult to know a priori how many distinct threads will invoke the functions. The issue was more pressing because it impedes functionality rather than performance.

**Key observation 2: in most cases, complex code paths are infrequent while frequent code paths are simple**

A lock protects shared data against various code paths that access the data. While the complexity of each code path’s critical section varies, there is a strong correlation between the complexity and frequency of
the code paths. We observed many complex critical sections that cannot easily incorporate qunodes. It turns out that all such complex code paths infrequently enter the lock. Local spinning in these code paths thus does not improve scalability much. We never encountered frequent and complex code paths for an intuitive reason. Such complex critical sections run longer, and thus cannot be repetitively invoked over short durations. In fact, all of the frequent lock users that caused bottlenecks were found to have substantially simple critical sections. For example, the kernel developers found that one of the locking bottlenecks in Linux occurs when a 3-line code block triggers more than 100,000 lock acquisitions per second [109].

The kernel developer stated that:

"Out of the 300+ places that make use of the dcache lock, 99% of the contention came from only 2 functions. Changing those 2 functions to use the MCS lock was fairly trivial because both of them had straightforward lock/unlock calls within the same function. The call-sites with the complicated locking made up much less than 1% of the bottleneck since they were called less often." [108]

FOEDUS developers made a similar statement.

To summarize, we observed that skews and inequality fundamentally abound in locking. Throughout the entire kernel and database code, extremely frequent code paths (e.g., > 100k/s) in a highly contended lock are very rare. We can address 99% of the locking bottleneck by imposing the duty of a queue-based protocol on a few code paths (i.e., top 1% of lock users). On the contrary, 99% of the development cost to employ MCS locking is attributed to other, infrequent code paths. Relieving the other 99% of lock users from code modification significantly eases the adoption of MCS locks especially because they might have complex critical sections spanning multiple functions and modules. These observations naturally guide us to the dual-interface design of our new MCS lock variant explored in this chapter.

6.1.2 Summary

The key contribution of this chapter is a new variant of the MCS lock, the MCSg lock, which addresses the aforementioned issues. The MCSg lock provides a different interface for both frequent and infrequent code paths. The interface for frequent code paths, or "regular" users, is the same as that of the MCS lock. It provides the regular users with high scalability and fairness at the cost of bring-your-own-context. Another interface, for infrequent code paths, or "guest" users, is context free, but the code paths receive less scalability and fairness to access the lock.

MCSg facilitates adoption into complex systems in two ways. First, MCSg can replace MCS locks in existing code to allow new guest code paths, such as sporadic background tasks. MCSg is a perfect drop-in replacement for MCS that keeps all the good properties of MCS with minimal code changes. Second, MCSg can replace non-scalable context-free locks in existing code to improve scalability. Unlike with MCS, one can gradually adopt MCSg with minimal effort, starting with zero changes (all guest users), identifying a small number of frequent lock users, and then modifying only those specific code paths as regular users.

The rest of this chapter is organized as follows. Section 6.2 details the key properties MCSg is designed to satisfy. Section 6.3 explains the new MCSg algorithm. Section 6.4 proposes MCSg++, which extends MCSg to give fair scheduling in the presence of guest users. We have introduced how the MCS lock
works in Section 2.3.2, so it is not repeated here; interested readers may refer to the previous discussion for details. Section 6.5 empirically evaluates MCSg and MCSg++. Finally, Section 6.6 concludes.

6.2 Desiderata

MCS locks have properties that make them applicable to a wide range of settings. MCSg is designed to keep all of them in addition to the new context-free interface. This section details these desiderata to clarify the key principles behind MCSg.

6.2.1 High Scalability

As already described in Section 6.1, MCS provides a wait-free doorway, NUMA-friendly local spinning, and FIFO ordering among lock users. MCSg must maintain the same scalability at least for regular users unless guest users enter the lock extremely often, in which case such guest users should be modified to become regular users.

6.2.2 Simplicity, Applicability, and Pluggability

Performance is often deemed to be the most important factor for synchronization mechanisms. However, the simplicity of the algorithm sometimes weighs even more in practice, especially in huge, complex, and critical codebases, such as OSes and database engines. This is where more basic spinlocks (e.g., TATAS) are still preferable. For a locking mechanism, we categorize the concept of simplicity into the following aspects.

Single-Word Lock State

The MCS lock places just a single word as the shared state, the lock tail. The MCSg lock must keep this property. This has two benefits: reduced space consumption and code complexity. Some locking algorithms require additional words as the lock state. For example, cohort locks [43] have to reserve additional lock memory for each NUMA node.

Even when space consumption is not an issue, the complexity of the code to allocate/deallocate and identify such memory becomes significant. This issue is especially significant when we do not know a priori the number of locks or the number of lock users in the system, which is the case for many dynamic storage systems, such as database systems and file systems.

For instance, such applications often embed locks into data pages. It is not even known in advance which bytes will be used as a lock. Hence, trivial initialization/destruction by a simple memzero is vital.

Context-Free Interface

This is the feature the MCS lock does not provide. MCSg must provide a context-free interface similar to those of TATAS/TKT locks. It should receive only a pointer to the lock without any thread-specific context or global information.
Chapter 6. MCS Locks that Welcome Guests

Inter-Process Uses

The MCS lock is applicable to inter-process mutual exclusion. Many systems run a collection of individual processes with shared memory instead of running threads in the same process [90, 104, 111, 183]. For instance, virtually all major databases share memory among multiple processes.

Pointers (i.e., virtual addresses) do not work across processes. Hence, a typical MCS lock implementation on shared memory stores an identifier of the process/thread and a memory offset in each qnode [90]. This offset approach comes with an added benefit for advanced synchronization methods to be combined with the lock. It allows more bits than raw pointers for additional information, such as delete-flags and ABA counters [53].

Some locking algorithms, however, cannot use offsets because they share a pointer to stack memory or thread-local-storage (TLS) that does not allow accesses from another process. MCS does not have this issue; MCSg must also avoid it.

Environment Independence

Some lock algorithms depend on environment-specific features. For example, qspinlock, which has recently been introduced to the Linux kernel [36], requires kernel-space ability to disable pre-emption. Another example is a locking algorithm that requires efficient access to TLS. Some platforms are equipped with a special register to efficiently support TLS, e.g., the %fs segment register in x86 and tpidr_el registers in ARM. Without such hardware support, accessing TLS involves far higher costs. Even x86/ARM incurs high overhead for TLS variables in shared libraries or other modules due to the cost of adjusting TLS offset (e.g., _tls_get_addr).

MCS does not demand TLS, and MCSg must not.

Composability

The MCS lock can easily be used in combination with other locks, albeit not as easily as TATAS. For instance, an MCS lock can trivially provide the cohort-detection [43] property by checking its own qnode and provide the thread-obliviousness [43] property with a minor change. MCSg should keep the exact same composability as MCS. In other words, MCSg must be a drop-in replacement for MCS, and work everywhere MCS works.

6.3 Basic MCSg Locks

We now introduce MCSg, a new variant of the MCS lock that satisfies all desiderata in Section 6.2. Algorithm 16 shows the MCSg algorithm, along with the proper memory barriers that need to be associated with load and store instructions. As shown by Algorithm 16, MCSg only slightly modifies the original MCS algorithm. MCSg does not require any change to the MCS lock’s data structure and adds only additional logic in the lock acquire procedure for regular users. It behaves exactly the same as the original MCS lock when there are no guests. The basic idea is for guests to treat the MCS lock word (the tail pointer) like a TAS/TATAS lock when trying to acquire it; regular users will spin-wait when they notice that a guest has acquired the lock and re-join the queue after the guest has released the lock. The following subsections describe in more detail how guests and regular users interact with MCSg locks.
Algorithm 16 MCSg Algorithm. Guests simply spin with CAS. Regular users differ from original MCS only in lines 7-10.

```python
def regular_acquire(lock_tail, my_qnode):
    tail_qnode = my_qnode
retry:
    pred = XCHG(lock_tail, tail_qnode)
    if pred == NULL:
        return
    elif pred == π:
        # A guest has the lock, put back π and retry
        tail_qnode = XCHG(lock_tail, π)
goto retry
    else:
        # A regular user holds the lock, join the queue
        my_qnode->flag = WAITING ............................... <release barrier>
        pred->next = my_qnode ................................. <release barrier>

# Spin on my (local) wait flag
spin_while my_qnode->flag != GRANTED ............................... <acquire barrier>
return

def regular_release(lock_tail, my_qnode):
    # Exactly the same as the original MCS lock, included for completeness:
    if my_qnode->next == NULL:
        if CAS(lock_tail, my_qnode, NULL) == my_qnode:
            return
        spin_while my_qnode->next == NULL
        my_qnode->next->flag = GRANTED

    def guest_acquire(lock_tail):
        while (CAS(lock_tail, NULL, π) != NULL)

    def guest_release(lock_tail):
        while (CAS(lock_tail, π, NULL) != π)
```

6.3.1 Guests

The lock is treated like a TAS/TATAS lock for guests trying to acquire it. Instead of joining the wait queue using an XCHG instruction, the guest issues a CAS against the lock tail (lines 28–29), trying to change it from NULL to a special sentinel value π. The thread retries until the CAS succeeds, which indicates that it has acquired the lock.

Lock release is also straightforward for guests although it is more complex than in TAS/TATAS. The lock holder issues (and retries if it fails) a CAS against the lock tail (lines 31–32), trying to change it from π back to NULL. The retry is needed because the lock word may become non-π as regular users try to acquire the lock (details described in the next section).

In a nutshell, the guest only needs to issue and retry a CAS in each acquire and release operation. The only requirement for guests is to hold a pointer to the lock word. They do not have to provide a self-prepared qnode (context). The acquire procedure for regular users provides guarantees for this machinery to work.
6.3.2 Regular Users

In the original MCS lock’s acquire procedure, lock users could see either a valid (lines 11–17) or a null pointer (lines 5–6) on swapping the tail pointer. In the MCSg lock’s acquire procedure, however, they might see the sentinel value $\pi$ when the lock is being held by a guest user.

To handle this case, MCSg adds a “swap-and-spin” loop (lines 7–10) for the regular user when it notices that a guest holds the lock. Suppose a guest user has acquired the lock right before a regular user $R$ comes to line 4. $R$ will receive $\pi$ as the return value on performing $\text{XCHG}$ with the lock tail. In this case, $R$ will first $\text{XCHG} \pi$ back to the lock tail. Note that this $\text{XCHG}$ can return a value $V$ that might not be the same as a reference to $R$. This is because, between the time $R$ performed an $\text{XCHG}$ with the tail pointer and the time when it performs another $\text{XCHG}$, an arbitrary number of other regular users might have $\text{XCHG}$ed the tail pointer and enqueued behind $R$ (lines 12–17). Therefore, when $R$ retries to acquire the lock (lines 4), it $\text{XCHGs}$ this latest lock tail $V$ in its list, instead of a reference to its own $\text{qnode}$.

The above procedure causes two atomic operations, yet $R$ does not acquire the lock. One optional optimization for reducing memory traffic, again, is to spin-wait with a backoff until the lock tail becomes non-$\pi$. We note that this optimization should be triggered after at least one iteration of $\text{XCHG}$. We empirically observed that, when we read the lock tail for this purpose before the initial $\text{XCHG}$, it causes more traffic on the contended cacheline and slows down the most important use case: no or few guests.

The protocols shown in Algorithm 16 could employ many optimizations. For example, an exponential backoff strategy with TATAS-style spin-and-retry can be used to reduce memory traffic: at line 10 of Algorithm 16, the requesting thread jumps back to line 4 to retry the $\text{XCHG}$ operation only if it finds the lock tail does not contain $\pi$. These are obvious optimizations that is orthogonal to the design of MCSg, so we omit them in Algorithm 16 and other algorithms to follow in the rest of this Chapter.

Note that when retrying to acquire the lock, a regular user is not guaranteed to maintain its original place in the wait queue; another regular user could conduct the $\text{XCHG}$ at line 4 faster and get a return value of $\text{NULL}$, violating FIFO order among groups. We discuss and address this issue in Section 6.4.

6.3.3 Key Properties of MCSg

Before moving on to the extended version of MCSg, let us analyze the basic MCSg algorithm regarding the desiderata listed in Section 6.2.

Assuming the frequency of guest users entering the lock is negligibly low, MCSg preserves scalability benefits for regular users brought by the original MCS lock: scalable doorway, local spinning, and FIFO ordering.

For guests, MCSg behaves similarly to a TATAS lock. We rely on a sentinel value $\pi$ stored in the lock tail to indicate that the lock is held by a guest. Guests’ lock acquisitions and releases succeed if the $\text{CAS}$ successfully changes the lock tail from $\text{NULL}$ to $\pi$ and from $\pi$ to $\text{NULL}$ respectively. Regular users whose $\text{XCHG}$ against the lock tail returns the sentinel $\pi$ are responsible for swapping $\pi$ back to the lock tail and then fall back to a spin-retry cycle. As a consequence, guests do not need any context to join and leave the lock, satisfying the context-freeness.

MCSg does not change anything on the MCS’s lock-state data structure. Therefore, it also maintains the single-word lock state property, the inter-process property, and the compose-ability. It does not pose any new requirement on the environment, either. In sum, MCSg is an ideal “drop-in” replacement for MCS locks with minimal changes. In fact, we have replaced MCS locks with MCSg locks in FOEDOS,
an open source database system [90], replacing its MCS code with just \( \sim 20 \) LoC changes. Section 6.5.3 evaluates the performance of the MCS\( g \) lock in FOEDUS.

### 6.4 MCS\( g++ \) Extensions

MCS\( g \) satisfies the aforementioned scalability and simplicity requirements for regular users. MCS\( g \), however, can potentially starve guest users and violate FIFO order among regular users.

#### 6.4.1 Issues with MCS\( g \)

**Guest Starvation**

MCS\( g \) retries a \texttt{CAS} for guests to acquire and release the lock. Although the steps are straightforward, a guest might starve by repeatedly failing the \texttt{CAS} when competing with regular users and other guests. It is possible that a steady stream of regular users lock out all guests forever. This is due to an inherent limitation of \texttt{CAS}: there is no guarantee that \texttt{CAS} will succeed in a bounded number of steps, violating MCS’s wait-freeness of its doorway.

**Non-FIFO Behaviors among Regular Users**

Figure 6.2 pictorially represents how non-FIFO ordering ensues in the presence of guests. For clarity, we follow the TATAS-style spin-and-retry optimization described in Section 6.3. In this example, a guest user initially holds the lock (a). Another thread \( T1 \) tries to acquire the lock (b). Yet another thread \( T2 \) trying to acquire the lock will enqueue itself behind \( T1 \) because \( T2 \) observed the lock tail was neither \texttt{NULL} nor \( \pi \) (c). \( T1 \)’s \texttt{XCHG}, executed at line 9 of Algorithm 16, will return a lock tail pointing to \( T2 \) because \( T2 \)’s \texttt{XCHG} at line 4 happened before \( T1 \)’s at line 9. As shown in Figure 6.2(d), \( T1 \) will start to spin on the lock tail and retry after the guest has released the lock. Meanwhile, \( T2 \) has set \( T1 \)’s \texttt{next} field to point to its \texttt{qnode}.

In the meantime, another two regular users—\( T3 \) and \( T4 \)—attempt to acquire the lock while \( T1 \) is spinning on the lock tail. \( T3 \) and \( T4 \) will go through the same steps as \( T1 \) and \( T2 \) did, as shown in (e). This might result in the intermediate state shown in (f). After \( T3 \) realizes that a guest is holding the lock, it will also issue an \texttt{XCHG} and start spinning on the lock tail. As a result, there could be multiple threads spinning on the lock tail. Each thread spinning on the lock tail leads a \textit{group} of users trying to acquire the lock. We call such threads \textit{group leaders}. Formally, group leaders are the threads that found the lock tail’s previous value returned by the \texttt{XCHG} to be \( \pi \). \( T1 \) and \( T3 \) in (f) are two group leaders. \( T2 \) and \( T4 \) are not group leaders but two regular successors that spin on their own \texttt{flag} fields.

If the guest now released the lock, \( T1 \) might notice that the lock tail is pointing to \texttt{NULL}. \( T1 \) then executes another \texttt{XCHG} to retry acquiring the lock. When the guest releases the lock, the group leaders will notice that the lock tail has changed and retry the \texttt{XCHG} (line 4 of Algorithm 16), setting the lock tail to point to the tail of the group (obtained at line 9 of Algorithm 16).

If a later arriving group leader wins in installing its group tail, then FIFO ordering among groups of regular users is not guaranteed. In (g), \( T3 \)’s \texttt{XCHG} succeeded earlier than \( T1 \)’s. Group 1 (led by \( T1 \)) then queues up after Group 2 (led by \( T3 \)) by installing \( T1 \) in \( T4 \)’s \texttt{next} field.

In most cases, MCS\( g \) gives FIFO ordering if guest users are rare. However, this could potentially become an issue when there is an unexpected burst of guest users in a highly contended lock.
6.4.2 Guaranteed Guest Lock Acquisition

MCSg++, an enhanced variant of MCSg, addresses the above issues.

The first enhancement addresses the guest starvation issue. The key idea is that a guest user attaches itself to a regular user’s qnode in a scalable manner. In MCSg++, a guest follows a “declare-and-wait” paradigm to acquire the lock without using a self-provided qnode. This protocol is reminiscent of regular users and the CLH lock [38,112].

In MCSg++, a regular user followed by a guest is responsible for passing the lock to the “enqueued” guest in its release protocol. To release the lock, the guest atomically swaps the lock tail with NULL so that group leaders and other incoming users can resume to compete for the lock.
Algorithm 17 MCSg++ locking protocol for guests.

```python
def guest_acquire(lock_tail):
    retry:
        pred = XCHG(lock_tail, π)
    if pred == NULL:
        return
    elif pred == π:
        goto retry
    else: # The predecessor is a regular user
        pred->next = GuestWaiting
        # Wait for the predecessor to pass the lock
        spin_while pred->next != GuestGranted
        # Acknowledge the predecessor
        pred->next = GuestAcquired

def guest_release(lock_tail):
    tail = XCHG(lock_tail, NULL)
    if tail != π:
        tail->next = NoSuccessor
```

Handshakes between Guest and Regular Users

MCSg++ relaxes the meaning of π and the use of regular user’s qnode. In addition to the original meaning of π in MCSg (a guest has acquired the lock), in MCSg++ a π value in the lock tail could also mean that a guest is waiting for the lock.

MCSg++ introduces sentinel values to the qnode’s next field for communicating with guests. Therefore, the next field has a dual use of (1) holding a pointer to a regular successor and (2) serving as a communication channel between the regular user and its guest successor. MCS and MCSg only use the next field for (1).

The sentinel values that could appear in the next field are:

- GuestWaiting: The successor is a guest waiting for the lock;
- GuestGranted: The lock is granted to the guest successor;
- GuestAcquired: The guest successor has acquired the lock;
- NoSuccessor: No successor.

We next describe how guest and regular users interact with the lock.

Guest Lock Acquisition

The acquire protocol for guests under MCSg++ is reminiscent of that in the CLH lock [38, 112]. Instead of retrying CAS on a centralized memory location or spinning on its own qnode, a guest registers itself in the next field of its regular predecessor’s qnode by storing GuestWaiting there. The guest then spins on the next field and waits to be “woken up”. A regular user that has a guest successor must be responsible to set GuestGranted in its own next field to pass the lock to the guest.

Algorithm 17 (lines 1–14) gives details on how guests proceed to acquire the lock. Instead of retrying CAS, the guest issues an XCHG against the lock tail, setting it to π (line 3). A π value in the lock tail makes
other incoming users (regular or guest) aware of the existence of a guest. The return value \( \text{pred} \) of this \text{XCHG} identifies the predecessor: no predecessor (\text{NULL}), a guest (\( \pi \)), or a regular user. If \( \text{pred} \) points to \text{NULL}, then the lock is not contended and the guest now has successfully acquired the lock (line 5). If \( \text{pred} \) is \( \pi \), another guest has already acquired the lock or announced its intention to acquire the lock. In this case, the guest simply retries by going back to line 2.\(^2\) When the predecessor is a regular user, the acquiring guest needs to indicate its existence in the predecessor’s \text{qnode} instead of in the lock tail (line 9). The guest then spins on the next field in the predecessor’s \text{qnode} (line 12) until the predecessor changes its next field to \text{GuestGranted}. The guest then stores \text{GuestAcquired} in next to inform the predecessor that it has acquired the lock. This acknowledgment is necessary for the regular predecessor to safely reuse the \text{qnode} for future lock acquisitions.

**Guest Lock Release**

Releasing the lock as a guest is wait-free in MCSg++: the guest simply atomically swaps the lock tail with \text{NULL} (line 17 of Algorithm 17). This enables waiting group leaders and guests to compete for the lock again. If the return value of this \text{XCHG} is not \( \pi \) (i.e., the successor is a regular user), as shown by line 19, the guest “marks” the successor as the tail of the group with a sentinel value \text{NoSuccessor}. The regular user will observe \text{NoSuccessor} when releasing the lock. The next requester (regular user or guest) whose \text{XCHG} returns \text{NULL} will acquire the lock.

As we have discussed above, MCSg++ relies on the next field in the \text{qnode} to track guest status. In particular, it takes advantage of the following invariant:

**Invariant 1** If there is a waiting guest, the lock-holding regular user’s next field will eventually become non-NULL.

When a regular user fails the \text{CAS} for lock release in the original MCS lock, the next field of the regular user is guaranteed to become non-NULL eventually. Invariant 1 still holds in the presence of guests. In Algorithm 17, guests maintain this invariant by storing and reading sentinel values in next. We now show regular users’ protocols that rely on and also preserve the invariant.

**Regular User Lock Acquisition**

The lock acquisition protocol for regular users in MCSg++ differs from that in MCSg when the acquiring regular user notices that the lock tail contains \( \pi \). If the lock is not contended or the lock tail is pointing to a regular user, the user can acquire the lock in the same fashion as in MCS and MCSg (lines 5–6 and 23–26 of Algorithm 18).

When a guest is present (either having acquired or still waiting for the lock), the acquiring regular user will form a new group as more regular users come to acquire the lock (lines 7–22). Unlike in MCSg, a regular user in MCSg++ does not immediately put \( \pi \) back to the lock tail. Therefore, unless there are more guests coming to compete for the lock, incoming regular users will queue up after one another. The first regular user whose \( \text{pred} \) points to \( \pi \) will be the group leader. For a group leader, the gist of the algorithm is a loop in which it tries to reach the end of the group (represented by \text{group_tail}).

Based on Invariant 1, the group leader first waits for its successor field (next) to become non-NULL (line 12). \text{group_tail} initially points to \text{my_qnode}, the requester’s own \text{qnode}, as indicated by line

\(^2\) Similarly to the optimization in Section 6.3.2, one could reduce memory traffic with a backoff when the lock tail is obviously \( \pi \).
2. Depending on the status of other concurrent users, the current group tail’s next field could be NoSuccessor/GuestWaiting or point to another regular user. If it is pointing to a regular user, the group leader simply follows the pointer and enters the next iteration, jumping from line 22 to line 8.

The other two cases are results of interactions with guests. If the current group tail’s next field is NoSuccessor (line 13), it means a guest has released the lock and “notified” the acquiring group leader that there are no more successors in the current group as described in line 16 of Algorithm 17. As shown by lines 13–15 of Algorithm 18, the group leader can now lead the group to retry as if nothing had happened. The only difference is that the group leader will put the group tail—instead of its own—in the lock tail. This process is similar to how a group leader retries to acquire the lock on behalf of its members in MCSg. The difference is that a group leader in MCSg++ follows the next fields to find out the “real” group tail and guest status. MCSg could easily obtain this information from the return value of XCHG.
Chapter 6. MCS Locks that Welcome Guests

The case illustrated by lines 16–20 happens when another guest—other than the one noticed by the group leader earlier at line 4—tries to acquire the lock. Recall that a guest will register itself to the regular predecessor by storing \texttt{GuestWaiting} in the predecessor’s \texttt{next} field (line 9 of Algorithm 17). In this case, the acquiring group leader then needs to install π back to the lock tail and retry so that incoming regular users will form new groups (lines 17–20 and 3–4 of Algorithm 18). Concurrent guests in this case will have to retry until a regular user becomes its predecessor or the current guest that is holding the lock exits.

Regular User Lock Release

Similar to the original MCS, releasing an MCSg++ lock as a regular user also starts by attempting a \texttt{CAS}, expecting that the lock was not contended (line 29 of Algorithm 18). If this \texttt{CAS} fails, it means another guest or regular user has tried to acquire the lock: the releasing regular user is responsible for notifying the successor.

Relying on Invariant 1, the regular user first waits until the \texttt{next} field becomes non-NULL. If the successor is a regular user, it will register itself in \texttt{next}. If the successor is a guest, it stores \texttt{GuestWaiting} and waits for \texttt{GuestGranted} in the \texttt{next} field (lines 9–12 of Algorithm 17). Therefore, Invariant 1 always holds in the presence of guests. To pass the lock to a regular successor, the releasing regular user simply writes to the successor’s \texttt{flag} field as MCS and MCSg do. To pass the lock to a guest, the releasing regular user puts \texttt{GuestGranted} in \texttt{next}. When the guest detects the transition from \texttt{GuestWaiting} to \texttt{GuestGranted}, it will set \texttt{next} to \texttt{GuestAcquired}, acknowledging to the regular user that the guest has acquired the lock. Upon detecting \texttt{next} has become \texttt{GuestAcquired}, the regular user can now leave (lines 35–36 of Algorithm 18).

The above state transitions \texttt{GuestWaiting} → \texttt{GuestGranted} → \texttt{GuestAcquired} guarantee the integrity of the communication channel between the guest and its regular predecessor. Suppose that a guest was pre-empted after setting its predecessor’s \texttt{next} field to \texttt{GuestWaiting}. Suppose also that the predecessor has released the lock without waiting for the acknowledgment before the guest starts to spin at line 12 of Algorithm 17. If the regular user started another round of lock acquisition and release using the same \texttt{qnode}, the guest might be spinning indefinitely for \texttt{next} to become \texttt{GuestGranted}. Therefore, we need to ensure that the guest has picked up the lock before letting the regular user leave.

6.4.3 Reducing Non-FIFO Behaviors

In the presence of guests, an older regular user group might queue up after a younger one in MCSg. FIFO order is preserved for all users within the same group, but not always among groups. MCSg++ does not guarantee FIFO ordering between a regular user and a guest, either. Completely preserving a total order among both regular users and guests generally requires maintaining a list-like structure for them. This violates the simplicity principle for guests because guests need to be associated with some context as well. Because of the relative rareness of guest users, the chance of forming many groups is not high in our targeted use cases. We therefore focus on reducing the FIFO behavior among regular user groups.

Our solution is inspired by ticket locks. A ticket lock has two counters: \texttt{ticket-owner} and \texttt{next-ticket}. Threads atomically read and increment \texttt{next-ticket} to enter the lock. The thread whose ticket equals to \texttt{ticket-owner} can enter the critical section. Other threads will spin for \texttt{ticket-owner} to match their tickets.
Algorithm 19 MCSg++’s lock acquire protocol that reduces FIFO order violations among regular user groups. Additional logics are added on top of Algorithm 18, most of which are omitted for clarity.

```python
def regular_acquire(lock_tail, my_qnode):
    group_tail = my_qnode
    my_ticket = INVALID_TKT
    retry:
        if my_ticket != INVALID_TKT:
            spin_while(ticket_owner != my_ticket)  # acquire barrier
            pred = XCHG(lock_tail, group_tail)
            if pred == NULL:
                if my_ticket != INVALID_TKT:
                    FAA(ticket_owner, 1)
                    return
            elif pred == π:
                if my_ticket == INVALID_TKT:
                    my_ticket = FAA(next_ticket, 1)
                    ... lines 8 -- 22 of Algorithm 18 ...
            else:
                if my_ticket != INVALID_TKT:
                    FAA(ticket_owner, 1)
                    ... lines 24 -- 26 of Algorithm 18 ...
```

Similarly, MCSg++ maintains these two counters in addition to the lock tail. Whenever a regular user realizes it is a group leader, it obtains a ticket and waits for its turn before retrying. Algorithm 19 shows the revised lock acquire protocol for regular users. For clarity we omit most of the code that overlaps with Algorithm 18. At line 3 a user enters the lock without obtaining a ticket. It then conducts the XCHG operation at line 7. If the regular user finds that the return value is π (line 13), it is a group leader and will obtain a ticket by atomically reading and incrementing next_ticket. This is usually done through an atomic-fetch-and-add instruction. The group leader continues as illustrated by lines 8–22 of Algorithm 18. Before the group leader retries at line 7, it first spins on the ticket_owner to wait for its turn (lines 5–6 of Algorithm 19). Ticketing gives ordering to all group leaders competing for the lock in the presence of guests.

The ticket_owner field is incremented under two circumstances: (1) the group leader acquired the lock (lines 9–10) and (2) the group leader has to queue after another group (lines 17–18). In case (1), the group leader waited for its turn and the return value of the XCHG is NULL (line 8). Case 2 is the “unlucky” scenario where another regular user’s XCHG succeeded earlier while the group leader is waiting for its turn. This could happen if a regular user got the lock right after a guest released the lock (i.e., the regular user’s XCHG returned NULL) while the group leader was spinning on next_ticket. Therefore, ticketing only reduces FIFO order violations among regular user groups. We quantify the impact of case 2 in Section 6.5. In most cases, ticketing can reduce 50–70% of FIFO order violations.

6.4.4 Discussion

Under MCSg, guests might starve in the presence of a steady stream of regular users. MCSg++ uses XCHG to allow guest users to attach after regular users, which upon lock release will pass the lock to the awaiting guest. MCSg++ makes it easier for guests to acquire the lock. However, we also note that MCSg++ does not always guarantee a guest will be able to acquire the lock, especially when guests are the majority. Specifically, if there is a single guest and an arbitrary number of regular users, then the
guest will enter its doorway in bounded time. If most users are guests, however, or if there is a steady stream of guests intermixed with regular users, then an individual guest can starve.

6.5 Evaluation

This section empirically evaluates MCSg/MCSg++ and compares them with other candidates (described later) to confirm the following claims:

- MCSg maintains the same scalability of MCS when guests are rare (Section 6.5.2);
- MCSg admits guests yet preserves all the good properties of MCS as a drop-in replacement (Section 6.5.3);
- MCSg++ gives more fairness to guests and can reduce FIFO order violations (Section 6.5.4).

6.5.1 Setup

We conducted experiments on a server equipped with 16 Xeon E7-4890 processors clocked at 2.8 GHz, each of which has 15 cores. The server has 240 physical cores and 12 TB of DDR3 DRAM clocked at 1333 MHz. The processor has 256 KB of L2 cache per core and 38 MB of L3 cache per socket. We use a microbenchmark and a database workload for our experiments.

All threads are pinned to physical cores in a compact manner, meaning we assign threads to a minimal number of sockets. We always leave an unused core per socket so that watchdog and other kernel tasks do not hinder our threads. We do not use hyperthreaded hardware contexts to maximize the performance under high contention. We thus use at most 224 cores over 16 sockets.

Lock Algorithms in Microbenchmarks

We have implemented MCSg and MCSg++ in addition to MCS, CLH [38, 112], cohort lock [43], and a few more variants of MCS for comparison. A TATAS lock is implemented as a baseline. Below we describe the experimental setup for these lock algorithms; a more detailed explanation on these algorithms themselves can be found in Section 8.3. To measure the overhead and effectiveness of MCSg++’s ticketing machinery for preserving FIFO ordering among regular users, we have also implemented MCSg+, a stripped-down version of MCSg++ without the ticketing machinery. To show that MCSg can be used as a drop-in replacement of MCS everywhere, we implemented a cohort lock, C-MCSg-MCS, in which we use an MCSg lock in place of the global MCS lock. The C-MCSg-MCS lock is described in detail in Section 6.5.3.

The qnodes used by MCS, CLH, and C-MCSg-MCS are pre-allocated. Regular users in MCSg and MCSg++ also use pre-allocated qnodes. These pre-allocated qnodes are organized as a global array, with an entry for each thread. For CLH, we use the algorithm described in [152]. Because a successor in CLH spins on its predecessor’s qnode, a qnode cannot be reused until the successor notices the next field has been changed by its predecessor. In this implementation, each thread inherits its predecessor’s qnode to solve this problem.

Our implementation of the K42 operating system’s MCS variant (K42-MCS) follows the algorithm described in [152], which allocates qnodes on the stack. We also compare with our own extension of K42-MCS to analyze its performance without the restriction on inter-process use described in Section 6.2.
Instead of using a stack-allocated qnodes, our K42-MCS-TLS maintains a global qnode pool from which a thread can atomically borrow and return a qnode. To avoid the cost of borrowing and returning a qnode each time, K42-MCS-TLS uses a thread-local (TLS) variable to hold the borrowed qnode for each thread. This introduces another requirement on efficient TLS support in the platform, but this experiment is run on a CPU that satisfies the requirement (x86_64, on which Linux can use the %fs register for TLS).

Likewise, our CLH implementation uses a global array of pre-allocated qnodes for inter-process use without stack variables. To enable guest access, we implement CLH-TLS, a variant of CLH with standard interface based on the proposal in Section 4.3.2 of [152]. We use a TLS variable to store thread.qnode.ptrs[self] as [152] recommends.

Each thread in the critical section reads four cachelines and then releases the lock. We run each experiment for ten seconds and report the averages of five runs. For experiments that involve guest users, we run guest users as frequently as regular users to stress-test lock implementation. Note that this setting deviates from our design assumption that guest users are infrequent. Nevertheless, it is helpful for stress-testing MCSg and other variants.

Database Workload

MCS is used in various complex systems for its performance and simplicity. For example, modern database systems designed for massively parallel hardware, such as Shore-MT [76] and FOEDUS [90]. In such systems, there often are infrequent yet various background tasks in addition to regular worker threads that run database transactions.

We have implemented MCSg in FOEDUS for its superior performance on manycore systems. The guests are background threads that need to hold a page lock when installing snapshots. MCSg is applied to FOEDUS’s existing codebase. We run the Payment transaction in TPC-C [167], which is a write-heavy transaction that updates the customer’s balance and generates relevant statistics about the warehouse. The workload is run with 192 worker threads. The database size is set to one warehouse to generate enough contention. We compare its throughput (in million transactions per second, or MTPS) among MCS, MCSg, and TATAS (supposing FOEDUS had to use a centralized lock to allow guests).

6.5.2 Maintaining MCS’s Scalability

In this section, we focus on two basic settings: no guests and one guest. MCSg and MCSg++ should preserve the scalability of the original MCS when there are few guests. In both settings, we evaluate the locks under high contention: threads repeat the acquire–access–release cycle without any delays.

No Guests

Although the sole purpose behind MCSg and MCSg++ is to allow guests, it is crucial to maintain the performance in existing code paths for MCSg and MCSg++ to be superior drop-in replacements of the original MCS lock.

Figure 6.3 shows the throughput of each lock implementation with a varying number of regular users. Both MCSg and MCSg++ match MCS’s throughput. K42-MCS and CLH exhibit less scalability than MCS/MCSg. Although K42-MCS can handle guests, its use of CAS rather than XCHG causes many

\[3\] Available at https://github.com/hewlettpackard/foedus_code.
retries under high contention, making it less scalable than MCS/MCSg. K42-MCS sacrifices some scalability due to its use of CAS; CLH’s node inheritance design puts more pressure on the interconnect, making it less scalable than MCS.

Figure 6.4 shows the result. The throughput of MCSg is statistically equivalent to that of MCS. MCSg++ has a slight slowdown due to the additional ticketing mechanism, but still matches the original MCS lock. In line with the results shown in Figure 6.3, K42-MCS and CLH-TLS do not scale as well as other MCS variants. K42-MCS-TLS again closely follows the performance of K42-MCS. TATAS, not surprisingly, does not scale at all. We describe how the locks perform with more guests in Section 6.5.4.

With One Guest

We now repeat the same experiment with one guest and 223 regular users. Locks that cannot support guests (MCS and CLH) are excluded from this experiment, but we still include the throughput of MCS and CLH with 224 regular users for reference. Figure 6.4 shows the result. The throughput of MCSg is statistically equivalent to that of MCS. MCSg++ has a slight slowdown due to the additional ticketing mechanism, but still matches the original MCS lock. In line with the results shown in Figure 6.3, K42-MCS and CLH-TLS do not scale as well as other MCS variants. K42-MCS-TLS again closely follows the performance of K42-MCS. TATAS, not surprisingly, does not scale at all. We describe how the locks perform with more guests in Section 6.5.4.
Figure 6.4: Lock throughput with 223 regular users and one guest for locks that support guests. The numbers for MCS and CLH with 224 regular users are used as a baseline.

Table 6.1: Throughput of TPC-C’s Payment transaction under high contention (one warehouse, 192 threads). MCSg achieves the same performance as MCS yet provides a standard interface for guests.

<table>
<thead>
<tr>
<th>Lock</th>
<th>Throughput (MTPS)</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TATAS</td>
<td>0.33</td>
<td>±0.095</td>
</tr>
<tr>
<td>MCS</td>
<td>0.46</td>
<td>±0.011</td>
</tr>
<tr>
<td>MCSg</td>
<td>0.45</td>
<td>±0.004</td>
</tr>
</tbody>
</table>

6.5.3 MCSg as a MCS’s Drop-in Replacement for MCS

MCSg provides both the context-free and non-standard MCS interfaces for easier adoption. This section demonstrates that MCSg can be used as a drop-in replacement of MCS in the context of databases and cohort-locks.

MCSg in the FOEDUS Database System

As Section 6.5.1 described, we implemented MCSg in FOEDUS and ran TPC-C. Table 6.1 shows the throughput of TPC-C’s Payment transaction under high contention with one warehouse and 192 worker threads. We observe that MCS and MCSg improve end-to-end performance by up to 50% compared to TATAS. MCSg has performance equivalent to MCS yet allows guest users. Although the 50% difference is not as dramatic as the orders of magnitude differences seen in the microbenchmarks, it is a surprisingly significant improvement considering that database transactions are substantially more complex than locking itself. The complexity is also the source of much higher variance in this experiment. Especially with TATAS, the progress of transactions is highly random, it sometimes adds milliseconds of latency to a single transaction that usually finishes in sub-microseconds.

Because MCSg does not change the data structure and interface of the original MCS, the change required to adopt MCSg in FOEDUS was minimal—as few as 20 LoC—showing MCSg’s pluggability. Keeping the same data structure and interface is vital. Like many other complex systems, FOEDUS needs to embed many lock objects in each fix-sized data page (usually of 4 KB). Combining MCS with other locks (e.g., to compose a cohort lock) or adopting locks with a different data structure/interface incurs much more space overhead and code complexity compared to MCS and MCSg, which occupy only a single word.

Although K42-MCS provides a standard lock interface, it requires the ability to point to each thread’s stack memory. Its use is therefore limited to intra-process synchronization. Like many other databases,
Table 6.2: Microbenchmark comparison of MCS, MCSg, MCSg++ and C-MCSg-MCS with 240 regular users (no guests).

<table>
<thead>
<tr>
<th>Lock</th>
<th>Throughput</th>
<th>Average latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCS</td>
<td>1.77M/s</td>
<td>126.21µs</td>
</tr>
<tr>
<td>MCSg</td>
<td>1.86M/s</td>
<td>120.34µs</td>
</tr>
<tr>
<td>MCSg++</td>
<td>1.72M/s</td>
<td>129.58µs</td>
</tr>
<tr>
<td>C-MCSg-MCS</td>
<td>5.22M/s</td>
<td>42.79µs</td>
</tr>
</tbody>
</table>

FOEDUS uses shared-memory for most data objects to be referenced across processes. K42-MCS cannot be implemented in such systems because the pointer must be valid in all processes. One can extend K42-MCS for multi-process use with TLS, such as our K42-MCS-TLS. However, as we observed in previous experiments, it incurs another requirement on efficient TLS support, an overhead to access TLS variables, and complexity of a global qnode pool.

MCSg satisfies all the requirements and works as a pure drop-in replacement for MCS without any special hardware or additional complexity.

**MCSg in a Cohort Lock**

MCS is an important building block for more advanced/complex locks. Cohort locking [43] is such a composite lock implementation. For example, the C-MCS-MCS lock uses an MCS lock as the global lock and another for each NUMA node to get better scalability. In order to show that MCSg can be used as a drop-in replacement anywhere MCS is useful, we used MCSg to compose a cohort lock, C-MCSg-MCS, which uses MCSg as the global lock, and the original MCS as local locks. Guests contend directly on the global lock. Regular users first try to acquire the local MCS lock. The first winning regular user will then compete for the global lock and pass it to its successors for a “threshold” number of times. In our experiments, we set the threshold to 64, which is the recommended value in [43].

Table 6.2 shows the throughput in the microbenchmark with 224 regular users. C-MCSg-MCS outperforms MCSg and MCSg++ by ~3x, which coincides with the findings of [43]. Plugging MCSg in place of MCS is trivial. Composing C-MCSg-MCS thus did not require any more effort than C-MCS-MCS. In large NUMA systems with the need to support guests, C-MCSg-MCS is a preferable lock implementation. MCSg can also provide equivalent enhancement to other flavors of cohort locks and MCS-based hierarchical locks.

### 6.5.4 Getting More Fairness with MCSg++

We have discussed the performance of regular users in previous sections. This section focuses on guests and their interactions with regular users.

**Guest Starvation**

Recall that MCSg might starve guests because guests rely on CAS to acquire and release the lock. This will become a serious problem when the lock is highly contended. To solve this problem, MCSg++ uses XCHG for guests to acquire and release the lock, with extra handshake protocols between regular users and guests. Figure 6.5 compares the average latency using locks that support guests for a single guest to
Figure 6.5: Average latency for a guest to acquire the lock when running at 223 regular users and one guest. MCSg starves guests, while MCSg+ and MCSg++ significantly improves guests’ fairness.

acquire the lock with 223 regular users and one guest. The y-axis is in log scale. MCS and CLH are not included in the figure because it does not support guests.

As shown in the figure, when the lock is heavily contended, guests starve under MCSg because \texttt{CAS} does not have any bounded guarantee on when a guest can get the lock. In almost all runs, the guest under MCSg acquires the lock only for a few times. K42-MCS supports guests by giving them an implicit \texttt{qnode} living on the stack, which gives guests equal opportunity as regular users. However, K42-MCS could issue many \texttt{CAS}es during lock acquire and release, resulting in high latency for both guests and regular users. We observed the exact same behavior in K42-MCS-TLS. MCSg++ and MCSg+ achieve orders of magnitude lower average guest latency because they issue a single \texttt{XCHG} instead of \texttt{CAS} in most operations. Moreover, the only guest is the absolute minority among all users in this experiment. Therefore, under MCSg+ and MCSg++ the guest will be able to easily find a regular predecessor to attach to and grab the lock thereafter with low latency.

Figure 6.6 shows the total throughput with a varying number of guests. The total number of threads is fixed at 224. Among the locks we tested, K42-MCS/K42-MCS-TLS and CLH/CLH-TLS give fair scheduling for both guests and regular users. Therefore, they all maintained steady performance across the x-axis in Figure 6.6. MCSg behaves similarly because it favors regular users much more and starves guests. Correspondingly, Figure 6.7 shows that MCSg does not perform well for guests. MCSg+ and MCSg++ both give much more fairness to guests, but at the cost of lower throughput for regular users when there are many guests, as shown by Figure 6.8. This is because of the complexity to handle handshakes between guest and regular users, which MCSg does not have.

FIFO Ordering

In order to evaluate the effectiveness of MCSg++’s ticketing mechanism, we compare the number of FIFO order violations among regular users that happen during 10-second runs with a varying number of guests. Since MCSg often starves guests under high contention, we focus on comparing MCSg+ and MCSg++ in this experiment. We use a TLS counter to record the number of FIFO order violations at runtime, and sum up all the counters after each experiment. As shown in Algorithm 19, a group leader can exit the retry loop only in two cases: (1) its \texttt{XCHG} returned \texttt{NULL} (lines 5–6), or (2) it successfully attached after another regular user (lines 24–27). Therefore, combined with Algorithm 18, we increment the TLS counter whenever a group leader queues after another regular user with a valid ticket (lines 24–27 of
Figure 6.6: Total throughput when running at a varying number of guests (224 users in total). MCSg favors regular users and starves guests, thus providing stable performance. K42-MCS and K42-MCS-TLS also show stable performance because they treat all users equally. MCSg+ and MCSg++ sacrifice total throughput to give guests fairness.

Figure 6.7: Guest throughput when running at a varying number of guests with a total number of users of 224. MCSg++ performs consistently better than MCSg+, because ticketing causes more bias toward guests; regular users are “throttled” by waiting for turns.

Algorithm 18). We added the ticketing machinery in MCSg+ for counting FIFO order violations only. Regular users acquire tickets like they do with MCSg++, but do not use them. In other experiments, MCSg+ does not implement the ticketing machinery at all.

Figure 6.9 shows the result. Compared to MCSg+, MCSg++ can reduce FIFO order violations by up to 70%. Because a group leader in MCSg++ waits for its turn via the ticketing machinery, guests have a higher chance to acquire the lock while group leaders are waiting for their turns. Figure 6.7 verifies that ticketing favors guests. MCSg++ consistently achieves much higher guest throughput than MCSg+. 
Figure 6.8: Regular user throughput when running at a varying number of guests. The number of total users is fixed to 224. The numbers for MCSg+ and MCSg++ drops faster than K42-MCS and CLH families due to their bias toward guests. Compared to MCSg+, MCSg++’s is more biased toward guests because with ticketing, regular users are “throttled” by waiting for turns.

Figure 6.9: The number of non-FIFO behaviors with 224 users in total. Ticketing in MCSg++ can reduce up to 70% of non-FIFO behaviors among regular user groups.

Finally, we note that ticketing trades off regular user throughput for guest performance. Figure 6.8 shows that, as the number of guests increases, the regular user throughput of MCSg++ drops faster than that of MCSg+. With more guests, regular users have a higher chance to be “chopped” and form more groups to acquire and wait on their tickets, causing a higher contention.

6.6 Summary and Discussion

We have described a new variant of MCS locks, which allows lock acquisition and release without any bring-your-own-context and without degrading the high scalability of MCS locks. The key observation
behind this work is that complex multi-thread/multi-process software often has two kinds of lock users: regular users and guest users.

MCSg behaves as an MCS lock to regular users and as a centralized lock to guest users, providing benefits of both. We recommend using MCSg as a drop-in replacement for existing locks in three scenarios.

The first scenario is to replace an existing MCS lock that needs to allow guest users. As we have observed in Section 6.5.3, it requires only a small change to transform an existing MCS lock to MCSg.

The second scenario is to replace an existing centralized spinlock (e.g., TATAS) that is a scalability bottleneck. It is trivial for the developer to replace the lock with an MCSg lock where all existing lock users (i.e., functions) are guests. Then, the developer can gradually identify the few most frequent lock users and modify them to be regular users with MCS queue nodes. While the lock will enjoy high scalability as an MCS lock, the majority of lock users can still remain as guests without any code change.

The third scenario is to use MCSg as a building block for combined locks, such as cohort locking. MCSg keeps the simplicity and pluggability of MCS locks, hence it can be used wherever MCS locks could be used. For example, C-MCSg-MCS instead of C-MCS-MCS and C-BO-MCSg++ instead of C-BO-MCS can provide the guest user functionality in addition to the high scalability of the original cohort locks.

Finally, we have also proposed an extended version of MCSg, MCSg++. MCSg++ provides guest users with a guaranteed lock acquisition on highly contended locks. MCSg++ also reduces non-FIFO behaviors between groups of regular users at the cost of less simplicity (e.g., no longer a single word). We recommend developers to start with MCSg because of its simplicity and perfect compatibility to the original MCS lock. As frequent code paths are upgraded to regular users, MCSg rarely poses any issue.

When it is somehow difficult to upgrade a frequent code path to be a regular user (e.g., an untouchable code path in a complex legacy codebase), we recommend MCSg++ to ameliorate the issues.

* * *

This chapter concludes our discussion on single-node systems, from removing the logging bottleneck to devising robust concurrency control protocols, and better synchronization primitives for today’s high-end and future hardware. These solutions focus on “scaling-up” with modern and future hardware. Real applications also require high availability: if a primary database server is down, a secondary server should be able to take over and continue servicing client requests without any data loss or long transition time. The high speed brought by modern main-memory databases, however, becomes a double-edged sword in such high availability scenarios. On the one hand, it enables fast transaction processing; on the other hand, it creates a “freshness gap” between the primary and backup servers, where the backups lag behind the primary and cannot provide safety guarantees or data freshness. The next (and last) chapter of this thesis focuses on solving this problem, by proposing Query Fresh, a log shipping system that utilizes new hardware features (RDMA over NVRAM and fast networks such as InfiniBand) to provide both strong safety and data freshness.
Chapter 7

Fast and Fresh Hot Standbys

Log shipping is widely used to build hot standby systems. However, hot standby systems often have to trade safety (i.e., not losing committed work) and freshness (i.e., have access to recent updates) for performance. Providing safety requires synchronous log shipping that blocks the primary until the log records are durably replicated in one or multiple backups; maintaining freshness necessitates fast log replay on backups, but is often defeated by the dual-copy architecture and serial replay: a backup must generate the “real” data from the log to make recent updates accessible to read-only queries.

This chapter proposes Query Fresh, a hot standby system that guarantees both safety and freshness without sacrificing the primary’s high performance. The crux of Query Fresh is an append-only storage architecture used in conjunction with fast networks (e.g., InfiniBand) and byte-addressable, non-volatile memory (NVRAM). Query Fresh avoids the dual-copy design and treats the log as the database, enabling lightweight, parallel log replay that does not block the primary server. Experimental results show that backup servers can replay log records faster than they are generated by the primary server, using one quarter of the available compute resources. With a 56Gbps network, Query Fresh is able to keep up to 4–5 synchronous replicas, each of which receives and replays more than 1.4GB of log records per second, while maintaining high performance on the primary with 4–6% overhead over a standalone server that achieves 620kTPS on the TPC-C benchmark. 1

7.1 Introduction

Hot standby is a popular approach to high availability and is employed by many production database systems [67,133,134,144,161]. A typical hot standby system consists of a single primary server (“the primary”) that processes read/write transactions, and one or more backup servers (“backups” or “secondaries”). The primary periodically ships log records to backups, which continuously replay log records. Log shipping can be configured as synchronous to guarantee strong safety [61]: transactions are not committed until their log records are persisted in all (or a majority of) nodes. Synchronous log shipping ensures that if the primary fails, a backup can take over as the new primary without losing committed work. In addition to replaying log records, backups can serve read-only transactions, improving resource utilization and performance.

1 This chapter is based on materials presented at HPTS 2015 [177] and a full paper that appeared in VLDB 2018 [178].
7.1.1 The Freshness Gap

The ability to serve read-only transactions on backups while maintaining strong safety guarantees is a salient feature of log shipping: modern database servers are high-end machines that constitute non-trivial parts of the total cost of ownership, and offloading read-only transactions to backups can significantly increase hardware utilization and read-only throughput. Despite their popularity and usefulness, existing hot standby solutions often exhibit stale data access: queries on backups can only use a potentially much earlier snapshot of the primary database, as updates and new additions of data are not reflected on backups very quickly [184].

The reason for the freshness gap is two-fold. First, without logical log shipping and deterministic execution [115, 164, 185], the primary must continuously transfer large amounts of log records to backups, demanding high network bandwidth that traditional network often lacks. This is particularly true for modern main-memory database engines [42, 85, 89, 90, 169] that can easily produce 10–20Gb of log data per second (see details in Section 7.4). Such log rate is well beyond the bandwidth of ordinary 10Gbps networks. To support strong safety, log shipping must be synchronous, and the primary must wait for acknowledgement from backups on persisting log records. This puts network and storage I/O on the critical path, making the primary I/O bound.

However, the more important, fundamental reason is inefficient log replay in existing systems. These systems rely on the dual-copy architecture and permanently store data twice: in the log and the “real” database. Log records must be fully replayed before new updates can be queried in backups, involving non-trivial data movements and index manipulations. Moreover, in many systems log replay is single-threaded [134, 184], although it is commonplace for the primary to generate log records with multiple threads concurrently. It is unlikely for backups to easily catch up with the primary and provide fresh data access to read-only transactions.

7.1.2 Query Fresh

This chapter proposes Query Fresh, a hot standby solution that provides both safety guarantees and fresh data access on backups while maintaining high performance. The key to realizing this is an append-only storage architecture that is built on modern hardware and allows fast log data transfer and lightweight, parallel log replay.

Modern hardware. Query Fresh utilizes high-speed networks (such as InfiniBand) and byte-addressable, non-volatile memory (NVRAM) to alleviate the staleness caused by slow network and storage I/O. Recent fast networks provide high bandwidth and low latency, and allow fast remote direct memory access (RDMA). Coupled with fast NVRAM devices such as Intel 3D XPoint [39] and NVDIMMs [2, 171], the primary ships log records directly from its log buffer to NVRAM-backed log buffers in backups. Log records are instantly persisted once they reach NVRAM, without explicit access to the (expensive) storage stack. Moreover, Query Fresh takes advantage of RDMA’s asynchronous nature to hide the latency of data transfer and persistence behind forward processing, moving I/O out of the critical path with simpler implementation. Log data can be gradually de-staged in background to other bulk storage devices such as flash memory and disks when the log buffer is full.

Lightweight, parallel log replay. Fast network and NVRAM do not mitigate staleness due to inefficient log replay, caused by the traditional dual-copy architecture. Instead of maintaining two durable copies, Query Fresh employs append-only storage and keeps the log as the only durable copy of data,
i.e., the log is the database [89]. This is made possible by redo-only logging: the log only contains actual data generated by committed transactions. Redo-only logging is common in modern main-memory database engines [89, 90, 169] and we design Query Fresh based on it. With the log as the database, worker threads access data through indirection arrays [34, 103, 151] where each entry maps a (logical) record identifier (RID) to the record’s physical location (in memory or storage, i.e., the log). Indirection arrays are purely in-memory, but can be checkpointed for faster recovery. Indexes map keys to RIDs, instead of physical pointers. Updates can be reflected in the database by simply updating indirection arrays, without updating indexes.

The combination of append-only storage and indirection allows us to build a lightweight, parallel replay scheme without excessive data copying or index manipulations. Replay becomes as simple as scanning the log buffer, and setting up the indirection arrays. This process is inexpensive and faster than forward processing on the primary, guaranteeing backups can catch up with the primary without exhausting all compute resources. This improves freshness and frees up more resources to run read-only transactions on backups.

Evaluation results using TPC-C [167] show that with 56Gbps InfiniBand, before the network is saturated, Query Fresh can support up to 4–5 synchronous backups, each of which receives \( \sim 1.4 \text{GB} \) of log data from the primary, whose throughput does not drop by more than 6% over a standalone server without replication (620kTPS). Our evaluation also shows that backups are able to finish replaying 16MB of log records in \( \sim 12 \text{ms} \) using around one quarter of the compute resources, leaving more resources for read-only transactions.

7.1.3 Contributions and Chapter Organization

Our first contribution is an efficient log shipping scheme that takes advantage of NVRAM and the asynchronous nature of RDMA to easily keep network I/O and replay out of the critical path. We also highlight caveats for using RDMA over NVRAM for safe persistence. Second, we propose to utilize append-only storage for log shipping and replay. Third, we devise a lightweight, parallel replay scheme for backups to keep up with the primary with fewer resources than the primary needs for forward processing.

The ideas of append-only storage and indirection have been used in other systems [11, 19, 34, 89, 100, 103, 151]. Our focus is utilizing them to speed up log replay and improve freshness.

Next, we cover background in Section 7.2. Section 7.3 describes the design of Query Fresh, followed by evaluation in Section 7.4. We summarize in Section 7.5.

7.2 Background

In this section, we begin by defining the type of systems we focus on. Then we introduce the basics of fast networks, RDMA, and discuss issues related to using RDMA over NVRAM.

7.2.1 System Model and Assumptions

We focus on hot standby systems based on log shipping. As described in Section 2.1.4 on page 15, in such a system a single primary server processes both reads and writes, and backup servers process log records shipped from the primary and read-only transactions; a backup can also take over as the new primary when the current primary fails. A hot standby solution can provide strong safety using synchronous log
shipping, which is the focus of this Chapter. We follow the extended definition “strong safety” based on 2-safe for multi-node systems in Section 2.1.4 on page 15. The design of Query Fresh is orthogonal to, and can be applied on top of, both traditional 2-safe and quorum-based systems [37, 170]. In this chapter, we tackle the cases where a transaction is declared committed only when its log records have been persisted across all nodes.

7.2.2 RDMA over Fast Networks

RDMA allows nodes in a cluster to access each other’s designated memory areas, without having to go through multiple layers in the OS kernel, avoiding unnecessary data copying and context switches between the user and kernel spaces which are normally unavoidable using the TCP/IP stack. RDMA is also non-blocking: posting an RDMA operation does not block the caller. The caller instead should explicitly check for completion of the posted work requests, e.g., by polling the completion queue.

RDMA exhibits a “verbs” interface [117] that is completely different from TCP/IP sockets. Peers communicate with each other by posting verbs (work requests) to queue pairs (similar to sockets in TCP/IP). A queue pair consists of a send and a receive work queue. There are two types of verbs: channel semantic (SEND and RECV) and memory semantic (READ, WRITE, and atomics). Channel semantic verbs need coordination with the remote CPU (a RECV must be posted for each remote SEND). Memory semantic verbs operate directly on remote memory without involving the remote CPU. In addition, RDMA Write with Immediate allows the sender to accompany the payload with an immediate value (e.g., to carry metadata). The only difference compared to RDMA Write is that the receiving end should post a receive request to obtain the immediate value [117]. Query Fresh uses RDMA Write and RDMA Write Immediate for log shipping.

7.2.3 RDMA over NVRAM

NVRAM can be attached to the memory bus and accessed remotely through RDMA. However, a completed RDMA write request does not indicate that data is written in remote NVRAM. Rather, it only guarantees that the data has arrived at the remote side, although the data might be cached by the remote NIC or CPU, instead of being stored in NVRAM due to techniques such as Intel Data Direct I/O (DDIO) [71]. As a result, data visibility and durability are decoupled: data is visible on the remote side once it is out of the I/O controller and arrives at the CPU cache; data will only get written to memory when it is evicted from CPU cache [71].

There are two stop-gap solutions, the “appliance” and “general-purpose server” methods [44]. The former requires turning off DDIO and enabling non-allocating writes in the BIOS. Each RDMA Write should be followed by an RDMA Read for the remote node to force data to NVRAM. The use of non-allocating writes and RDMA Read adds additional overheads. The remote node needs no additional operation. The required BIOS changes make it less ideal in environments such as the cloud. The general-purpose server method requires no BIOS change, but the remote side needs to explicitly issue the CLFLUSH or CLWFB instruction followed by a store fence for persistence and correct ordering [72].

The general-purpose server method can incur 50% higher latency than the appliance method [44]. Both methods need at least two round trips for data durability. Simply pushing data to the remote site does not guarantee persistence [192]. Nevertheless, the general-purpose server method should give us worst-case performance; we quantify its overhead in Section 7.4. We expect RDMA protocol changes for
NVRAM [159] to be the ultimate solution.

7.3 Query Fresh

Query Fresh consists of a synchronous log shipping mechanism that exploits RDMA and NVRAM, an append-only storage architecture with indirection designed for log shipping, and a parallel, lightweight log replay scheme that utilizes append-only storage to allow backups to keep up with the primary. Like other hot standby systems, the primary server in Query Fresh executes read/write transactions and ships log records to backups when needed. It uses a background daemon to listen to new backups requesting to join the cluster and sets up a connection for each backup server. During connection setup, the daemon registers the log buffers in both the primary and backups to conduct RDMA. Memory registration is a one-time operation that does not affect forward processing.

Several database systems employ group or pipelined commit [77] to keep I/O out of the critical path and issue large sequential writes to storage for better performance. In these systems, worker threads submit processed transactions to a commit daemon and continue to serve the next request. A transaction’s commit result is not returned to the client until its log records are persisted. Log flushes are typically issued by request, timeout, or when the log buffer’s occupied space exceeds a threshold (e.g., 50% full). Query Fresh piggybacks on the commit daemon and ships log records whenever a log flush or group commit happens. This way, we take advantage of RDMA’s asynchronous nature to overlap network and local I/O, as well as forward processing, hiding most of the cost of data transfer.

Query Fresh uses two key techniques to accelerate log replay on backups: (1) replay pipelining that moves log replay (mostly) out of the critical path and (2) a parallel, lightweight scheme that uses our append-only storage and indirection, to reduce the amount of work done by replay threads, making log replay itself faster.

In the rest of this section, we first describe how Query Fresh utilizes RDMA and NVRAM for log shipping. Then we cover how replay pipelining works. We then introduce the append-only storage architecture designed for log shipping and fast log replay, followed by a comparison with other popular approaches.

7.3.1 RDMA and NVRAM based Log Shipping

In Query Fresh, log shipping is synchronous and largely a memory-only operation that does not involve the storage stack, thanks to RDMA and NVRAM. For backups, we place the log buffer in NVRAM and register it to perform RDMA operations. The primary, however, does not have to be equipped with NVRAM, unless it needs to re-join later as a backup (e.g., after failover).

Transferring log data. Query Fresh takes advantage of RDMA’s non-blocking nature to efficiently overlap the process of persisting log records locally with network I/O. Figure 7.1(a) illustrates how we ship log records. Upon group commit or log flush, the primary first issues an RDMA Write with Immediate request (one per backup) to store log records in each backup’s NVRAM log buffer, with the size of the data being shipped as the immediate value (step 1). Since we use RDMA Write with Immediate to ship log records, the backup needs to post a receive request to retrieve the immediate value. After sending the RDMA request, the primary flushes the log locally and/or to shared or local storage if NVRAM is not in place for the primary. Otherwise it suffices to persist data in NVRAM using CLWB (followed by a store fence), and log records can be de-staged in the background [174]. Meanwhile, the
network I/O initialized in step 1 will happen in parallel with log flush. Once the log is persisted locally, the primary starts to poll the result for the write posted for each backup (part of step 3), to ensure that the write request is sent and its completion event is consumed so that the RDMA library’s internal data structure (libibverbs in our implementation) is kept in its correct status. We find the polling overhead is negligible.

**Ensuring persistence.** Returning from the polling operation in step 3 only indicates that the write request has been successfully sent; there is no guarantee on when the data will be persisted in the backup, even with NVRAM log buffer on backups (discussed in Section 7.2.3). If the general-purpose server method is employed, the primary should explicitly wait for acknowledgement from backups to ensure log records are received and persisted on backups. Our current implementation of the general-purpose server method uses RDMA Write (by the backup) and atomic read (by the primary) for acknowledgement. The primary exports for each backup a status word that can be remotely written using RDMA Write and spins on it for a “persisted” signal, which is written by the backup after it persisted the data. Upon receiving acknowledgement, the primary commits all transactions covered by the just-shipped log records.

Two approaches could be used to remove/reduce the spinning on the “persisted” signal. The first approach is to rely on large battery arrays which provide de-facto persistence across the memory bus. They are being adopted by data centers [86]. Backups could directly acknowledge the primary—or even skip it—as data is “persisted” implicitly. The other approach reduces spinning, by allowing the primary to return before backups acknowledge, and check for persistence before reusing the log buffer space, e.g., before sending the next batch. Until getting acknowledgement, the primary does not commit the transactions covered by the shipped log records. This approach trades commit latency for throughput, so it is not recommended if low commit latency is desired.

To persist the received log records, backups issue CLWB and a store fence while the primary is waiting in step 3 of Figure 7.1(a). The speed of persisting data in NVRAM is determined by memory bandwidth; a single flusher thread could make the system NVRAM-traffic bound. To reduce persistence latency, we introduce a persist-upon-replay approach that piggybacks on multiple replay threads to divide the work. As Section 7.3.4 describes, each log replay thread is responsible for redoing log records stored in one part of the log buffer. Before starting replay, each replay thread first issues CLWB and store fence to persist its share of log records. The log shipping daemon then acknowledges the primary once all replay threads have finished persisting their share of log data.

**Guaranteeing strong safety.** Our log shipping design follows the definitions of strong safety in Sections 2.1.4 and 7.2.1: durability is ensured on the commit path for all backups, and transactions are
never committed if their log records are not durably replicated across all or a majority of nodes in the cluster. Hence, Query Fresh provides strong safety. Note that log replay does not affect safety guarantees: a safe system only needs to ensure log records are properly persisted to not lose committed work; whether the log records are quickly replayed has more impact on data freshness, not safety guarantees.

**Discussion.** Overlapping network and storage I/O is not new and can be done with TCP/IP, but often requires extra implementation efforts. RDMA’s asynchronous nature makes it possible to easily piggyback on the existing commit daemon. RDMA also provides superior performance with kernel bypassing. An inherent limitation of RDMA Write/with Immediate is that these operations are only available under unicast [117], limiting the number of backups that can be kept synchronous. For example, in theory a 56Gbps network can keep up to seven synchronous backups, if the primary’s log rate is 1GB/s. The number will likely become lower with necessary overheads (e.g., acknowledgements). Section 7.4 shows this effect.

### 7.3.2 Replay Pipelining

Maintaining fresh data access requires the read view on backups does not fall much behind the primary’s. A straightforward solution is synchronous log replay that ensures replay is finished before sending persistence acknowledgment to the primary. Although it guarantees absolute freshness, i.e., the primary and backup always exhibit exactly the same read view, synchronous replay significantly slows down the primary with replay on the critical path.

We solve this problem by replay pipelining. The key is to overlap log replay with forward processing without lagging behind. After persisting log records locally using CLWB, instead of waiting for replay to finish, the backup acknowledges immediately (if needed). Meanwhile, the backup notifies its own log flusher to de-stage the received log records from NVRAM to storage. Without waiting for the flush to finish, the backup starts immediately to replay log records by notifying its replay threads, each of which scans part of the log buffer. As a result, the replay of batch $i$ is “hidden” behind forward processing that generates batch $i+1$. This way, log replay is kept out of the critical path as long as replay is fast enough to finish before the next batch of log records arrives.

For replay pipelining to work, the primary and backup need to coordinate with each other on when a batch of log records can be shipped: the backup must ensure the log buffer space to be used for accepting new data is available, i.e., replayed and de-staged to storage. We solve this problem with a **ReadyToReceive** status on the status word exported by the primary. Before sending a batch of log records, the primary waits for the backup’s status to become **ReadyToReceive**, which is posted by the backup using RDMA Write after it has replayed and de-staged the previous batch.

Replay pipelining works nicely with multi-buffering, a widely used technique to reduce log buffer wait time. As Figure 7.1(b) shows, while the commit daemon is shipping a batch of log records, worker threads may insert to the other half of the log buffer. With multi-buffering, the log buffer space is essentially divided into multiple ranges, allowing the backup to accept new log data while replaying a previous batch. For each range, we maintain an “end LSN”. If a range’s end LSN is no greater than the most recently persisted and replayed batch’s LSN, then the range is available for new data. The backup could issue **ReadyToReceive** and accept new data as soon as it finds that the next range is free, allowing batch $i+1$ to flow in while batch $i$ is being replayed. Alternatively, the backup could delay the issuing until each current batch is replayed for better freshness. Compared to synchronous replay, replay pipelining could reduce freshness because commit does not guarantee immediate visibility on backups. As we show in
7.3.3 Basic Append-Only Storage

The ideas of append-only storage and indirection are not new and have been employed in many systems [11, 19, 89, 100, 103, 151]. We first review how it works for single-node systems, and then extend the design for log shipping and fast log replay.

With append-only storage, data is never overwritten; updates to the database are always appended to the end of the log. Each record is associated with an RID that is logical and never changes. Transactions access data through another level of indirection that maps RIDs to the record’s physical location. Indirection can be used by both single and multi-version systems; we base our work on ERMIA [89] (a multi-version system) and employ its append-only design. As Figure 7.2 shows, each table is accompanied with an indirection array. Each entry in the indirection array is an 8-byte word, excluding the “RID” field shown in Figure 7.2, which is an index into the array and only shown for clarity. The indirection array entry could point to the in-memory version or contain an offset into the log; a later access can use approaches such as anti-caching [40] to load the version to memory and replace the entry with a pointer to the in-memory record. Our current implementation reserves a lower bit in the indirection array entry to differentiate virtual memory pointers and offsets into the durable log [89].

With indirection, indexes map keys to RIDs, instead of the records’ physical locations. All indexes (primary and secondary) on the same table point to the same set of RIDs. RIDs are local to their “home” indirection array, and allocated upon inserts. We assign each indirection array (i.e., logical table) a unique file ID (FID), and the combination of FID and RID uniquely identifies a database record. FIDs are maintained in exactly the same way as RIDs; a special “catalog” indirection array maps FIDs to indirection arrays. Each index stores a pointer to the underlying indirection array (or simply the FID). Worker threads can query an index first for the RID, then consult the table’s indirection array to access the record.

---

2 Query Fresh is also applicable to other systems, as long as they can be equipped with indirection arrays for data access and more importantly, fast log replay.
The benefit of employing append-only storage with indirection is two-fold. First, updates that do not alter key fields require no index operations: all they need to do is to obtain the RID (e.g., by traversing an index) and modify the indirection array entry to point to the new version (e.g., by using a \texttt{compare-and-swap} instruction or latching the version chain). This is especially beneficial if a table employs multiple secondary indexes, as without indirection arrays the updater will have to make the key in every index point to the new version. Second, thanks to the adoption of redo-only physical logging, recovery becomes as simple as setting up indirection arrays while scanning the log records, without generating “real” records (unless otherwise required) like in systems without indirection.

### 7.3.4 Append-Only Storage for Log Shipping

Query Fresh employs indirection in both primary and backup servers, but follows a slightly different design to support lightweight, parallel log replay, concurrent read-only queries, and failover.

#### Making Replay Lightweight

The design in Section 7.3.3 has the potential of reducing the amount of work of replay down to its minimum: replay threads simply store the physical location of each log record (which is the actual data tuple) in the indirection array. However, blindly applying this technique could significantly hurt read performance: an indirection array entry usually points to a list of in-memory versions, and replacing the pointer with a log offset in fact forcibly evicts the in-memory versions, because after replay, new readers will not be able to see them and thus must reload versions from storage. Moreover, additional tracking is needed to avoid leaking memory by recycling the in-memory versions once they are not needed by existing readers.

To solve this problem, for each table we employ an additional \textit{replay array} which only stores the latest physical, permanent address of the record. Figure 7.3 describes the detailed design. The data array still holds pointers to all in-memory records (or possibly pointers to the durable log after recovery). Replay threads simply parse the log in NVRAM and store each version’s physical address in the replay array. Log replay threads never incarnate versions by loading data from storage or NVRAM to main memory. This way, we reduce the amount of work that must be done by log replay to its minimum while still maintaining high read performance. For inserts, the replay threads must also insert the key-RID pairs to indexes. This method works for any index. An alternative is to also use indirection arrays for index nodes, so that index node updates are logged and replayed in the same way we handle database records. Indexes such as the Bw-tree \cite{103} will fit in this approach more easily. After the log data is persisted globally (in NVRAM or storage) and replayed, the (new or updated) records become visible to read-only transactions being run by threads that are not dedicated for log replay. Since the latest updates are all only reflected on replay arrays, readers must also examine them when trying to find suitable versions to access.

#### Parallelizing Log Replay

In Query Fresh we employ on the backup multiple threads to replay the log received from the primary. Besides reducing the amount of work that needs to be done for each record by employing the replay arrays, another important aspect is to reduce the amount of data that must be examined by each thread. We achieve this by assigning each replay thread part of the log buffer to scan and replay. For this to work,
Figure 7.3: Append-only storage with indirection for log shipping. An additional replay array that is indexed by the same RIDs as the data array, stores the latest physical address of each record.

in addition to the log buffer data, the primary also sends metadata that describes the logical partitions of the shipped data for each replay thread to follow. Note that blindly assigning each thread an equal portion of the log to replay is not feasible: log records are of variable sizes, so we must ensure replay threads can locate the right log record in its partition to begin replaying.

The partitioning information is generated by the primary during forward processing. We logically divide the log buffer space into a predefined number (n) of partitions of equal size. Whenever a log buffer allocation crosses a partition boundary, the primary records the allocation’s end log record offset. The primary sends such metadata using RDMA Write with Immediate whenever it ships log records. Upon receiving the log data and metadata, the backup will employ multiple threads to replay the log records in parallel, using the metadata. Note that we do not have to employ n threads on the backup for replay; each thread can replay multiple partitions. In addition, as we have described in Section 7.3.1, these replay threads can also issue CLWB and store fence instructions in parallel to persist log data before start replaying, amortizing the cost of persistence.

Cooperation of Readers

Backups must guarantee consistent read access. Versions created by the same transaction (on the primary) must not be visible to read-only transactions (on backups) before all of them are successfully replayed and persisted globally, i.e., installed in the replay arrays and durably stored in NVRAM or storage of all or a majority of nodes. Ensuring read-only transactions only see globally persisted data guarantees correctness if the primary fails: multiple backups in the cluster could replay and persist locally in different speeds, so without proper visibility control, read-only transactions on the faster nodes could see uncommitted data if the primary crashes before receiving acknowledgement from all backup nodes.

We use LSNs to control read visibility, which is a common approach in multi-version systems. Here we describe the approach in ERMIA [89], which Query Fresh is based on; other systems might use different approaches but the general principle is similar. When entering pre-commit, the transaction acquires a
globally unique commit timestamp and reserves log buffer space by atomically incrementing a shared counter by the size of its log records, using the atomic \texttt{fetch-and-add} (FAA) instruction.\footnote{This instruction atomically increments the given memory word and returns the value stored in the word right before the increment~\cite{faa}.} The FAA’s return value indicates the transaction’s commit order and its log records’ starting offset in the durable log. For each record we use its location in the physical log, as its commit stamp. All the log records generated by the same transaction are co-located in the log and a record with a larger LSN is guaranteed to be stored in the later part of the log.

To control read visibility in backups, in each backup we maintain two additional LSNs: (1) the “replayed LSN” which is the end offset in the durable log of the most recently replayed log record, and (2) the “persistent LSN” which indicates the end durable log offset of the most recently persisted log record. A read-only transaction on the backup starts by taking the “read view LSN”, which is the smaller of the visible and persistent LSNs as its begin timestamp $b$. It accesses versions with the maximum commit LSN that is smaller than $b$, and skips other versions. This way, a read-only transaction always reads a consistent snapshots of the database. During replay we install versions and bump the replayed LSN only after we have replayed all the log records for the same transaction. The persistent LSN is updated by the primary using RDMA Write after receiving acknowledgements from all backups. Thus, read-only transactions on backups will never see uncommitted data.

With a begin timestamp and target RID (obtained after querying the index or by a full-table scan), a read-only transaction on the backup starts by examining the target table’s data array. If the head version is already newer than the transaction’s begin timestamp, the transaction will continue to traverse the version chain to find the proper version to read. However, if the head version is visible to the querying transaction (i.e., its commit stamp is smaller than the begin timestamp), the transaction will have to examine further the replay array in case a newer version is visible. Since a record’s physical position in the log also reflects the global order of the transaction that generated it, when probing the replay array, the transaction directly compares its begin timestamp with the RID entry value. If the RID entry’s value is larger than the LSN of the head version on the data array, then we continue to follow the physical address recorded by the RID entry to instantiate the latest version—no matter if it is visible to the querying thread or not—and continue to load older versions for the RID until we hit a visible version or the latest version represented by the data array (whichever is younger). Then, the querying thread will try to install this “sub-chain” of versions to the version chain on the data array and read the correct version, and retry the whole read operation if the installation failed (e.g., because another thread acted faster doing the same operation).

When loading versions, worker threads opportunistically read from the NVRAM-backed log buffer for better performance. It verifies that data read is still valid by checking the LSN range currently being represented by the log buffer. If the LSN range does not contain the read data’s range, we read from storage.

\section*{Failover and Catch-up}

All nodes (the primary and backups) in Query Fresh use exactly the same binary. If the primary is down, a backup (previously designated or elected~\cite{designated}) will take over and become the new primary. This failover process involves both the new primary and other backups. First, the new primary needs to finish replaying the last batch of log records received from the old primary (if there is any). At the same time,
it switches to the “primary” mode by notifying all other backups about the take-over. Then, the other backups will re-establish connections to the new primary and obtain the latest log data from it. Once all the remaining nodes are ready, the new primary can start processing read/write transactions. The new primary can continue to finish processing the read-only transaction requests received while it was a backup, and use the previously log replay threads for read/write transactions. When all the “old” read-only transactions are finished, the primary can employ all its threads for read/write transactions. This allows the user to run long, or even mission critical read-only queries without worrying about impact by failover. Replay arrays are the primary home for new updates on backups. Thus, when a backup takes over and starts to process read/write transactions, it must also consult the replay arrays as if it still were a backup server if the data array’s head version is visible (following the steps described in Section 7.3.4). However, we do not expect this to become a major problem, as the read-only transactions that ran when the primary was a backup server should have already loaded most recent versions from storage/NVDRAM to DRAM.

When the failed primary comes back online, it joins the cluster as a backup by connecting to the current primary. The process follows the same procedure as a new backup server wanting to join the cluster would follow. After connected to the primary, the new backup starts to accept and replay log records as if it already caught up with the primary using the replay techniques we proposed, but postpones the update of its replayed LSN to ensure correct read visibility. At the same time, the backup in another thread obtains from the primary a recent checkpoint plus the log records generated after the checkpoint was taken, but before the first batch of log records received from the primary since the connection is established; we call this data “catch-up records”. Alternatively, the backup could read catch-up records from shared storage if data is stored there. The replay of catch-up records and “live” log records shipped synchronously from the primary are replayed concurrently. If a recent checkpoint is available, or in the case of a re-joining primary, the new backup can start processing read-only transaction once the checkpoint is recovered or the node is recovered and synchronized with the new primary, respectively. The data might be stale depending on how old/new the checkpoint is and/or how long the node has been offline. We also parallelize the replay of catch-up records. This can be done in various ways and is orthogonal to the design choices of Query Fresh. In our current implementation, we logically partition the checkpoint and log files by RID. Each thread only replays its own partition. Recall that our replay mechanism only stores the location of the latest record in replay arrays, so the replay of catch-up records must not update the replay array entry if a newer one is already there.

Note that the size of the catch-up records is fixed, because once connected the new backup starts to accept new log records synchronously and keep up with the primary. As a result, the new backup is guaranteed to catch up with the primary eventually. Once the catch-up records are all replayed, the backup can update its replayed LSN and declare it has caught up with the primary.

7.3.5 Discussion

Figure 7.4 reasons about the relative merits of Query Fresh and three other popular approaches, by examining how well or badly each design satisfies various design goals. In the figure, we place properties of interested as dots and use the relative distance between blue and grey dots to represent the overhead. Dotted lines connect the ideal cases, while solid lines connect the real cases. Figure 7.4(a) summarizes our previous discussion on synchronous log shipping in a slow network: the primary is often bound by network I/O, despite its strong safety guarantees. To avoid being network bound, as Figure 7.4(b)
Figure 7.4: Relative merits of hot standby solutions. (a) Synchronous log shipping sacrifices performance and freshness in a slow network. (b) Asynchronous log shipping trades safety for performance. (c) Logical logging reduces network traffic, but is tightly coupled with concurrency control and trades utilization for freshness. (d) Query Fresh strikes a balance on all aspects.

shows, log records are often shipped asynchronously and a transaction is allowed to commit locally in the primary without ensuring the log records are durable also in backups. Asynchronous log shipping gives neither safety nor freshness, although it does not block the primary. Backups have to serve read-only queries with stale data that is a potentially much earlier snapshot of the primary database. Committed work whose log records are not yet shipped when the primary fails could be lost, so inconsistencies might arise after a stale secondary took over.

Logical log shipping and deterministic execution [115, 164, 185], as shown in Figure 7.4(c), replicate the operations to perform, instead of results (i.e., physical data) of the operations. This approach saves network bandwidth, but falls short on other aspects. First, many deterministic execution schemes only support stored procedures. Applications that use data-dependent operations may not work. Second, as Section 7.4 shows, the amount of compute resources needed for replay on the backup is similar to that needed on the primary, leaving less resource for read-only transactions. Third, it requires careful handling of any randomness that might arise during transaction processing for correct log replay, such as random numbers and lock acquisition sequence, and all the work must be redone from scratch at each backup. The problem becomes even worse for multi-version concurrency control, under which commit order and serial order could be different. Replaying the log (i.e., based on commit order) might cause the primary and backup to diverge; a scheme that works for single-version 2PL might not work for snapshot isolation [115]. Solving this problem needs tight integration with the underlying concurrency control mechanism. As Figure 7.4(d) shows, Query Fresh strikes a balance among all the properties of interest, using modern hardware for high primary performance and strong safety guarantees, and using append-only storage with indirection for fast replay, fresh data access, and high resource utilization.

7.4 Evaluation

This section empirically evaluates Query Fresh and compares it with traditional approaches, and confirms the following:

- With RDMA over fast network and NVRAM, Query Fresh maintains high performance for the primary server;
- The append-only architecture and replay strategy allow backups to keep up with the primary with little overhead;
Query Fresh gives better freshness for read-only queries on backup servers compared to traditional approaches.

### 7.4.1 Implementation

We implemented Query Fresh in ERMIA [89], an open-source main-memory database engine designed for modern hardware. We augmented ERMIA with our lightweight replay and indirection techniques for log shipping. Each replay array uses exactly the same data structure as in original ERMIA. Therefore, with the replay arrays, the memory space needed by indirection is doubled. However, given the abundant memory space in modern main-memory database servers, we expect such addition to be acceptable. For example, since indirect array entry is 8-byte long, a million-record table would require around 16MB additional memory space (8MB for the data and replay arrays, respectively).

For indexes, we use Masstree [116] in ERMIA and all accesses are done through indexes. Our current implementation does not employ indirection for the index itself. Log replay only needs to insert new keys to the index(es) for inserts; for updates we setup indirection arrays, and do not touch indexes. In our experiments, we did not find manipulating indexes only for inserts to be a bottleneck.

### 7.4.2 Hardware

**Testbed.** We run experiments on the Apt cluster [149] testbed, an open platform for reproducible research. We use eight dual-socket nodes, each of which is equipped with two 8-core Intel Xeon E5-2650 v2 processors clocked at 2.6GHz (16 physical cores in total) and with 20MB cache. Each node has 64GB of main memory, a Mellanox MT27500 ConnexX-3 NIC and an Intel X520 Ethernet NIC. All nodes are connected to a 10Gbps Ethernet network and a 56Gbps FDR 4× InfiniBand network.

**NVRAM emulation.** The only DIMM form-factor NVRAM available on the market is NVDIMM [2, 171] that exhibits exactly the same performance characteristics as DRAM at runtime. There is also a current trend in data centers to use battery arrays for persistence [86]. So we use DRAM in our experiments.

We show results for (1) the ideal case which assumes batteries in data centers or future RDMA protocol enhancement that guarantees NVRAM persistence, and (2) variants that employ the general-purpose server method for NVRAM persistence. In the ideal variant, backups acknowledge the primary right after receiving data without additional work. We test two variants of the general-purpose server method, denoted as “CLFLUSH” and “CLWB-EMU”. The former uses the CLFLUSH instruction to persist log data. CLFLUSH will evict cachelines and impose non-trivial overhead. We only show these numbers for reference as lower-bound performance. Compared to CLFLUSH, the CLWB instruction does not evict the cacheline during write-back. CLWB is not available in existing processors, so we emulate its latency by busy-spinning. We calibrate the number of cycles for spinning as the number of cycles needed to write the same amount of data using non-temporal store (MOVNTI), which do not pollute the cache [72]. Our emulation is best-effort; we expect the actual CLWB instruction to perform better. For the experiments that follow, we use the ideal variant unless specified otherwise.

---

4 Truly NVRAM products (e.g., Intel 3D XPoint [39]) are available, but are only offered in PCIe interface as of this writing.
7.4.3 Experimental Setup

**Workloads.** On the primary server, we run the full transaction mix of the TPC-C [167] benchmark to test transactional read/write workloads. We fix the number of warehouses to the number of concurrent threads and give each thread a “home” warehouse. A major advantage of hot-standby solutions is the ability to serve read-only transactions to backup servers. Since the full TPC-C mix is a read-write workload, we run TPC-C’s read-only transactions (Stock-Level and Order-Status, 1:1 breakdown) on backup servers.

**System settings.** To focus on evaluating the impact of network and system architecture, we keep all data memory-resident using tmpfs. Log data is still flushed through the storage interface. For all experiments we set the log buffer size to 16MB and ship the log at group commit boundaries. Each run lasts for 10 seconds, and is repeated for three times; we report the averages. Like most (physical) log shipping mechanisms, concurrency control and replication are not tightly coupled in Query Fresh. Our experiments use snapshot isolation; other isolation levels are transparently supported by Query Fresh, so we do not repeat experiments for them here.

**Variants.** We compare Query Fresh with other approaches described in Section 7.3.5 using end-to-end experiments. Then we conduct detailed experiments for Query Fresh to show the impact of individual design decisions; the details are described later. We compare four variants in end-to-end experiments:

- **Sync:** Traditional synchronous log shipping; log records are shipped upon group commit, and transactions are not considered committed until log records are made durable in all replicas.
- **Async:** Asynchronous log shipping that commits transactions regardless log persistence status on backup nodes.
- **Logical:** Logical log shipping that sends over the network only commands and parameters, instead of physical log records; log records are batched and shipped synchronously as in Sync.
- **Query Fresh:** Synchronous log shipping with modern hardware and append-only storage architecture.

All variants are implemented ERMIA for fair comparison. For Logical, we use command logging [115], an extreme case of logical logging that generates only one log record per transaction. It requires all transactions must be stored procedures. A log record only contains the unique ID of the stored procedure and other necessary parameters. We choose command logging for its small logging footprint (thus reduced network traffic). However, it does not support correct replay of cross-partition transactions in multi-version systems (see Section 7.3.5 for details) [115]. Studying how to guarantee correctness for command logging in multi-version systems is a separate issue and out of the scope of this chapter. Nevertheless, we can still use it to measure the impact of log data size on performance, by slightly deviating from the TPC-C specification and restricting each worker to always access data associated with its home warehouse, i.e., no remote transactions. The benchmark may perform slightly better due to lack of conflicts, but it keeps the key benefit of command logging, which is the focus of our experiments.

Unless specified otherwise, we use four threads for log replay, and on backups threads that are not dedicated to log replay run read-only queries. Except Query Fresh, log replay is in the background and versions are fully instantiated without using replay arrays; Query Fresh uses replay pipelining. We run Sync, Async, and Logical with TCP/IP in the 10Gbps Ethernet network, and run Query Fresh in the 56Gbps InfiniBand network. This way, we show the impact of both system architecture and network on performance. To quantify the impact of individual design decisions, e.g., effect of the append-only
Table 7.1: Throughput and log data rate of a standalone server, vs. the primary’s throughput in a dual-node cluster under 56Gbps InfiniBand RDMA and 10Gbps Ethernet TCP/IP.

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>Standalone (kTPS)</th>
<th>Log size (MB/s)</th>
<th>RDMA (kTPS)</th>
<th>TCP/IP (kTPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>53.82</td>
<td>108.21</td>
<td>51.65</td>
<td>50.66</td>
</tr>
<tr>
<td>2</td>
<td>99.11</td>
<td>198.83</td>
<td>93.86</td>
<td>92.84</td>
</tr>
<tr>
<td>4</td>
<td>191.95</td>
<td>383.23</td>
<td>180.99</td>
<td>176.80</td>
</tr>
<tr>
<td>8</td>
<td>346.34</td>
<td>694.83</td>
<td>326.17</td>
<td>294.54</td>
</tr>
<tr>
<td>16</td>
<td>624.74</td>
<td>1456.62</td>
<td>586.80</td>
<td>336.20</td>
</tr>
</tbody>
</table>

architecture and replay policies, we run additional experiments using Query Fresh that turn on and off features (e.g., replay pipelining).

### 7.4.4 Network Bandwidth and Log Data Rate

We first calibrate our expectations by comparing the performance of a standalone server and a dual-node cluster. We focus on measuring the impact of network bandwidth and log data rate, i.e., how much bandwidth is needed for synchronous log shipping so that network is not a bottleneck. To achieve this, we turn off log replay and do not impose NVRAM delays. The backup does not run transactions; it only receives and persists log records. We then extrapolate the expected scalability of Query Fresh and other approaches, and compare our expectations with experimental results later.

Table 7.1 lists the throughput of a standalone server and its log rate. The standalone system scales and generates up to 1.42GB/s of log records. Such log rate well exceeds the bandwidth of a 10Gbps Ethernet. The 56Gbps InfiniBand network should in theory support up to five synchronous backups. As Section 7.3.1 explains, this is an inherent limitation of unicast. The log rate grows roughly at the same speed as the amount of parallelism grows. Since network bandwidth will likely be in the same ballpark with memory bandwidth (with multiple links/channels) [68], we estimate for larger servers, it is viable to at least have 1–2 synchronous backups. More backups can be added in a hierarchical architecture that is already widely in use today.

As the table shows, RDMA-based log shipping does not slow down the primary much, with 4–6% overhead over the standalone server. With 10Gbps Ethernet, the primary could perform up to ∼43% slower (16 threads). This indicates that 10Gbps Ethernet is unable to support any synchronous replica without significantly lowering the primary’s performance. This experiment verifies the importance of network bandwidth for physical log shipping. Compared to TCP/IP, we do not observe significant performance gain from RDMA’s kernel bypassing, either. This is largely because both interconnects are able to sustain high performance with bulk data transfer, which is usually the case for physical log shipping. However, the prospect of having safe RDMA over NVRAM (with protocol extensions) will likely make RDMA the preferred choice. Next, we expand our experiments to more (up to eight) nodes.

### 7.4.5 End-to-End Comparisons

Section 7.3.5 qualitatively compares related approaches. Now we compare them quantitatively. We focus on measuring (1) primary performance, (2) backup freshness, (3) resource utilization, and (4) implementation efforts in terms of lines of code (LoC).
Primary performance. We first measure the impact of the way (logical vs. physical) and timing (before vs. after committing locally) of log shipping. Figure 7.5 shows the throughput of the primary server under different approaches. As the figure shows, Query Fresh can support up to five synchronous backups while maintaining high performance for the primary. Once the log rate comes close to the network bandwidth (with five backups), primary throughput starts to drop. Sync bottlenecks on the network and cannot support any synchronous backups without significantly slowing down the primary. These results match our expectation in Section 7.4.4.

Async keeps high primary throughput regardless of the number of backups, because transactions are committed locally in the primary, before log records are sent to backups. Thus, it is possible to lose committed work and sacrifices both safety and freshness. Logical also preserves high primary performance across the x-axis in Figure 7.5, because command logging significantly reduces log data size, making it easy to support a large number of synchronous backups before saturating the network. However, it also exhibits various drawbacks as we have discussed in Section 7.3.5.

Freshness and utilization. Now we compare backup freshness and resource utilization of different approaches using two nodes; we obtained similar results with more nodes, so they are not shown here for brevity. We represent resource utilization by the percentage of CPU cores dedicated to read-only transactions. Freshness is measured by comparing the read view LSNs on the primary and backup. We synchronize the clock in each node with the same source, and take a snapshot of the read view LSN on each node every 20ms. We define the backup’s freshness score at a given time point as $\frac{b}{p} \times 100\%$, where $b$ and $p$ are the read view LSN of the backup and primary, respectively. For example, if at time $t$ the primary has a read view of 100, and the backup has finished replaying the log up to LSN 80, the freshness score will be 80%. A higher score indicates transactions can access a more recent snapshot of the database. Ideally, we can match the times between nodes and calculate freshness scores precisely. But it is difficult to obtain read view LSNs on two nodes at the exact same time. So for calculation, we take the LSN at the nearest 500ms boundaries. This allows us to estimate freshness scores more easily.

Figure 7.6 shows the results. We also give the number of replay threads used for each variant (e.g., Sync with four replay threads is denote as “Sync/4”). Query Fresh keeps up with the primary and provides over 99% of freshness using four replay threads, i.e., it never lags behind more than 1% of the primary’s read view. This shows the effectiveness of Query Fresh’s append-only architecture and indirection, which allow very lightweight, parallel replay. Although Logical never bottlenecks on shipping the log, it cannot reach the same level of freshness of Query Fresh using fewer resources (four replay
### Chapter 7. Fast and Fresh Hot Standbys

#### Figure 7.6: Backup freshness under 75% resource utilization (12 workers, 4 replay threads). Query Fresh keeps up with the primary (≥ 99% freshness). Others must trade utilization for more freshness.

<table>
<thead>
<tr>
<th>Time (second)</th>
<th>Sync/4</th>
<th>Logical/4</th>
<th>Query Fresh/4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Implementation effort

Based on ERMIA, whose source code has more than 70kLoC, the core implementation of Query Fresh took around 1300LoC. The number for TCP-based physical log shipping (Sync and Async) is ~800LoC. Because the only difference between Sync and Async is whether log replay happens in background, their code differs very little, so we do not separate them here. The extra ~500LoC in Query Fresh are for implementing functionality related to log replay and data arrays. Our implementation of Logical took ~330LoC, but it is not a complete implementation that guarantees correct replay for cross-partition transactions. We expect a complete implementation for Logical will need much more code tightly integrated with concurrency control and the application, whereas Query Fresh requires no change to concurrency control or the application.

#### Summary

These results show that Query Fresh strikes a balance among primary performance, freshness, and resource utilization. It also keeps the easy-to-implement feature of log shipping, and does not require any change in the application. Next, we explore how each design decision impacts Query Fresh’s behavior.
7.4.6 Effect of Replay Policy

We first explore the effectiveness of replay pipelining, by comparing it with synchronous replay described in Section 7.3.2. Figure 7.7 (top) plots the primary’s throughput. “NoReplay” denotes the variant where replay is disabled and reflects the amount of back pressure put on the primary by synchronous backups. Similar to what we have seen in the end-to-end experiments, after saturating the network (4–5 backups) throughput starts to drop significantly. Pipelined replay behaves similarly to NoReplay, because the overhead of log replay is hidden behind forward processing and is out of the critical path. However, synchronous replay does not scale after two backups with log replay on the critical path.

Figure 7.7 (bottom) shows the corresponding aggregate throughput of backups. Each backup employs all of its physical threads. In NoReplay, all the 16 threads in each backup are used to run read-only transactions. Due to the lack of log replay and because we start with a warm database that has all records in memory, reading a version involves no I/O. Thus, NoReplay gives an upper bound on how fast read-only transactions can run on backups. Synchronous uses eight threads for log replay, while the number for Pipelined is four. Compared to Synchronous, Pipelined is left with more threads for read-only queries, thus achieving higher throughput.

As Figure 7.8 (top) shows, Pipelined needs no more than four threads to keep up with the primary (after four backups network bandwidth becomes a scarcity). With 1–2 backups, Synchronous needs roughly half of the resources for log replay to keep up, as Figure 7.8 (bottom) shows. Since Synchronous and Pipelined perform exactly the same amount of work for log replay, this experiment shows the importance of moving replay out of the critical path.

7.4.7 Effect of Indirection

Pipelined replay employs indirection, so log replay does not actually instantiate “real” versions. Now we quantify its effect by comparing with a variant that fully replays the log (denoted as “FullReplay”).
Chapter 7. Fast and Fresh Hot Standbys

Figure 7.8: Throughput of the primary under pipelined (top) and synchronous replay policies.

Figure 7.9: Throughput of the primary with and without indirection.

FullReplay is exactly the same as Pipelined except that replay threads instantiate each version from the log record and install it on the data arrays, without using replay arrays. FullReplay employs multiple threads and can avoid replaying log records whose successors have already been replayed and installed on the data array. Figure 7.9 reflects the amount of back-pressure each variant could put on the primary. Both Query Fresh and FullReplay employ our pipelined log replay design. We use four replay threads and dedicate the remaining threads to read-only transactions. Compared to Query Fresh, FullReplay imposes significant overhead as it needs to fully instantiate versions,\(^5\) making log replay a major bottleneck.

7.4.8 Bootstrapping

Now we measure the time needed to start a new backup, including the time for recovering from a recent checkpoint and replaying the log generated afterward. The primary first generates a checkpoint after started, and then starts processing transactions for five seconds. Then we start a new backup and measure the time needed to finish checkpoint recovery and replay. The new backup uses 16 threads for checkpoint

\(^5\) We use a NUMA-aware allocator in ERMIA [89]. Our profiling results show that memory allocation is not a bottleneck.
recovery. Once it is done, it uses eight, four, and four threads for replaying catch-up log records, pipelined log replay, and running read-only transactions, respectively. The checkpoint size is 1.8GB, and the log data size is \( \sim 6GB \). Checkpoint recovery took 5.8 seconds, and replaying all the remaining catch-up log records took 24.21 seconds. So in total it took \( \sim 30 \) seconds for the backup to replay and catch up with the primary. To replay catch-up records, the threads have to load data from storage. So it is slower than replay pipelining which scans the log buffer directly. Moreover, our current implementation stores the whole checkpoint and log data in two separate files. Although we parallelize the replay process, using multiple threads to operate on the same file concurrently incurs severe bottleneck in the file system [122]. We expect an optimized implementation will be able to achieve even shorter catch-up time.

### 7.4.9 Commit Latency

Synchronous log shipping often greatly increase commit latency on the primary in a slow network. This is avoided in Query Fresh by overlapping log shipping with local I/O and replay pipelining. We collect the primary’s commit latency numbers for two Query Fresh variants using pipelined and synchronous replay, with a varying number of replay threads (denoted as “Pipelined/4” and so on). The group commit boundary is set to 4MB, i.e., log shipping is triggered whenever 4MB of new log data is generated. Other events (e.g., commit queue full or timeout) could also trigger log shipping, but we found the system is most sensitive to log data size.

Figure 7.10 shows the results. As a baseline, the commit latency of the standalone case (no replication) is 3.38ms. Synchronous replay exhibits \( \sim 1.04-4.74 \times \) higher latency, even before saturating the network (four backups). With replay pipelining, Query Fresh exhibits at most 1.16\( \times \) of the standalone latency before the network is saturated. Once the network is saturated (\( \geq 5 \) backups), however, all variants starts to add non-negligible latency on the commit path.

### 7.4.10 Persistence Delay

Now we quantify the impact of delays caused by persisting log records on backups using pipelined replay. We compare three types of variants: a variant that adds no delay (denoted as “Ideal”), a variant that does not use NVRAM (“No-NVRAM”) and four other variants that impose NVRAM delays. No-NVRAM assumes DRAM log buffer. Both Ideal and No-NVRAM use four replay threads, leaving 12 for read-only queries. For the remaining four variants, we use \texttt{CLFLUSH/CLWB-EMU} and vary the number of replay
threads between four and eight (denoted as “CLFLUSH/4” and so on); persist-upon-replay is enabled to spread the work of persisting the log records to multiple threads.

Figure 7.11(top) shows primary’s throughput. Ideal maintains high throughput with 1–5 backups. With four and more backups, network gradually becomes the bottleneck. Since CLWB-EMU does not evict cachelines, both CLWB-EMU variants achieve performance that is close to Ideal’s with up to four backups. With five and more backups, CLWB-EMU performs up to 9% slower than Ideal. CLFLUSH/8 incurs $\sim 5–26\%$ overhead over Ideal, due to the cache misses caused by CLFLUSH during log replay. With fewer replay threads, CLFLUSH/4 could add as much as 42% overhead on Ideal. These results show that data persistence is the bottleneck, rather than log replay, as we have shown that four threads are enough for pipelined replay to keep up with the primary.

Figure 7.11(bottom) shows the read-only transactions’ aggregate throughput on backups. Despite CLFLUSH/4 shows much lower primary performance than CLWB-EMU/4, it gave higher backup throughput, because slower replay reduces the chance of a transaction seeing a newer version posted on the replay arrays. More reads can be directly served through the data arrays without having to read versions in the “gap” between the replay and data arrays from storage. However, read-only transactions will access stale data. Variants with eight replay threads (CLFLUSH/8 and CLWB-EMU/8) exhibit lower backup throughput, due to the reduced number of threads for read-only transactions and higher chance for readers to have to load versions from storage since replay is faster.

7.5 Summary

Hot standby systems often exhibit a freshness gap between the primary and backup servers, for two reasons. First, network can easily be a bottleneck, limiting the speed of log shipping and making most backups operate in asynchronous mode, especially so for modern main-memory OLTP engines. Second, the traditional dual-copy architecture stores data in two permanent places: the log and the “real”
database, mandating (expensive) log replay before data can become accessible to read-only transactions on backups.

We propose Query Fresh to solve these problems with modern hardware and software architecture re-design. Query Fresh leverages RDMA over fast network and NVRAM for fast log shipping, and employs append-only storage with indirection for lightweight, parallel log replay. The result is a hot standby system that provides strong safety, freshness, and high resource utilization. Our evaluation using an 8-node cluster shows that with 56Gbps InfiniBand, Query Fresh supports up to 4–5 synchronous backups without significantly lowering the primary’s performance.
Chapter 8

Related Work

This chapter discusses existing work that is related to our proposals in previous chapters. Here we begin with logging, then we discuss related work in concurrency control and its relationship with synchronization primitives. Finally, we discuss primary-backup replication.

8.1 Logging

Most related work in logging focuses on utilizing NVRAM and/or optimizing for multicore processors.

8.1.1 Utilizing NVRAM

There has been active research on using NVRAM to reduce logging overhead, e.g., removing the disk and using NVRAM as the sole logging device [48]. Such techniques typically require two (expensive) epoch barriers [34] per log write and partial overwriting to tolerate holes in the log (necessary to improve parallelism), and are thus not implementable on current hardware; the partial overwriting also accelerates device wear-out. PCMLogging [51] employs PCM as both data buffer and log space. Pelley et al. [140] demonstrated different uses of NVRAM in database systems, but also relies on the epoch barrier. MARS [33] removes LSN and checkpointing, and moves most functionality of logging into hardware. Write-behind logging [6] writes out changes to the database during transaction processing before logging them logically, allowing near-instant recovery. SOFORT [136] operates directly on NVRAM without using a dedicated log to provide instant recovery. FlashLogging [30] uses multiple USB flash sticks to boost logging performance. Flag Commit [131] embeds transaction status into flash pages by a chain of commit flags to minimize the need of logging. Although these proposals differ in how they achieve improved logging performance, most of them remain thoroughly centralized and thus subject to the kinds of contention we propose to eliminate in Chapter 3.

8.1.2 Optimizing for Parallel Processors

Group and asynchronous commit [52,143] are two widely used techniques for reducing logging overhead. Group commit aggregates multiple commit requests into a single I/O. It improves throughput at the expense of transaction response time. Asynchronous commit allows transactions to complete without waiting for the log records to be de-staged to disk. It completely eliminates log flush delays, but risks
losing committed work upon failure due to the volatility of DRAM. Recent research focused more on parallel hardware. Aether [77] allows locks to be released earlier to reduce lock contention due to long log flush waits, reducing the impact of policies such as group commit. Aether also reduces log contention, but those mechanisms rely on shared caches and do not perform well in multi-socket NUMA environments [78].

C-ARIES [155] enhances ARIES with multi-threaded recovery, without changing log insertion (or the contention that accompanies it) in any way. Much recent work has focused on accelerating recovery for main-memory database systems. PACMAN [182] utilizes application-specific information that can be extracted through static analysis to speed up recovery. Adaptive logging [185] judiciously employ physical and logical logging to reduce recovery time.

8.2 Concurrency Control

Concurrency control methods are usually categorized in two orthogonal dimensions, in terms of the way they access data (pessimistic vs. optimistic) and the way they organize data (single vs. multi versioned). A common property of interest for all these concurrency control schemes is the isolation level a given concurrency control method provides. The possibility of various isolation levels, some of which are not serializable, was defined by Gray et al. [60]. Definitions of isolation properties based on patterns of dependency edges are given by Adya [1]. With a focus on serializable concurrency control schemes, now we discuss related work in concurrency control roughly following the categories we have given above.

8.2.1 Pessimistic Locking

Many database systems use pessimistic locking, i.e., a transaction must acquire a lock that protects the desired tuple to access. Two-phase locking (2PL) is the most widely used pessimistic locking approach that guarantees serializability [46]. The basic rule for guaranteeing serializability is that no locks can be granted after the transaction starts to release any lock. Additional locks (e.g., next-key locks [125] and gap locks [106]) prevent phantoms.

8.2.2 Optimistic Concurrency Control

OCC was initially formulated by Kung et al. [93]. We have described it in detail in previous chapters, so we do not repeat it here. OCC is lightweight and reduces cacheline invalidation, so it is especially attractive for future parallel hardware with thousands of cores and deep memory hierarchies. Recent OCC-based systems focus on eliminating physical contention to achieve good performance. FOEDUS [90] uses an extremely decentralized design inspired by Silo [169] to provide high performance on traditional TPC-C like workloads. BCC [189] identifies specific dependency graph patterns to reduce false aborts caused by vanilla OCC. TicToc [188] avoids centralized timestamp allocation and can commit certain transactions that would be aborted by traditional timestamp ordering schemes.

Certification approaches that guarantee serializability are forms of optimistic concurrency control. Obtaining exact accuracy using serialization graph testing was proposed by Casanova and Bernstein [28], and extended to support multi-versioning by Hadzilacos [62]. A different approach tests for cycles before transactions start in a real-time database system [98].
8.2.3 Hybrid Approaches

There is prior work that combines pessimistic locking and OCC. Static and dynamic OCC [186] switch from OCC to locking when retrying a transaction that was aborted under OCC. Static OCC acquires all locks before rerun, while dynamic OCC acquires locks upon data access. Thomasian [163] proposes to lock both reads and writes before the global validation in distributed OCC. Locks are retained and re-used should the validation fail. Despite their similarity to MOCC, most prior approaches are designed for disk-based or distributed systems, instead of modern main-memory systems. For instance, a committing transaction under OCC with broadcast during rerun [186] aborts conflicting rerun transactions but allows first-run transactions to run to completion so that all data are brought from disk to main memory. Compared to MOCC, prior hybrid approaches require a transaction use either pure locking or OCC, without considering data temperature.

Among recent systems, Hekaton [42] supports pessimistic locking on top of its MVCC protocol. Ziv et al. [194] formalize the theory of composing computations over multiple data structures using different concurrency control schemes. Cao et al. [27] use OCC and pessimistic concurrency control to handle low and high conflict accesses, respectively. They focus on single entry accesses, instead of serializable executions over multiple operations. Contrary to traditional pessimistic and optimistic concurrency control methods characterized by whether there will be conflicts among transactions, LarkTM [191] combines pessimistic and optimistic methods that are characterized by whether a lock will be “required” by the same reader or writer. LarkTM operates on the granularity of objects and gives serializability over multiple operations.

8.2.4 Multi-Version Concurrency Control

MVCC maintains multiple versions of each tuple and determines whether a transaction can access a version by comparing its begin timestamp and the version’s timestamp. MVCC has been widely adopted in recent main-memory systems [42, 89, 102]. However, a straightforward implementation of MVCC, such as snapshot isolation (SI), is not serializable [14]. Most recent efforts on adding serializability to SI are certification-based. Serializable SI (SSI) [25] tracks and avoids the “dangerous structures” that will appear in non-serializable executions. The original SSI algorithm runs specifically along with SI to ensure serializable executions [26]. An improved form (which we call SSI in Chapter 4) was implemented in PostgreSQL [141]. Reivilak et al. proposed accompanying SI with an exact certification using serialization graph testing [148]. Other certification algorithms have been developed that can be used in a snapshot-based system (one where all reads within a transaction come from a common snapshot). Lomet et al. [105] choose a commit timestamp from an allowed interval, and the chosen timestamp is the effective serial order commit time. SSN, on the other hand, uses the commit time as timestamp, and the chosen timestamp does not necessarily coincide with the actual serial order commit time. Hekaton [42] specifically aims for main-memory stores and rejects all back edges. Deuteronomy [102] separates physical and logical operations with dedicated data and transaction components, and thus supports phantom protection while keeping a separation between the CC and storage layers [101]. BOHM [47] analyzes transactions before execution to determine serializable schedules. Neumann [129] et al. adapts precision locking [79] and uses undo buffers in HyPer [87] to validate serializability.
8.2.5 Pre-Analysis for Serializable Execution

Another class of proposals ensure Serializable execution by doing static pre-analysis of the application mix [4, 49, 80]. In the context of main-memory optimized systems, Bohm [47] determines Serializable schedules prior to transaction execution, requiring that transactions submitted in their entirety with the write sets deducible before execution. Unlike the other approaches we have discussed, these methods are not suitable with ad-hoc queries or data dependent queries.

8.2.6 Separating Concurrency Control and Transaction Execution

Most concurrency control methods assume transaction execution and concurrency control are handled by the same thread. Orthrus [145] takes a different approach, by using dedicated threads for concurrency control. Orthrus completely avoids deadlocks and alleviates the overhead to dynamically coordinate between executor threads by handing off the coordination to concurrency control threads via message passing and ordering locks in a consistent order like Silo and MOCC. The key drawbacks of Orthrus, as we have discussed in detail in Section 5.5, are that (1) it needs frequent interthread communication between concurrency control and execution threads, and (2) it is a static approach, i.e., the assignment of concurrency control and execution threads relies on offline oracle.

8.3 Synchronization Primitives

Our work stands upon the shoulders of much prior work on queuing locks. Since we have already covered the details on MCS lock in previous chapter, we do not repeat them here. In this section, we discuss the CLH queuing lock, followed by other related work.

8.3.1 CLH Lock

The CLH lock [38, 112] is a variant of the MCS lock where each lock requester, instead of spinning on its own flag, spins on its predecessor’s. Each queue node in a CLH lock maintains a pointer to its predecessor, whereas in the MCS lock it maintains a pointer to its successor. The head of the queue is a dummy node. A key difference between CLH and MCS is that, in CLH, a requester leaves behind its record for its successor and reclaims its predecessor’s record during its release protocol. As a result of reclaiming the predecessor’s record, the CLH lock needs to additionally manage its memory. Scott [152] proposes a technique to avoid the overhead by thread-local memory allocations.

8.3.2 Advances in MCS and CLH Locks

There have been many prior efforts to enhance the MCS and CLH locks to accomplish other objectives. Mellor-Crummey and Scott [120] relaxed the MCS lock for reader-writer synchronization accommodating multiple readers in a critical section. They explored three variants—fair-reader-writer, reader-preference, and writer-preference locks. For example, the fair-reader-writer lock enables a requester to safely access fields in its predecessor’s record during its release protocol. A reader who is enqueued immediately after another reader can notice the status of its predecessor (waiting or holding the lock) and enter its critical section without waiting for the predecessor to finish. If a successor reader finishes before a predecessor reader, the last finishing reader takes the additional responsibility of passing the lock to the first waiting writer.
Scott and Scherer [153] enhanced both MCS and CLH locks with the timeout capability allowing an enqueued process to abort after a period of waiting. The enhancement to the CLH lock adds additional states—transient, leaving, and recycled—to the flags field. These states are used to establish a handshake among the aborting thread, its predecessor, and its successor. When a thread wanting to abort is at the tail of the queue, it follows a more complex protocol to ensure consistency with an intervening successor that might abort as well. The modifications to the MCS lock are even more complex, especially when the aborting thread is at the tail of the queue. The MCS queue is transformed into a doubled-linked list from its original singly-linked list. The modified MCS lock uses a few special values to make sure an aborting thread leaves without causing dangling pointers in the linked list.

### 8.3.3 Queuing Locks that Support Guests

The idea of using “special” value(s) to indicate deviation(s) from normal behavior is not uncommon in the synchronization literature. For example, the aforementioned MCS and CLH locks with timeout also used a handful of “special” values when aborting. However, we are not aware of any prior work that uses such a special value in order to treat one type of user differently from another. Put in another way, to the best of our knowledge, our work is the first to explore the problem of occasional guest users that cannot provide a context (i.e., a queue node).

Notable prior art in terms of context-less locking is the variant of MCS lock implemented in the K42 operating system [9]. It is a queue lock, but a lock user does not need to provide a context. However, it loses the wait-freeness of MCS’s doorway protocol because it uses CAS to enter the lock acquisition. Experiments in Section 6.5 show degraded scalability due to this. Moreover, queue nodes in K42 live on the stack, ruling out its use in inter-process locking and in cohort-locking described next.

In Section 6.5, we evaluate a variant of the K42 lock that addresses these shortcomings with TLS [152]. However, we observe that this approach demands efficient TLS support in the platform and also does not scale as well as MCS/MCSg due to its lack of NUMA-awareness.

### 8.3.4 Lock Cohorting

In the context of NUMA locks, Dice et al. [43] devised lock cohorting, which composes two different synchronization protocols. Cohort locks are two-level locks—a global lock and a local lock. The global lock can be of any kind, whereas the local lock should be able to express the fact that there are waiters—e.g., MCS, CLH, TKT etc. The cohort locks dedicate a local lock per socket on a node and there is one global lock. Each thread wanting to enter the critical section competes for its local lock. The first thread to acquire the local lock proceeds to compete for the global lock; other threads wait for the local lock. Once a thread acquires the global lock and finishes its critical section, it releases its local lock if it notices local waiters. A waiting thread immediately enters the critical section without competing for the global lock after it is granted a local lock, effectively inheriting the global lock. A thread can pass the global lock within its NUMA domain for a “threshold” number of times to take advantage of locality. On reaching the threshold, the global lock is relinquished to another NUMA domain. A global back-off lock (BO) with local MCS locks makes a cohort C-BO-MCS lock. Similarly, one can devise C-BO-CLH, C-MCS-MCS, C-CLH-CLH, C-BO-TKT, C-TKT-MCS locks, among others.
8.4 Primary-Backup Replication

Many open-source and commercial database systems implement log shipping based hot standby solutions [67,133,134,144,161] and can be set to be synchronous or asynchronous, as we have discussed in Chapter 7. Most of these solutions are part of the database system itself. KuaFu [184] enables parallel log replay in MySQL by tracking transaction dependencies. Much recent effort has gone to optimizing replication for cloud environments. SHADOW systems [88] utilize cloud infrastructure to offload log shipping and other related jobs to the storage layer, reducing the amount of data to be duplicated at the database level. Similarly, Amazon Aurora [170] offloads redo processing to storage, reducing network load. The design of Query Fresh is largely orthogonal to these systems as our goal is to accelerate redo processing. RemusDB [123] provides high availability guarantees at the virtualization layer, requiring little or no customization of the database engine itself. Query Fresh’s indirection architecture is orthogonal to and can be applied in these environments to further accelerate redo processing for more fresh data access. Another popular approach to high availability is deterministic execution [164,165] and logical log shipping [115]. We have reasoned about the relative merits of these approaches in Section 7.3.5, so we do not repeat them here. BatchDB [113] supports hybrid workloads using dedicated OLTP and OLAP components that can situate on the same or two separate nodes. BatchDB uses logical logging for the OLTP node and propagates updates explicitly from the OLTP node to the OLAP node, which is dedicated to OLAP and does not takeover upon primary failure. Query Fresh is a general log shipping mechanism that enables fast synchronous log shipping without degrading performance of the primary, while providing strong safety guarantees.

8.4.1 Utilizing Fast Networks

The use of fast networks such as InfiniBand are being actively investigated in key-value stores [83,124], transaction processing and query processing engines. Rödiger et al. [150] propose a distributed query engine design for high-speed networks, tackling problems associated with traditional TCP/IP network such as TCP overheads. Barthels et al. [13] utilize RDMA-enabled buffers to distribute and partition data efficiently for parallel in-memory hash join algorithms. To fully realize the performance such fast networks can provide, Binnig et al. [22] suggest that one should use RDMA (instead of the TCP/IP stack over InfiniBand) to redesign distributed databases from scratch, and propose a network-attached memory (NAM) architecture where compute and storage nodes which provide a shared distributed memory pool are logically decoupled. NAM-DB [190] is a distributed transaction processing engine built upon the NAM architecture. It uses snapshot isolation (SI) and mitigates the centralized timestamp bottleneck typically found in SI protocols using vector clocks. FaRM [45] is a more general platform that exposes memory as a shared address space. It uses RDMA for both message passing and direct data access. While most systems utilize one-sided RDMA, FaSST [84] uses two-sided RDMA (unreliable datagram) to provide fast RPC calls, as the large number of queue pairs typically needed by reliable one-sided RDMA can significantly limit scalability.

8.4.2 Append-only Storage and Indirection

Distributed shared logs are a popular design that carry similarity as they abstract storage to be append-only, despite the different system model than assumed here. CORFU [11] organizes raw flash chips as a global, shared log, on top of which database engines can execute transactions optimistically [16,19,20] for
high performance. Instead of treating all the storage units as a global log, we focus on making individual
database engines in a distributed system append-only. Utilizing CORFU, Tango [12] provides mechanisms
to build in-memory data structures backed by a shared log.

The use of indirection is not unique in Query Fresh. Traditional disk-based systems have long been
using record IDs (RIDs). RIDs are around until some main-memory systems started to use direct memory
pointers to avoid extra indirection [90,169]. File systems such as BPFS [34] have also employed indirection.
Several recent proposals and evaluations, however, note the usefulness of indirection in main-memory
environments, especially for multi-version database systems [181]. The Bw-Tree [103] uses indirection
(the “mapping table”) to aid the build of lock-free B+-tree operations. Sadoghi et al. [151] use indirection
and store RIDs in indexes to reduce index maintenance cost. ERMIA [89] adopts the same philosophy
for easier logging, recovery and low index maintenance cost. In Query Fresh, indirection is a powerful
tool to reduce log replay time and ease the maintenance of secondary servers.

8.4.3 Utilizing Logging Techniques

Log shipping based hot standby is closely related to logging and recovery protocols. Query Fresh can
work with existing ARIES-like [126] logging and recovery mechanisms. In fact, in Query Fresh backup
servers use exactly the same recovery protocol for replay. Techniques such as in-page lazy recovery [57]
that accelerate log replay could also be employed by Query Fresh for faster replay and failover. Compared
to physical logging which Query Fresh currently is based on, logical logging saves network load [115] but
could be tightly coupled with concurrency control.
Chapter 9

Concluding Remarks

To deliver high performance and the functionality needed by applications, database engines must adapt to the underlying hardware. We have examined recent hardware trends, including the ways modern and future servers compute, store data, and transfer data between nodes. Specifically, we focused on massively parallel processors, large DRAM, NVRAM, and ultra-fast network interconnects. Existing database engines fall short on fully utilizing such new hardware. This thesis identifies the short-comings and proposes solutions.

In Chapter 3, we eliminated the logging bottleneck by using NVRAM to resurrect distributed logging, a once prohibitively expensive design. With the logging bottleneck removed, we continued to devise robust concurrency control mechanisms in Chapters 4–5. The goal is to gracefully handle a wide spectrum of OLTP workloads, rather than just their “sweet-spot” workloads. The serial safety net (SSN) is a cheap, general-purpose serializability certifier that can be applied on CC schemes that produce anomalies. More importantly, SSN is amenable to emerging read-mostly transactions, a type of applications enabled by emerging hardware but is often ignored by existing engines. Mostly-optimistic concurrency control explores ways to schedule transactions on large NUMA machines with high interthread communication costs. The key take-away is that communication must be minimized in such systems, and pessimistic locking should be judiciously used to provide robust performance for high-contention workloads.

The robustness and functionality requirements also extend to physical layer design, especially synchronization primitives which are an important building block of database engines. In Chapters 5 and 6 we highlighted the need for high-performance queue-based spinlocks to have read-writer modes, cancellation support, and a standard interface. The MOCC queuing lock and MCSg lock solved these problems.

Going forward from a single-node system to multi-node, hot standby systems, in Chapter 7 we proposed Query Fresh which provides both strong safety and fresh data access using storage architecture re-design on top of fast networks and NVRAM. The key to realizing this goal is an append-only storage design that allows lightweight, parallel log replay. Backup servers only need many fewer resources (around one fourth) to replay the incoming log records for query threads to consume.

A large body of work in this thesis was done before “true” NVRAM devices with large capacity became available on the market. A promising future direction is reconsidering how the design of database engines changes in the context of actual NVRAM devices, instead of emulated environments. Another promising future direction is devising more efficient indexing structures. Specifically, indexing appears to be the next component that costs the most CPU cycles in ERMIA.
In summary, we have examined recent hardware trends in compute, storage, and networking, and identified problems. Our solutions focus on getting both performance and robustness in the context of emerging hardware, covering major database engine functionality: logging, concurrency control, and replication. The techniques proposed in this thesis can serve as a toolbox for building future database systems. Although these techniques were proposed for database systems, they can be applied in other systems that face similar problems, such as file systems and operating systems.
Bibliography


