Abstract

High Performance Multi-core Transaction Processing via Deterministic Execution

Jose Manuel Faleiro
2018

The increasing democratization of server hardware with multi-core CPUs and large main memories has been one of the dominant hardware trends of the last decade. “Bare metal” servers with tens of CPU cores and over 100 gigabytes of main memory have been available for several years now. Recently, this large scale hardware has also been available via the cloud; for instance, Amazon EC2 now provides instances with 64 physical CPU cores. Database systems, with their roots in uniprocessors and paucity of main memory, have unsurprisingly been found wanting on modern hardware.

In addition to changes in hardware, database systems have had to contend with changing application requirements and deployment environments. Database systems have long provided applications with an interactive interface, in which an application can communicate with the database over several round-trips in the course of a single request. A large class of applications, however, does not require interactive interfaces, and is unwilling to pay the performance cost associated with overly flexible interfaces. Some of these applications have eschewed database systems altogether in favor of high-performance key-value stores.

Finally, modern applications are increasingly deployed at ever increasing scales, often serving hundreds of thousands to millions of simultaneous clients. These large scale deployments are more prone to errors due to consistency issues in their underlying database systems. Ever since their inception, database systems have provided applications to tradeoff consistency for performance, and often nudge applications towards weak consistency. When deployed at scale, weak consistency exposes latent consistency-related bugs, in the same way that failures are more likely to occur at scale. Nearly every widely deployed database system provides applications with weak consistency consistency by default, and its widespread use in practice significantly complicates application development, leading to latent Heisenbugs that are only exposed in production.

This dissertation proposes and explores the use of deterministic execution to address these concerns. Database systems have traditionally been non-deterministic;

given an input list of transactions, the final state of the database, which corresponds to some totally ordered execution of transactions, is dependent on non-deterministic factors such as thread scheduling decisions made by the operating system and failures. Deterministic execution, on the other hand, ensures that the database's final state is always determined by its input list of transactions; in other words, the input list of transactions is the same as the total order of transactions that determines the database's state.

While non-deterministic database systems expend significant resources in determining valid total orders of transactions, we show that deterministic systems can exploit simple and low-cost up-front total ordering of transactions to execute and schedule transactions much more efficiently. We show that deterministic execution enables low-overhead, highly-parallel scheduling mechanisms, that can address the performance limitations of existing database systems on modern hardware. Deterministic database systems are designed based on the assumption that applications can submit their transactions in one-shot prepared transactions, instead of multiple round-trips. Finally, we attempt to understand the fundamental reason for the observed performance differences between various consistency levels in database systems, and based on this understanding, show that we can exploit deterministic execution to provide strong consistency at a cost that is competitive with that offered by weak consistency levels.

High Performance Multi-core Transaction Processing via Deterministic Execution

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Jose Manuel Faleiro

Dissertation Director: Daniel J. Abadi

December, 2018

Contents

List of Figures

List of Tables

Acknowledgements

Thank you Daniel Abadi, for advising me during my PhD. You gave me the space to think independently while always being there to overcome roadblocks. Thank you for supporting my decision to spend the majority of my PhD in the Bay Area; without that support, I would have been hopelessly lost, away from my partner.

Thank you Mahesh Balakrishnan, for being an incredible mentor and friend. You taught me to uncompromisingly think through ideas end-to-end, to be humble and self-confident, and to be brutally honest with myself.

Thank you Phil Bernstein, for mentoring me during the summer of 2014 and serving on my dissertation committee. Your attention to detail and humility are truly astounding. Whatever clarity of discourse exists in this dissertation is due to your wonderful feedback.

Thank you Joe Hellerstein, for being a great mentor and collaborator while I spent the majority of my PhD in the Bay Area. Thank you for teaching me how to add panache to my reseach.

Thank you Wyatt Lloyd, for teaching me to be positive and optimistic, and think through a problem in all its depth. Thank you for graciously sitting through so many iterations of my job talk in early 2018.

Thank you Bryan Ford, for being an accommodating mentor during my first year in grad school.

Thank you Rebecca Isaacs and Paul Barham, for teaching me about multi-core hardware during the summer of 2013. The seeds of this dissertation were planted that summer.

Thanks to the many collaborators who put up with me; Abhinav Sharma, Kun Ren, Josh Lockerman, Chenggang Wu, Alex Thomson, Juno Kim, Stan Swidwinski, and Soham Sankaran.

Thanks to the friends who kept me sane; John Maheswaran, Mobin Javed, Yifan Wu, Rohit Nahar, Gaurav Paruthi, Adit Madan, Kaleem Rahman, Srinivas Eswar, and Pranav Saxena.

Thank you Nithya Sambasivan, for being the Yin to my Yang.

Thank you to my family, for your love and support through everything.

Chapter 1

Introduction

The last decade has seen dramatic changes in the hardware landscape across which server applications, such as database systems, are deployed. Most modern database systems derive their architecture and interfaces from the SystemR [52] and Ingres [154] research database systems pioneered in the 1970s. These systems were designed under hardware and workload assumptions that are no longer true. SystemR and Ingres provided a transaction processing interface that was also largely a function of 1970s workloads. For instance, one of the biggest use cases for transactions was interactive transactions, in which a human terminal operator would issue commands to the database.

Modern server systems are equipped with tens of CPU cores, hundreds of gigabytes of main memory, and low-latency durable storage devices. This environment is fundamentally different from the environments of paucity of main memory, high latency durable storage, and uniprocessors that the pioneering database systems were designed for. Not surprisingly, therefore, research from the mid-2000s onwards has repeatedly shown that the architectures, algorithms, and design principles of conventional database systems have been found wanting on modern hardware. For instance, in 2008 Harizopoulos et al. [94] found that a conventionally architected database system would spend the less than 4% of its cycles doing useful work directly related to transaction processing when deployed on a modern server equipped to hold the entire database in memory. In 2009, Johnson et al. [109] found that the multi-threading mechanisms in conventional database systems, designed for hiding the latency of writes to durable storage, were ill-equipped to exploit the hardware parallelism in the then upcoming (and now mainstream) multi-core processors.

Applications and workloads have also changed considerably since the pioneering systems of the 1970s. The biggest change has been the trend towards specialized systems designed for transaction processing, and a variety of analytics. While conventional systems would have to support both transactional applications and analytics applications, modern transaction processing systems can be specially designed to handle only transactional workloads. Conventional database systems provided applications with flexible interactive interfaces, in which applications were allowed to interact with the database over several round-trips in the course of a single transaction. Modern applications increasingly interact with storage systems via simple interfaces, such as the restricted interfaces of key-value stores or stored database procedures. A large class of applications therefore does not need the flexibility of conventional interactive transactions. Despite these application changes, however, the interface with which applications interact with database systems and the assumptions transaction processing systems make about applications have remained largely unchanged. Even though several modern database systems support a stored procedure interface (sometimes exclusively so), their internal transaction processing mechanisms do not exploit the restriction in interface to process transactions more efficiently.

One final important consideration in modern server applications is the scale at which they are deployed, and the impact of weak consistency at scale. Database systems provide applications with mechanisms to trade off performance for consistency via isolation levels. In practice, weak consistency is very widely deployed. Although weak consistency has become more prominent in the collective consciousness over the last decade, it has in fact been a fundamental part of database systems ever since their inception; in his Notes on Database Operating Systems, published in *1978*, Jim Gray noted that most systems at the time eschewed strong consistency because of its impact on performance [91]. This is as true today as it was in 1978. Although the vast majority of research on database isolation has focused on serializable consistency, the strongest level of consistency traditionally associated with database systems, serializability is far from widely used practice. Some database systems, such as Oracle and SAP HANA do not even provide serializable consistency, while others such as Microsoft SQL Server and PostgreSQL do not employ it by default. The lack of strong consistency in the database system punts the issue of consistency to the application; it is now up to applications to ensure that their execution appears strongly consistent. Writing applications that are resilient to weak consistency in the database system is notoriously error prone, and makes applications prone to serious bugs. For instance, two different Bitcoin exchanges in the recent past were exposed to security vulnerabilities due to their use of weak isolation. One of the exchanges – Flexcoin – had its entire reserves of coins stolen [2], while the second exchange – Poloniex – had 12% of its coins stolen [23]. At scale, weak consistency makes it more likely that latent bugs manifest themselves. (In the same way that failures are more likely to occur at scale.)

As a rough rule of thumb, a database system's state is equivalent to a serial execution of transactions[1]. Effectively, the total order of transactions determines the database system's state. Conventional systems are non-deterministic, in that the total order of transactions they arrive at depends on non-deterministic runtime factors such as the order in which the operating system schedules threads, deadlocks, system failures, and so forth. The non-determinism in transaction processing is necessary in order to support database systems' flexible interactive interfaces. While these interfaces give applications the flexibility to interact with the database over multiple round-trips, they fundamentally limit database systems' ability to efficiently schedule transactions because it forces database systems to account for a large space of possible application errors. For instance, the database must assume that clients can hang or fail in the middle of a transaction, that future database accesses in a currently executing transaction might conflict with existing transactions, that a client can arbitrarily rollback a transaction's effects at any point in its execution, and so forth.

These assumptions in turn limit database systems' ability to aggressively schedule transactions. For instance, the assumption that a transaction can potentially rollback its writes at any point in its execution prevents database systems from exposing a transaction's writes until the end of its execution (Chapter 6); the transaction's writes might be observed in the meanwhile, and these writes would correspond to data that never existed if they are rolled back. This delayed exposure of writes fundamentally limits the performance of strong consistency levels as compared to weaker ones. This is because weak consistency levels can often read stale data, and are thus robust to delayed exposure of records' latest values (Chapter 6).

This thesis proposes transaction processing mechanisms that are deterministic, in which the total order of transactions that determines the database state is determined prior to executing them. Conventional transaction scheduling protocols must guarantee that transactions are executed in a manner that is equivalent

1. Although technically true under serializable and strictly serializable consistency, Adya et al. generalized this notion to weaker consistency levels [42]

to some serial execution; they must constrain the execution of concurrent transactions so that it appears they executed serially. Deterministic database systems must do the opposite; given an predetermined total order of transactions, they must execute transactions concurrently in a manner that is consistent with the given total order.

While this reversal of concerns seems artificial, we show that the total order permits transaction scheduling mechanisms that impose low overhead and provide strong guarantees that are difficult to provide in conventional systems:

- **Lazy transaction evaluation. (Chapter 4)** We introduce a mechanism for *lazy* transaction execution, in which a transaction may be considered durably completed after only partial execution, while the bulk of its operations (notably all reads from the database and all execution of transaction logic) may be deferred until an arbitrary future time, such as when a user attempts to read some element of the transaction's write-set—all without modifying the semantics of the transaction or sacrificing ACID guarantees. Lazy transactions are processed deterministically, so that the final state of the database is guaranteed to be equivalent to what the state would have been had all transactions been executed eagerly. Our lazy transaction execution engine improves temporal locality when executing related transactions, reduces peak provisioning requirements by deferring more non-urgent work until off-peak load times, and reduces contention footprint of concurrent transactions.

- **Decouple reads from writes under serializable multi-version. (Chapter 5)**

  Multi-versioned database systems have the potential to significantly increase the amount of concurrency in transaction processing because they can avoid read-write conflicts. Unfortunately, the increase in concurrency usually comes at the cost of transaction serializability. If a database user requests full serializability, modern multi-versioned systems significantly constrain read-write concurrency among conflicting transactions and employ expensive synchronization patterns in their design. In main-memory multi-core settings, these additional constraints are so burdensome that multi-versioned systems are often *significantly* outperformed by single-version systems.

  We propose BOHM, a new concurrency control protocol for main-memory multi-versioned database systems. BOHM guarantees serializable execution while ensuring that reads *never* block writes. In addition, BOHM does not require reads to perform any book-keeping whatsoever, thereby avoiding the overhead of tracking reads via contended writes to shared memory. This leads to excellent scalability and performance in multi-core settings. BOHM has all the above characteristics without performing validation based concurrency control. Instead, it is pessimistic, and is therefore not prone to excessive aborts in the presence of contention. An experimental evaluation shows that BOHM performs well in both high contention and low contention settings, and is able to dramatically outperform state-of-the-art multi-versioned systems despite maintaining the full set of serializability guarantees.

- **Decrease the performance gap between strong and weak isolation levels. (Chapter 6)** To guarantee recoverable transaction execution, database systems permit a transaction's writes to be observable only at the end of its execution. As a consequence, there is generally a delay between the time a transaction performs a write and the time later transactions are permitted to read it. This *delayed write visibility* can significantly impact the performance of serializable database systems by reducing concurrency among conflicting transactions.

  Delayed write visibility stems from the fact that database systems can arbitrarily abort transactions at any point during their execution. Accordingly, we make the case for database systems which only abort transactions under a restricted set of conditions, thereby enabling a new recoverability mechanism, *early write visibility*, which safely makes transactions' writes visible prior to the end of their execution. We design a new serializable concurrency control protocol, piece-wise visibility (PWV) with the explicit goal of enabling early write visibility. We show that PWV can outperform serializable protocols by an order of magnitude and read committed by 3x on high contention workloads.

Chapter 2

Background

2.1 Concurrency control

A transaction processing database system must schedule concurrent transactions according to some well-defined high-level specifications, which determines the permissible set of interleavings between transactions. These specifications, termed isolation levels, specify permissible transaction interleavings in an implementation agnostic fashion. Each transaction submitted to the database is assigned a particular isolation level, by the user at submission time, and the database system then schedules the transaction in a manner that is consistent with the isolation level's specification. The scheduling mechanism employed by the database system is referred to as *concurrency control*.

Examples of concurrency control protocols include two-phase locking [78], optimistic concurrency control [115], and timestamp ordering [60, 143]. Various concurrency control protocols use different scheduling criteria to satisfy a particular isolation level. For example, two-phase locking proactively prevents conflicts between transactions by requiring that transactions acquire the appropriate locks before accessing a particular resource. On the other hand, optimistic concurrency control retrospectively prevents conflicts by allowing transactions to run unconstrained, and then validating that they did not conflict after they execute (taking necessary actions, such as rolling back transactions, in the event of conflicts). Each concurrency control protocol therefore permits different interleavings of transactions and has different sources of overhead.

2.1.1 Balancing protocol overhead and interleavings

The goal of a concurrency control protocol is to allow as many permissible interleavings of transactions as possible (as dictated by the isolation level specification), while at the same time imposing minimum overhead in making scheduling decisions. More permissible interleavings between transactions directly translates to more parallelism because it means that more CPU cores can be kept utilized, an important goal for software systems on modern hardware. The goal of producing as many interleavings as possible between transactions can be at odds with keeping scheduling overhead low; a sophisticated algorithm might be able to produce more interleavings between transactions, but the overhead in making these scheduling decisions might also increase with increasing sophistication.

For example, Cahill et al. proposed a concurrency control protocol for snapshot database systems called serializable snapshot isolation (SSI) [64]. The key insight SSI exploits is that if serializability (an isolation level) is violated in snapshot database systems, then a consecutive sequence of read-write dependencies between concurrent transactions must have occurred. SSI detects such consecutive sequences of read-write dependencies, and breaks them by aborting one of the involved transactions. Importantly, while preventing a consecutive pair of read-write dependencies is sufficient to guarantee serializability, it is not necessary. The SSI algorithm is thus susceptible to false positives, in that it may conservatively abort a transaction even under situations where serializability is not violated. Subsequently, Revilak et al. proposed Precisely Serializable Snapshot Isolation (PSSI) [147], which only aborts transactions if their execution actually violates serializability. However, this check involves an expensive cycle detection algorithm, which is more expensive to implement.

The differences between SSI and PSSI illustrate the tradeoff between balancing protocol overhead and interleavings. While SSI theoretically prevents certain valid interleavings of transactions, its core algorithm is lighterweight than PSSI's. Any new concurrency control protocol must be aware of this tradeoff, and appropriately balance its overhead with its ability to maximize interleavings of transactions.

Minimizing protocol overhead is especially important today given modern deployment environments, in which databases can cache larger working sets (or even the entire data set) in main-memory and reduce the cost of cache misses due to faster durable storage devices such as solid-state drives and non-volatile RAM.

Until the mid-2000s the cost of executing a transaction was dominated by high-latency IO, making it possible to hide or ameliorate concurrency control overheads. With low latency IO and networks, it is increasingly difficult to hide concurrency control overhead. In 2008, Harizopoulos et al. found that overhead in concurrency control protocols are higly burdensome. They found that transactions spend about 16% of their time on protocol overhead under two-phase locking when running Shore, a disk-oriented research database system entirely in memory on a single core server. [1]

While the reduction of concurrency control overhead has traditionally been focussed on reducing protocol and meta-data complexity, modern concurrency control protocols also have to contend with concerns on modern hardware. In 2017, Ren et al. found that two-phase locking overhead takes between 60% (under low contention) and 97% (under high contention) of a transaction's execution time on a large-scale 80 core multi-core server. This overhead is primarily due to synchronization and cache coherence on modern hardware [144]. The remainder of this section discusses multi-core hardware overhead in more detail.

2.1.2 Overhead on multi-core hardware

The maximum throughput that a database system is able to achieve is dependent on many factors, from the hardware on which it runs to the particular implementation details of the database software. While most of these factors can be overcome by spending more money, on better hardware or more skilled software developers, throughput will always be fundamentally limited by the presence of *contended operations* in a workload.

The definition of a "contended operation" varies depending on transactions' requested isolation levels, the ability of the database to prevent reads from conflicting with writes via multiversioning and the semantic commutativity of operations. Nonetheless, unless no isolation whatsoever is required, there will always be certain operations that cannot be executed concurrently and the presence of many of these contended operations in a workload will necessarily limit throughput. Thus, adding more processors to a system, which enables more transactions to be pro-

1. In fact, Harizopoulos et al. found that the database system would spend less than 4% of their cycles actually executing logic associated with transactions. In addition to concurrency control, other factors, such as database buffer pool/main-memory management and logging contributed to large chunks of overhead.

cessed in parallel, only increases throughput if the additional transactions that can be run do not conflict with the existing transactions that are currently running.

For decades, database systems had been designed for single processor machines. These conventional database architectures were ill-suited for the abundant parallelism in multi-core hardware. As a consequence, they could not achieve scalable throughput across the entire spectrum of transactional workloads. Particularly problematic was the fact that conventional database architectures were not able to scale throughput on *low contention* workloads, even though low contention workloads have no fundamental limit to scalability. To address this gap between hardware and database software, most work on multi-core database systems has focused on eliminating scalability bottlenecks in these systems' design [107, 112, 159]. As a result of this research, today's state-of-the-art systems can achieve close to linear scalability in transactional throughput on low contention workloads.

Unfortunately, as recently demonstrated by Yu et al., multi-core database systems continue to be plagued by performance problems on *high contention* workloads [172]. For the fundamental reason described above, it is impossible to achieve linear scalability in high contention workloads; the throughput of a database system should taper as cores are added under high contention. However, the *actual* shortfall relative to linear scalability on high contention workloads is much worse than what is theoretically required by the nature of the contention. In some cases, in fact, throughput actually *decreases* as more cores are added. The problem is that the overhead of managing transactions, particularly that of concurrency control, is proportional to the amount of workload contention. At high levels of contention, concurrency control overhead takes up a non-trivial fraction of each transaction's total execution time. As a consequence, modern multi-core database systems achieve nowhere near the theoretical performance determined by the achievable concurrency in high contention workloads.

This poor performance under high contention is because database systems tend to assign responsibility for a particular transaction to a single thread [97]. This *conflates database functionality*, which leads to poor instruction and data cache locality [95]. Worse, conflated functionality causes workload contention to directly impact physical contention in the database system.

A transaction's thread manages both the execution of the transaction's logic and the necessary interactions with the concurrency control module of the database system (e.g. a lock manager or the shared data structures needed for OCC validation). Several such threads concurrently execute on a single multi-core system. These concurrently executing threads make requests of the same concurrency control module[2]. This design pattern of multiple threads globally sharing a single concurrency control module can lead to severe scalability bottlenecks. We discuss these bottlenecks in this section.

Synchronization overhead. We first discuss the overhead associated with synchronization on concurrency control meta-data. In order to control the interleaving of concurrent transactions, any concurrency control protocol must associate some meta-data with the database's logical entities. The meta-data used is protocol dependent. For example, locking protocols use a hash-table of lock requests on records [89], while optimistic and multi-version protocols associate timestamps with records [117,159]. As part of the concurrency control protocol, several concurrent threads may need to read or write the meta-data associated with a particular database object. Concurrent threads must therefore synchronize their access to concurrency control meta-data. Note that synchronization is *not* implementation dependent; instead, it is intrinsic to any concurrency control protocol whose meta-data can be read or written by any database thread. Since meta-data is associated with database objects (such as records), contention for concurrency control meta-data is directly affected by workload contention; if a particular database record is popular, then threads will need to frequently synchronize on that record's meta-data. Unfortunately, on modern multi-core hardware, contention significantly degrades the performance of *atomic instructions* [62,79,80,159]. These atomic instructions – such as *fetch-and-increment* and *compare-and-swap* – are the basic building blocks of both latch-based and latch-free algorithms. Thus, under contention, both classes of algorithms are susceptible to severe performance degradation.

Data movement overhead. In addition to synchronization on concurrency control meta-data, another source of overhead in conventional database architectures is the *movement* of this meta-data across multiple cores. In order to access an object's

2. The discussion that follows assumes a shared-memory architecture where all threads have access to the same shared memory where the concurrency control data structures sit. It should be noted that some database systems, such as H-Store [155] and Hyper [113], use a shared-nothing architecture across threads. Such systems do not suffer from the overheads associated with conflated functionality discussed in this section. However, they introduce other performance problems — namely agreement protocols across threads necessary to handle transactions that access data in multiple separate partitions.

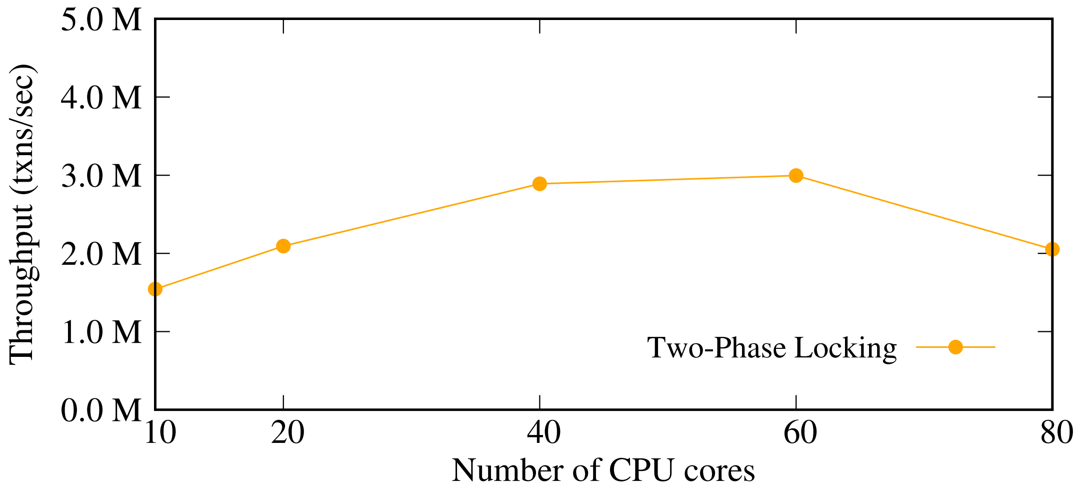


Figure 2.1: Scalability of read-only transactions under two-phase locking on a high contention workload.

meta-data, a thread must move the memory words corresponding to the meta-data into its CPU core's local cache. As multiple threads request access to a particular object's meta-data, the memory words corresponding to the meta-data are moved between cores. The movement of data between a machine's cores occurs because multiple threads are allowed to read or write the data. If a thread requires access to a latch-protected data structure, the thread must first acquire the latch and then move the data structure to its core. Only when these two steps complete can the thread actually access the data structure. Since the latch is acquired first, it is held for the time it takes to move the data structure. As a consequence, data-movement extends the duration for which latches are held. In the presence of contention, the increase in latch hold times can contribute to a decrease in concurrency.

In order to validate the deleterious effects of synchronization and data movement overhead, we ran a simple experiment to measure the scalability of short read-only transactions under two-phase locking on a high contention workload. Since transactions are read-only, the workload is conflict free (despite the presence of contention). Figure 2.1 shows the results of the experiment. Two-phase locking is unable to scale beyond 40 cores despite the absence of conflicts and surprisingly *decreases* in performance. The inability to scale is due to synchronization and data movement overhead. The source of synchronization overhead is two-phase locking's use of atomic instructions on contended memory words to manipulate the table of lock requests. Data movement overhead is a consequence of multiple cores manipulating the same list of lock requests (due to multiple cores requesting read locks on the same records).

Instruction and data cache pollution. If a single thread executes both concurrency control and transaction logic, then the thread's CPU core must cache data and instructions corresponding to each of these two functions. The data and instructions corresponding to concurrency control, and transaction logic thus compete for a single core's cache. Concurrency control cache-lines therefore evicts transaction logic cache-lines, and vice-versa. This has the overall effect of increasing the duration of each transaction, which in turn decreases overall throughput.

All three reasons for performance degradation — synchronization overhead, data movement overhead, and cache pollution — increase transactions' execution times. Not only does this reduce throughput by occupying system resources for longer periods of time, but the throughput reduction is compounded by increasing transaction time. This is particularly harmful in high contention settings because conflicting transactions either have to block (in pessimistic schemes) or abort (in optimistic schemes). The longer it takes to execute a transaction, the higher the probability that a later conflicting transaction will abort or block behind the original transaction.

2.1.3 Isolation

Database systems allow applications to tradeoff consistency for performance via isolation levels. Strong isolation levels, such as serializability, permit fewer interleavings between conflicting transactions, which provides strong consistency at the expense of concurrency. Weak isolation levels, such as read committed, permit more interleavings between conflicting transactions, allowing transactions to observe inconsistent states to increase performance.

Section 2.1.1 described the performance implications of a concurrency control protocol in the context of a particular isolation level; given a particular isolation level, concurrency control protocols differ in their overhead and permitted transaction interleavings. This section describes the role of different isolation levels on the performance of transaction processing database systems.

Serializability is arguably the most well known, and studied, isolation level provided by database systems. Serializability imposes the restriction that transactions are executed in a manner that is equivalent to some serial execution of transactions. Bernstein et al. [58] formalized serializability in terms of permissible orderings of transactions' constituent reads and writes, expressed as dependency graphs. Adya et al. [42] extended the dependency graph formalism to capture widely-used isolation levels. By definition, weak isolation levels impose fewer restrictions on transactions than serializability. These looser requirements mean that the implementation of a particular isolation level needs to impose fewer restrictions on transactions. For instance, under the popular read committed isolation level, transactions are allowed to read any committed value of a record, which can be arbitrarily stale. Under serializability, however, transactions must generally observe the latest value of each record. As a consequence, serializable concurrency control protocols must carefully constrain the execution of transactions whose reads and writes conflict, while read committed protocols can decouple conflicting reads and writes. Consider a scenario where record x is first written by transaction T_0, and next written by transaction T_1 (T_0 precedes T_1). If a later transaction T_2 reads x, then under serializability, T_2's read *must* return the value written by T_1. In contrast, read committed allows T_2 to read either of T_0 or T_1's writes.

The implementations of weak isolation levels typically permit more interleavings between transactions than those of strong isolation levels because their specifications impose fewer restrictions on transactions. Furthermore, weak isolation levels typically impose lower overhead than strong isolation levels, simply because the weak isolation specifications have fewer and simpler criteria they must guarantee adherence to. In practice, these differences have translated to significantly better performance. The majority of database systems employ weak isolation levels, such as read committed, by default. Some systems, such as Oracle and SAP HANA, do not even provide serializable isolation, while others, such as Microsoft SQL Server and PostgreSQL do not provide it by default.

2.2 Implications for deterministic execution

2.2.1 Transaction processing overhead

Given this background, this thesis explores the design of strictly serializable concurrency control protocols via deterministic execution. One of its key contributions is to show that it is possible to aggressively interleave transactions with minimal overhead. By relying on a pre-determined total order of transactions, deterministic transaction execution can guarantee serializability by simply ensuring that it executes transactions in a manner that is consistent with this total order. The total order effectively serves as an execution plan; rather than requiring threads to synchronize and coordinate their accesses to database records, deterministic systems can create an abstract, partially-ordered schedule of transactions consistent with the total order prior to executing them. Transaction processing is thus broken up into two phases, a planning or scheduling phase, and an execution phase. Our mechanisms for lazy transaction evaluation (Chapter 4), multi-version concurrency control (Chapter 5), and transaction decomposition (Chapter 6) all use this design principle.

Separating transaction scheduling from execution eliminates the need for shared concurrency control meta-data. Concurrency control is managed by a dedicated set of threads, and each thread is responsible for concurrency control on a mutually exclusive partition of the database. Concurrency control threads do not share data, and can therefore process transactions independently. This design principle therefore avoids the need for shared-memory synchronization to determine valid transaction schedules.

Furthermore, scheduling threads only determine the order in which transactions must execute; transactions' actual execution can occur in a shared-everything environment. By decoupling shared-nothing scheduling from (potentially) shared-everything transaction execution, we can avoid the deleterious performance consequences of both shared-nothing and shared-everything systems (multi-partition transactions, and shared-memory synchronization, respectively).

Finally, separating transaction scheduling from execution enables a lightweight adaptive mechanism to execute transactions. The vast majority of database systems use the thread-to-transaction model of execution [97]. By binding transactions to a specific thread, database threads are often subject to excessive context-switching overheads, especially under contention. The execution mechanisms described in this thesis do not use the conventional thread-to-transaction model of execution. Instead, we model a schedule of transactions as an explicit dependency graph. Database threads process transactions by simply crawling this graph of transactions. We have found that dependency-graph-based scheduling completely eliminates the cost of context-switching database threads, and automatically limits the number of active threads based on the degree of contention in a workload.

2.2.2 Bridging the performance gap across isolation levels

Although there exist fundamental differences in the permitted interleavings between various isolation levels, this thesis shows that the biggest differences in the actual performance of various isolation levels are *not* due their specifications, but often due to their implementations. We show that these limitations in implementation cannot be avoided in conventional database systems due to to their use of non-deterministic concurrency control. At a high level, conventional transaction processing must make worst-case assumptions about future actions of transactions, such as their potential to conflict with existing transactions, and their potential to rollback their writes. These worst-case assumptions force conventional systems to use heavy-handed mechanisms, which are sufficient but unnecessary.

Our research on multi-version concurrency control (Chapter 5) shows that state-of-the-art mechanisms for serializable multi-version concurrency control often provide no additional concurrency benefits than their single-version counterparts [117, 138]. As a consequence, serializable protocols cannot effectively exploit multi-versioning to reduce read-write conflicts among transactions (even though, in principle, multi-versioning can allow conflicting reads and writes to execute in parallel by directing reads to pre-existing versions of records, while allowing writes to create new versions). On the other hand, weaker isolation levels, such as snapshot isolation can completely decouple conflicting reads and writes, and consequently significantly outperform serializability in multi-version database systems. We show that the lack of read-write concurrency in serializable protocols is not fundamental, and can be circumvented by determining a valid schedule of transactions prior to their execution (Chapter 5).

Similarly, Chapter 6 shows that the biggest contributor to the performance gap between of serializable and weak isolation levels is the recoverability mechanisms employed by conventional database systems [81]. In particular, to prevent the writes of rolled back transaction from being observed, conventional database systems prevent a transaction's writes from being observed until the end of their execution (using mechanisms such as early lock release [71, 87] or using strict two-phase locking [58]). However, under serializability, this mechanism forces transactions to wait for longer durations under conflict-heavy workloads because transactions must (as a general rule of thumb) observe each record's latest values. Under weak isolation levels such as read committed, however, transactions can observe arbitrarily stale record values, and are therefore unaffected by the recoverability mechanism. We show that deterministic execution can enable a new recoverability mechanism (which we refer to as piece-wise visibility) to allow a transaction's writes to be observed even if its logic has not yet executed in its entirety.

Chapter 3

Primer on multi-core synchronization

3.1 Introduction

Access to large-scale multi-core servers is becoming increasingly democratized. For instance, it is now possible to now obtain access to virtual machine instances consisting of 64 physical CPU cores on Amazon EC2 [38], while large multi-core servers have been available on the market for several years. This trend has led to significant research interest in database system architectures that effectively exploit parallelism on a single machine.

Modern database systems exploit parallelism in multi-core hardware by employing some form of multi-threading [1]. The threads in a system exchange information via shared data structures. If multiple threads are allowed to concurrently access to shared data structure without any coordination, then the data structure is susceptible to corruption due to race conditions. In order to prevent race conditions, threads must *synchronize* their accesses to shared data structures [101].

One perceived issue with conventional DBMS architectures is their widespread use of locks or, as they are better known in the database systems community, latches.[2] Concurrent systems typically use latches. In order to protect the integrity of shared data structures within the database system, latches ensure that only one thread at a time can modify a shared data structure. However, for the

1. Note that both multi-processing and multi-threading are equivalent for the purposes of this discussion.

2. This chapter's use of the term "latch" or "lock" is different from "logical lock". Latches protect shared data structures from concurrent modification by multiple threads, while logical locks are used to guarantee correct interleavings among transactions in DBMS concurrency control.

same reason, because only one thread at a time can acquire a latch, latch-based algorithms are thought to be susceptible to performance problems at scale; if a thread is delayed while holding a latch, then no other thread in the system can acquire the same latch for at least the duration of this delay. In contrast to latch-based algorithms, *latch-free* algorithms provide strong theoretical progress guarantees which, at minimum, ensure that no thread is blocked due to the delay or failure of other threads [99,100,102]. Several research papers have therefore made the argument that latch-free algorithms scale better than or outperform latch-based algorithms [72,103,119,121].

This chapter argues that latch-free algorithms' theoretical guarantees are mostly *irrelevant* to performance and scalability on multi-core hardware. The most important factor that influences the scalability of a synchronization mechanism is its ability to avoid contention on global memory locations, irrespective of whether the mechanism is latch-based or latch-free. We argue, and show experimentally, that when latch-free algorithms' theoretical guarantees do become relevant, the performance problems in latch-based algorithms are not fundamental to their use of latches. Instead, the performance problems arise due to inefficient allocation of resources, such as using more OS threads than available CPU cores.

We highlight the oft under-appreciated fact that latch-free algorithms generally require idiosyncratic memory management mechanisms. Latch-free memory management mechanisms add complexity and overhead relative to latch-based algorithms. These mechanisms are often ignored or omitted in published descriptions of latch-free algorithms, and have consequently been a source of bugs in implementations of these algorithms [24,36].

We perform a set of microbenchmarks that make an apples-to-apples comparison between a latch-free algorithm and three classes of latch-based algorithms. We show that while the latch-free algorithm outperforms simple busy-waiting latches, the results in comparison to more sophisticated backoff latches are mixed. Furthermore, we find that the latch-free algorithm is never able to outperform a scalable queuing latch.

This chapter is not intended to make a case *against* latch-free algorithms. Instead, we show that the argument for the superior scalability of latch-free algorithms is more nuanced than a scan of the current literature on multi-core database architectures would suggest (often exacerbated by the fact that papers compare latch-free algorithms against inefficient coarse-grained latching algorithms). We highlight the various factors that influence the performance and complexity of synchronization primitives on multi-core hardware to inform future research and database system implementations.

3.2 Latch-free algorithms

Latch-free algorithms provide strong theoretical liveness guarantees that distinguish them from latch-based algorithms. These guarantees pertain to the progress that threads can make during concurrent execution. While several progress guarantees exist [99, 100, 102]; at a minimum, all of these guarantees ensure that no thread is blocked due to the delay or failure of other threads. Latch-based algorithms make no such guarantees. Latch-based algorithms guarantee correct concurrent execution of threads via *mutual exclusion*; in order to access a shared data structure, a thread acquires a latch, which prevents other threads from simultaneously accessing the data structure.

This section makes the case that the scalability and performance of a synchronization mechanism is dependent on its avoidance of contention on global memory locations (Section 3.2.1). This is far more important than any progress guarantees made by the mechanism. Furthermore, the progress guarantees of latch-free algorithms can lead to *increased* overhead due to memory management issues not present in latch-based algorithms (Section 3.2.2) and other areas of additional complexity (Section 3.2.3).

3.2.1 Scalability

Synchronization performance

Modern database systems exploit parallelism in multi-core hardware by employing some form of multi-threading [3]. The threads in a system exchange information via shared data structures. If multiple threads are allowed to concurrently access to a shared data structure without any coordination, then the data structure is susceptible to corruption due to race conditions. In order to prevent race conditions, threads must *synchronize* their accesses to shared data structures [101].

Latches are an explicit form of synchronization, which are used to ensure that only one thread can obtain exclusive access to a data structure. A latch typically

3. Note that both multi-processing and multi-threading are equivalent for the purposes of this discussion.

consists of a single word in memory, whose value indicates whether or not a thread currently holds the latch. Threads use various combinations of reads and writes to acquire a latch (a specific combination yields a particular latching algorithm). In a latch-free algorithm, threads use atomic instructions to ensure that they correctly update or read a shared data structure. These atomic instructions are executed on one or more words in shared memory. Threads in both classes of algorithms thus read and write one or more shared words in memory to synchronize their access to shared data structures. The performance of both classes of algorithms is therefore tied to the performance of concurrent reads and writes against a single word in memory.

There are two rules of thumb that determine the performance of concurrent reads and writes to a particular memory location on multi-core hardware [50, 62, 70, 108]. First, atomic instructions, such as `cmp-and-swp` and `xchgq`, that write (or attempt to write) a particular word in memory are executed serially. Thus, if several cores concurrently attempt atomic update instructions on a particular word in memory, the time taken to process these instructions is proportional to the number of writing cores. Second, the new value of a recently written word in memory is serially propagated to reading cores. That is, if a core writes the value of a word in memory, and several other cores read the value of the word, the new value of the word will propagate to the reading cores in time proportional to the number of readers.

Several latching implementations are unscalable under contention because they interact badly with the two characteristics of multi-core hardware above. In the simplest implementation of a latch, cores repeatedly attempt to atomically test-and-set the value of a word in memory from 0 to 1 (performed via an `xcghq` on x86 architectures). An atomic test-and-set unconditionally sets a memory word to a specified value and returns the previous value of the word. If a core's test-and-set returns 0, then it means that the value of the word successfully transitioned from 0 to 1 due to the core's test-and-set. If a core's test-and-set returns 1, then it means that the previous value of the word was 1, and hence that another core has already acquired the latch. Performing test-and-sets in a tight loop puts pressure on the memory controller associated with the latch word and increases traffic across NUMA nodes. This can delay non-conflicting memory requests by cores not executing the test-and-set, including the core executing the critical section. Furthermore, to release the latch, the latch holder must atomically change the value of the latch word from 1 to 0. This write competes with the test-and-sets performed by threads trying to acquire the latch. This delays releasing the latch, which effectively increases the length of the critical section.

To rectify these issues, Segall and Rudolph proposed an enhancement to test-and-set latches; instead of repeatedly performing a test-and-set on the latch word in a tight loop, cores attempting to acquire the latch could first *read* the value of the word, and only perform a test-and-set if the tested value is 0 [148]. This latch was termed a test-and-test-and-set (TATAS) latch. TATAS latches allow cores to spin on locally cached copies of the latch word while another core holds the latch. TATAS latches seem to address both issues with simple test-and-set latches; since cores first spin on locally cached values of the latch word, they avoid generating pressure on memory controllers and NUMA interconnects. In addition, latch release does not have to compete with test-and-set requests by cores attempting to acquire the latch.

Unfortunately, the above benefits only apply to lengthy critical sections. Anderson showed that if a critical section is relatively short, then the performance of the TATAS latch is dominated by transient behavior which occurs while the latch changes ownership across cores [50]. In particular, if a core C_0 releases the latch, then several cores C_1, C_2, ..., C_m will notice this change. The first of these cores to perform a subsequent test-and-set will then take ownership of the latch (say C_1). However, this ownership change will not prevent C_2, ... C_m from performing unsuccessful test-and-sets. This causes a transient flood of test-and-sets requests. Only when every one of C_2, ..., C_m has finished performing an unsuccessful test-and-set does the system quiesce. If this time to quiesce is comparable to the critical section length, then TATAS latches suffer from the same scalability problems as simple test-and-set latches.

The underlying problem with test-and-set and TATAS latches is that threads spin on a single *global* memory location. Spinning on a global location can cause serious scalability bottlenecks in latching algorithms that perform atomic modifications or reads on global locations. There exist two mechanisms to avoid spinning on global locations: backoff mechanisms and scalable latching data structures.

Backoff mechanisms separate each atomic modification or read by some number of `noop` instructions. The backoff between successive iterations is often increased via an exponential distribution [50]. Another commonly used backoff mechanism is to rely on the operating system to deschedule threads. This backoff mechanism is used in the GNU C library's Pthread mutex implementation [11].

Scalable latching data structures in which threads spin on thread-local or core-local data structures. Spinning on local data structures prevents the cost of spinning operations degrading with core counts. The most famous example of such a latching algorithm is the MCS latch [127]. The MCS latch constructs an explicit queue of threads waiting to acquire the latch. Each thread has an associated queue node, and this queue node is appended to a queue using an atomic operation (nodes are enqueued using a single `xchgq` instruction). The queue node at the beginning of the queue corresponds to the current latch holder. If a thread's appended queue node is not the first node in the queue, the thread spins on a flag in its local queue node. When the latch holder completes its critical section, it sets the flag in the next queue node. The corresponding thread then notices this change and begins executing the critical section. MCS latches avoid the scalability bottlenecks of conventional latch implementations because threads do not read or write a shared memory location in a loop; when the latch changes ownership, a single thread writes the next thread's queue node flag, and each thread spins on its local queue node's flag. In contrast, conventional latching implementations permit multiple threads to write to a single global memory location (via test-and-sets) in a loop. They are subject to slow downs because these test-and-set requests are serialized, and induce cache coherence traffic to invalidate and reload cache lines on cores that read the value of the latch word [50, 62, 70, 108].

Unlike latch-based algorithms, threads in latch-free algorithms never obtain mutually exclusive access to a shared data structure. Threads in a latch-free algorithm only use atomic instructions to make their writes atomic. In the vast majority of latch-free algorithms, threads *speculatively* read shared state, perform some local computation based on this speculative read, and attempt to atomically "commit" the computation based on the speculative read [96, 103, 121, 131]. In the time between the thread's speculative read and attempt to commit, another thread may have invalidated the thread's read due to a conflicting update. Threads typically use the `cmp-and-swp` instruction to validate that the shared state did not change values.

The `cmp-and-swp` instruction takes three arguments, the address of a word in memory, the value that the word is expected to contain, and a new value. `cmp-and-swp` checks that the word's value is equal to the old value, and if so, swaps the old value with the new value. If the word's value is not equal to the old value, the instruction leaves it unchanged.

Speculative latch-free algorithms are prone to the same scalability bottlenecks as latching algorithms. If updates fail often due to contention, then threads will repeatedly retry the operation. The repeatedly retried `cmp-and-swp` instructions are serialized, and lead to slowdowns because they are executed by threads that successfully speculate, as well as those whose speculation fails. These serialized instructions causes failures to slow down the successes. As a consequence, the "conflict window" of a speculative operation effectively increases. If the increase in the "conflict window" is comparable to the length of the speculative computation, then this effect is akin to increasing critical section size in TATAS latches.

Due to the optimistic nature of many latch-free algorithms, and the well-known costs of optimism (such as copying overhead — see Section 3.2.2), it is tempting to conclude that the differences between speculative latch-free and non-speculative latch-based algorithms are analogous to the differences between optimistic and pessimistic concurrency control in database systems [115]. However, this chapter aims to show that the *opposite* is closer to the truth. Speculative latch-free algorithms and latch-based algorithms are more alike than different because both classes of algorithms perform updates to shared memory locations. As a consequence, under contention, both classes of algorithms are governed by the same underlying hardware performance characteristics. In contrast, the differences between optimistic and pessimistic concurrency control in database systems arise due to their contention handling mechanisms — pessimistic concurrency control blocks the execution of a transaction in the presence of another conflicting transaction, while optimistic concurrency control aborts transactions upon detecting conflicts. Optimistic and pessimistic concurrency control are thus governed by a different set of tradeoffs; optimistic concurrency control wastes resources in a fully loaded system under high contention, while pessimistic concurrency control produces unnecessarily conservative schedules in an under-loaded system under low contention [45].

Moreover, the techniques used by scalable latching implementations cannot be directly used by latch-free implementations. Scalable latching implementations effectively determine the order in which threads can take ownership of a latch a priori. If a thread is delayed after having determined its priority, then every later thread of lower priority is delayed. However, latch-free algorithms must guarantee that the delay or failure of a particular thread *never* prevents other threads from making progress. Hence, pre-determining the order in which threads execute a critical section is incompatible with latch-free algorithms's theoretical guarantees, and can, at best, only be used as an auxiliary contention handling mechanism; when using pre-determined thread priorities a latch-free algorithm must have a way to time out of the pre-determined order and fall back to using `cmp-and-swp` operations [73].

Scheduling requests

Database systems use the notion of *multi-programming levels* (MPLs) to determine the maximum number of requests that can simultaneously execute. Database systems typically implement MPLs by assigning each request to an abstract *DB worker*, which corresponds to the execution context of the request within the database. DB workers are then mapped to an operating system execution context, such as a process or thread [97]. The choice of mapping from DB workers to OS contexts can have a significant impact on performance.

If the number of OS contexts exceeds the number of available CPU cores, then at least two contexts will be multiplexed over a single CPU core. The scheduling of OS contexts is handled by the operating system, which assigns a fixed time slice to each context, and preempts a context when its slice expires. Despite proposals for workload-aware schedulers [51], operating systems are usually unaware of whether preempted contexts have acquired latches. A context may therefore be preempted *while* it is holding a latch.

Most (but not all) database systems assign each DB worker to a single OS context (by either creating a new context or maintaining a pool of free contexts), and rely on the OS to timeslice contexts on CPU cores [97]. In addition, database systems traditionally use MPLs far higher than the number of available CPU cores. They therefore typically have more DB workers than available CPU cores. In database systems in which there is a 1:1 correspondence between DB workers and OS contexts (including most widely-used database systems), the number of OS contexts significantly exceeds the number of available CPU cores. This can lead to thrashing due to latch holder preemption.

In this traditional database system architecture, the progress guarantees of latch-free algorithms are extremely valuable because a preempted context will never block or delay the execution of other contexts in the system (Section 6.2). Prior research has therefore (correctly) advocated that latch-free algorithms offer a "drop-in" solution to the preemption problem.

However, the root cause of the preemption problem is *not* database systems' use of latches, but rather that database systems use more OS contexts than available CPU cores and their reliance on an OS with insufficient knowledge about user-level synchronization to schedule these contexts. There is no fundamental reason that forces database systems to implement multi-programming by assigning requests to unique contexts. Instead, a database system could itself implement a scheduling mechanism in user-space without relying on OS support [97]. This scheduling mechanism could ensure that it never uses more contexts than available CPU cores. Several research prototypes and new main-memory DBMS products already use this processing model [80,95,138,144,155,157,159]. Furthermore, database systems have a long tradition of implementing user-level scheduling of contexts (via DBMS threads) in environments where OS support for multi-processing was non-existent or inefficient [153].

Researchers in the software transactional memory (STM) community have also made the case that a system should limit the number of contexts it uses to the number of available CPU cores [77]. Early STM algorithms were designed to be non-blocking or wait-free so as to be robust to unexpected sources of delay beyond the control of the application (such as thread preemptions and page faults) [86]. However, more recent STM algorithms forego non-blocking and wait-free synchronization, and instead use latches to synchronize conflicting transactions [74,75,77]. Latches simplify STM algorithms and permit the use of important optimizations, such as in-place updates, which latch-free algorithms preclude. The next section discusses some of these issues in detail.

3.2.2 Memory management

Latch-based algorithms do not permit a thread access to a data structure if one or more conflicting threads are concurrently accessing the data structure. In contrast, due to their strict liveness guarantees, latch-free algorithms cannot restrict a thread from accessing a data structure due to the presence of conflicting threads. If the conflicting thread is delayed, then the restricted thread cannot make progress; which violates latch-free algorithms's liveness guarantees [99, 100, 102]. As a consequence, latch-free algorithms must permit threads *unrestricted* access to a data structure. This requirement has subtle implications on latch-free algorithms's design.

Copying overhead

In a latching algorithm, threads acquire latches to update shared data structures. A thread that has acquired a latch is guaranteed mutually exclusive access to the data structure protected by the latch. The thread can therefore perform in-place updates on the data structure. These updates may temporarily make the data structure inconsistent with respect to program invariants [39]. These inconsistencies are safe because the latch prevents other threads from accessing the data structure, and hence noticing temporary inconsistencies due to in-place updates.

In contrast, latch-free algorithms must permit a thread unrestricted access to a data structure, even while other threads attempt conflicting reads or writes. Threads must always be permitted unrestricted access to a data structure because of the stringent progress guarantees latch-free algorithms provide. To allow threads to make progress regardless of the presence of conflicting threads, the state of the data structure must *always* be consistent. As a consequence, to update a complex data structure, threads must make a copy of a data structure (or a portion of a data structure), and perform their updates against this local copy. These updates are made visible to other threads via an atomic instruction [48, 92, 102, 123].

Latch-free algorithms that perform updates on complex data structures must therefore pay the extra cost of copying a portion of a data structure. Furthermore, if the algorithm in question is speculative, then the cost of copying the data structure extends the duration of the conflict window in which the update has a chance of failing (Section 3.2.1). Finally, operating on copies of data structures can lead to worse cache utilization than in-place updates.

Garbage collection

Since latch-free algorithms permit multiple threads to simultaneously operate on an object, if an object is deleted by a thread, then its memory cannot be immediately freed to the operating system or allocator. This is because one or more threads may still be accessing the deleted object. Latch-free algorithms on dynamic data structures therefore typically use a form of deferred memory reclamation; such as deferred memory reclamation include hazard pointers [129] and epoch-based reclamation [126].

There are two problems associated with deferred memory reclamation. First, they impose extra overhead to determine when an object can be safely reclaimed. While some techniques, such as the epoch-based mechanism used in read-copy-update [126], have very low overhead, the choice of reclamation technique is not independent of the algorithm [103].

Second, and more importantly, deferred memory reclamation cannot be used as a black box, in the way that conventional memory allocators are used. Instead, memory reclamation logic is *algorithm dependent*, and therefore entangled with the implementation of a latch-free algorithm. For instance, the addition of correct memory reclamation to Michael and Scott's lock-free queue [131] requires non-trivial changes to the algorithm itself [129].

Memory re-use

If a thread uses `cmp-and-swp` instructions for correctness, it may miss concurrent updates by other threads. Consider the following sequence of events. Thread T reads the value A from a word W in memory. Thread T' changes the value of W from A to B, and then back to A. If T then attempts to `cmp-and-swp` W based on its earlier read, it will succeed despite the fact that W's value changed twice: from A to B, and back to A. This behavior, known as the *ABA problem*, can lead to subtle bugs if correctness depends on the the fact that no intervening updates occurred between T's read and its `cmp-and-swp` [101].

The ABA problem typically occurs in latch-free algorithms on pointer-based data structures, such as linked-lists and queues. For instance, consider a latch-free implementation of a sorted linked-list [96]. To insert a new node with value 7 (N_7) in the linked-list, a thread traverses the list until it finds an appropriate pair of adjacent nodes. Suppose the pair of adjacent nodes contain values 5 and 9 (N_5 and N_9). The thread performing the insertion sets N_7's next pointer to reference N_9. Next the thread attempts to atomically set N_5's next pointer to N_7 while validating that N_5 still points to N_9 using a `cmp-and-swp` instruction. However, if N_5 is deleted by another thread and then re-inserted with new value 8, N_7's `cmp-and-swp` will still succeed. This is because N_7's `cmp-and-swp` finds that N_5's next pointer still points to N_9. However, this insertion renders the linked-list inconsistent because nodes are no longer sorted. In the above example, the unfortunate sequence of events occurs because N_5 is re-used between the time N_7's thread performs a read and `cmp-and-swp` [101]. Preventing the ABA problem requires a mechanism that makes freed memory available to threads after no references to the freed memory can possibly exist. The ABA problem is subtly different from the garbage collection problem; garbage collection ensures that memory is not freed too early, while

ABA prevention ensures that memory is not *re-used* too early.

3.2.3 Complexity

Latch-free algorithms are notoriously complex to specify, let alone implement. Indeed, even experts have designed incorrect algorithms that have required corrections to be incorporated over time. For instance, Valois' lock-free linked list algorithm [160] was shown to contain a race condition [130]. Michael and Scott's lock-free queue algorithm [131] contained two errors, identified and rectified two years later [132].

Modularity

Researchers have argued that latch-free algorithms can simplify the design of operating systems. Operating systems are susceptible to deadlocks if interrupt handler code and non-interrupt handler code acquire the same latch [39]. Since latch-free algorithms can never lead to deadlocks, researchers and practitioners have proposed using latch-free algorithms to simplify interrupt handler code [65, 92].

Fortunately, database system threads and processes never have to deal with interrupts. Latch-free algorithms therefore do not provide the same modularity benefits to databases as they do to OS kernels. Indeed, one could argue that latch-free algorithms *decrease* modularity in non-interruptible systems, such as databases, because they require idiosyncratic memory management code (Section 3.2.2).

3.2.4 Discussion

While the focus of this section has been the limitations and overheads of latch-free algorithms, there certainly exist scenarios where latch-free algorithms may provide better scalability than latch-based algorithms. For instance, certain latch-free algorithms permit multiple threads to make changes to a data structure concurrently [96, 103, 160].

In general, developers and architects should determine how their algorithms interact with the performance characteristics of multi-core hardware (Section 3.2.1). Simply converting a latch-based algorithm to a latch-free algorithm is rarely a recipe for success. Indeed, there exist concurrent algorithms that combine latches and synchronization-free operations to good effect; for instance, Read Copy Update [126] and Masstree [125] both update no meta-data for reads, but use latches for writes.

3.3 Case study: Tree-based indexes

Indexes are well-known to be an important component of database systems. They allow fast access to database records, and typically implement an interface for inserting, deleting, and searching index entries. At any point in time, multiple insert, delete, and search requests may be concurrently executing against an index. Indexes must therefore support employ synchronization mechanisms to correctly order concurrent requests.

Tree-based indexes, such as B$^+$trees, are an important class of index because they provide an *ordered* record access method [67]. Ordered access methods can be used to evaluate range predicates and scans. B$^+$trees provide an associative mapping from index values to sets of records. Like all tree-based data-structures, B$^+$trees are hierarchical. Leaves store a set of record-identifiers corresponding to particular index values, while internal nodes only store meta-data to navigate to index values at the leaves [67]. Every B$^+$tree consists of a single root node, and every other node is accessed via the root. This hierarchy makes it challenging to implement B$^+$trees' interface in a scalable manner — every updater and reader must traverse the tree from the root, and must therefore synchronize their access to the root. The root and, in general, nodes higher up in the tree can therefore turn into scalability bottlenecks. This section discusses the design of concurrency control mechanisms for B$^+$trees that address the scalability challenges above.

3.3.1 Contention for logical locks

The most well known mechanism for correctly synchronizing concurrent B$^+$tree operations is *latch coupling* [152]. Starting from the root, a request acquires a latch on a node, determines a child node to follow, and recursively continues this process for the child node. A search acquires intention-shared (IS) latches on nodes, and releases a node's latch once a child node's latch is acquired. An update (insert or delete) acquires shared-intention-exclusive (SIX) latches on nodes. An updater holds SIX latches on a sequence of consecutive internal nodes until it is certain that the nodes are safe from modification [152]. If an operation attempts to insert a new entry in a full leaf node, the leaf node is split into two nodes. This causes an insertion to occur in the parent node, which in turn may cause the parent to split if it is full, and so forth. A node may similarly be deleted when the number of elements it contains is below a threshold. Update requests convert their SIX latches to exclusive (X) latches for every internal node that needs to be updated due to the insertion or deletion of child nodes.

SIX latch requests are compatible with IS requests, but incompatible with other SIX requests. Thus, an updater allows readers to access a node simaltaneously, but prevents other updaters from accessing the node. This latching strategy is pessimistic; updaters acquire SIX latches in *anticipation* of modifications and thus block other updaters from accessing the node, even though the node may never actually be modified. Because of B$^+$trees' hierarchical organization, the likelihood of a internal node being modified due to a deletion or insertion of a child node decreases *exponentially* from leaf to root. Hence, SIX latches on "higher up" internal nodes, such as the root, are mostly held for short durations, typically only while the updater checks whether child node is full. Nevertheless, even short duration SIX latches on the root can significantly impact the performance of B$^+$trees. Indeed, in their B$^+$tree performance study, Srinivasan and Carey found that SIX latches can significantly deteriorate performance even when the number of updates in a workload is much smaller than the number of searches [152].

Subsequent B$^+$tree algorithms were designed to avoid blocking requests on the root node, even for short durations. In these algorithms, updaters descend to the appropriate leaf using the same latch mode as searches (in S or IS modes). Upon reaching the leaf and detecting the need for node insertion or deletion, updaters perform internal node modifications moving from leaves to higher up nodes. Based on the techniques they use to correctly interleave tree descent and bottom-up modification requests, these algorithms can be classified into two categories. First, those such as ARIES/IM, which use a form of optimistic concurrency control to synchronize reads by descending requests and bottom-up modifications [134]. Second, those such as B^{link}trees, which maintain extra information in internal nodes so that reads can always correctly navigate the tree [118].

3.3.2 Contention on shared memory

In both ARIES/IM and the B^{link}tree, threads descending the tree latch couple their way down to leaves using shared latches. As a consequence, both algorithms permit significantly more concurrency than B$^+$tree algorithms in which descending update requests employ SIX latches [152]. On today's multi-core hardware, however, read latch acquisition is not a negligible cost in the presence of contention. Even though individual operations request the same compatible latch mode, each operation causes some modification of shared internal latch meta-data, such as a counter representing the number of readers [26, 144]. Since every operation traverses the tree via the root, this internal meta-data is updated on every operation, even if no pair of operations actually conflicts. If multiple requests concurrently attempt to traverse the tree, then the time to update the root latch's meta-data is proportional to the number of concurrent requests (Section 3.2.1). Modern B^{+}tree indexing algorithms are designed to avoid this frequent synchronization, while using bottom-up algorithms for tree modifications (in the spirit of B$^{\text{link}}$trees [118] and ARIES/IM [134]).

Cha et al. proposed an optimistic reader-writer synchronization mechanism for B$^{\text{link}}$trees, OLFIT [66]. In OLFIT, requests acquire exclusive latches to update a node and increment a node-specific version number. To read a node, a request waits for the latch to be released, reads the version number, and optimistically reads the node's contents. This read is then validated by checking that the node's version number is unchanged. Reading a node, performed by every request while descending the tree, therefore requires no writes to shared memory locations. Furthermore while node updates acquire an exclusive latch, nodes higher up in the tree, such as the root, are updated exponentially less often than leaves. As a consequence, OLFIT eliminates frequent synchronization on the root. OLFIT's use of timestamps to validate optimistic reads is a powerful design pattern for scalable reader-writer synchronization. Timestamp-based validation forms the basis of several recent systems, including Masstree [125], a main-memory multi-core index, and Silo [159], an optimistic main-memory multi-core database.

In a similar vein, the Bw-tree uses multi-versioning to eliminate the need for reads to update shared meta-data [121]. The Bw-tree maintains a node's state as a linked-list of immutable "deltas", which must be combined to produce the state of the node. Requests update a node by encoding the update in a new immutable delta, and appending the delta to a node's linked-list. Requests read a node's state by combining the linked-list of updates. The linked-list is periodically compacted to bound the overhead of combining deltas. Bw-tree requests can read a node's state without interacting with updates — reads construct a snapshot of a node state by following a linked-list of immutable deltas, while updates perform a latch-free append on the tail of the linked-list. However, the increase in concurrency comes at the expense of increased read overhead due to pointer dereferences and CPU cycles involved in constructing a node's state from a linked-list of deltas.

The OLFIT and Bw-tree algorithms differ due to their use of latch-based and latch-free mechanisms, respectively. This difference *does not* impact the scalability of either algorithm. Instead, both algorithms are scalable because requests avoid updating frequently accessed shared memory, such as the root node, while descending the tree. Both algorithms can therefore avoid the scalability limitations of sequential updates to a single memory location at the hardware-level (Section 3.2.1).

3.4 Experimental evaluation

3.4.1 Microbenchmarks

This section compares the performance of a latch-free synchronization algorithm to three classes of latching algorithms; a busy-waiting spinlock (TATAS spinlocks), backoff latches which put threads to sleep under contention (Pthread mutexes), and scalable latches which avoid spinning on a global memory location (MCS latches). We evaluate each synchronization primitive on a benchmarking framework which precisely controls amount of parallel and serial work each thread must perform. Threads execute in two phases — a serial and parallel phase. Both serial and parallel phases consist of a fixed number of `noop` instructions. The duration of the serial and parallel phases is varied by changing the number of `noop` instructions to execute.

The synchronization algorithms differ in the mechanism they use to execute the serial phase. In the latching algorithms, each thread can only execute the serial phase if it owns the corresponding latch. Threads in the latch-free algorithm use the approach proposed in Herlihy's generic methodology for constructing latch-free objects [102]. Each thread reads the value of a counter prior to executing the serial phase, speculatively executes the serial phase, and then attempts to atomically `cmp-and-swp` the old counter value with its incremented value. The thread successfully executes its serial phase if the `cmp-and-swp` succeeds. If the `cmp-and-swp` fails, the thread reads the value of the counter again and attempts to re-execute the serial phase. The latch-free algorithm is representative of other implementations of linearizable latch-free data-structures [100, 103, 121, 131]. The latch-free algorithm contains an optimization to reduce spurious `cmp-and-swp` instructions executed by a thread. After speculatively executing the serial phase, a thread will first read the value of the counter and check that the value is unchanged before attempting a `cmp-and-swp` instruction [61]. We do not account for latch-free algorithms' memory management overhead (Section 3.2.2).

We run our experiments on a single 80-core machine, consisting of eight 10-core Intel E7-8850 processors and 128GB of memory. The operating system used is Linux, kernel version 3.19.0-61. We perform three sets of experiments, each corresponding to a particular level of contention; low, medium, and high. The parallel phase in each experiment consists of 200,000 cycles, while the low, medium, and high contention serial phases respectively consist of 100, 1,000, and 10,000 cycles. These serial phase lengths respectively correspond to 0.05%, 0.5%, and 5% of the parallel phase.

Low contention

Figure 3.1 shows the results of the low contention experiment. All algorithms scale perfectly when the number of threads is less than or equal to the number of available CPU cores (note that the x-axis uses a log-scale). Figure 3.1b shows the latency distribution of each algorithm under 80 threads (the number of threads is equal to the number of available CPU cores) — there is no significant difference between the algorithms.

When the number of threads exceeds the available CPU cores, we see differences between the algorithms. The MCS latch's throughput completely collapses. The MCS latch constructs a queue of threads waiting to acquire the latch. If any thread in the queue is preempted, even if it does not yet own the latch, then all later threads are delayed until the preempted thread is rescheduled. In contrast, the TATAS latch's throughput does not degrade as significantly because, unlike in the MCS latch, preemptions of threads that do not hold the latch do not affect other threads. The Pthread latch's throughput also degrades with increasing thread count, but outperforms the TATAS latch. The Pthread latch puts threads to sleep in the kernel if they fail to acquire the latch [11]. If a thread is preempted while holding the latch, other threads will fail to acquire the latch and get put to sleep in the kernel. The kernel is eventually left with no choice but to execute the thread that holds the latch because other threads get put to sleep before their scheduling quantum expires. This has the effect of diminishing the impact of preemption on the Pthread latch's throughput.

In contrast to the latch-based algorithms, the latch-free algorithm's throughput

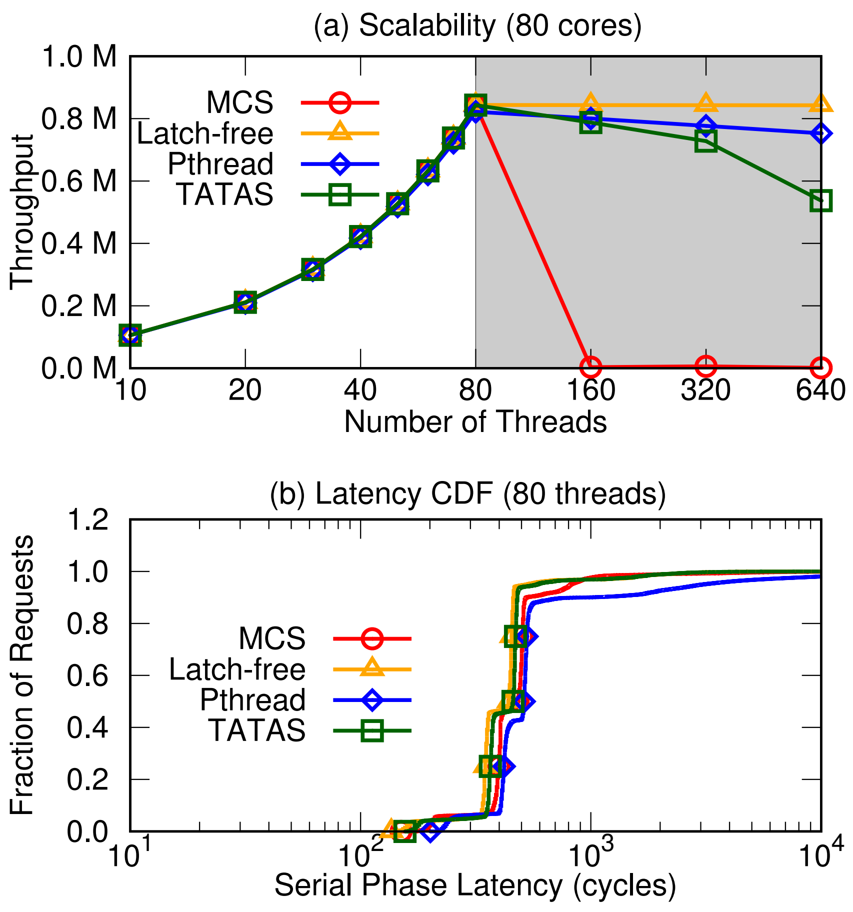


Figure 3.1: Performance of synchronization algorithms under low contention. Serial phase = 100 cycles. Parallel phase = 200,000 cycles. Serial phase = 0.05% Parallel phase.

is unaffected by increasing the number of threads beyond available CPU cores. Throughput does not decrease because thread preemption in the latch-free algorithm never impedes other threads. Throughput does not increase because the server's physical CPU resources are fully utilized at 80 threads. (The server contains 80 CPU cores.) The use of more threads, beyond 80, therefore does not increase throughput.

Medium contention

Figure 3.2 shows the results of the medium contention experiment (serial phase is 0.5% of the parallel phase).

The TATAS latch scales upto 40 cores, but its throughput drops dramatically

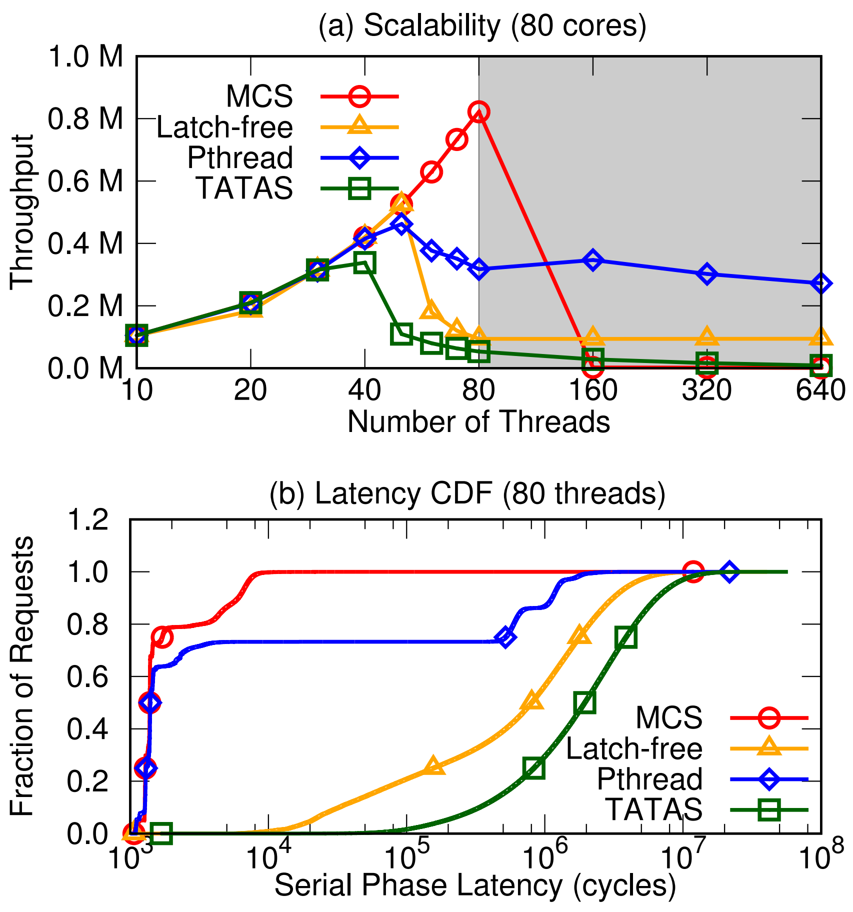


Figure 3.2: Performance of synchronization algorithms under medium contention. Serial phase = 1,000 cycles. Parallel phase = 200,000 cycles. Serial phase = 0.5% Parallel phase.

thereafter; its throughput at 80 cores is less than half its throughput at 10 cores. The reason for the drop in throughput is the transient flood of `xchgq` instructions executed by cores when the latch changes ownership. These `xchgq` instructions impede threads that hold the latch when they attempt to release the latch, effectively increasing the length of the serial phase. Furthermore, the effect of spurious `xchgq` instructions gets worse with increasing core count because more cores contribute to the transient flood of `xchgq` instructions.

The latch-free algorithm scales to 50 cores, but like the TATAS latch, its throughput drops significantly thereafter. The reason for the drop is also similar. Threads read the value of a counter before executing the serial phase, and validate it at the end of the serial phase using a `cmp-and-swp` instruction. If multiple threads attempt to execute the n^{th} iteration of the serial phase, then only one thread will succeed. However, some of the threads will also attempt spurious `cmp-and-swp` instructions because they do not notice the change in the counter value (despite the optimization which checks the value of the counter before attempting the `cmp-and-swp`). These spurious `cmp-and-swp` instructions will delay threads attempting to execute later iterations because atomic instructions on the same memory word are executed sequentially. As in the TATAS latch, this effectively increases the length of the serial phase.

The Pthread latch's throughput does not collapse after peaking. This is because the Pthread latch has a built-in contention handling mechanism. Threads attempt to acquire the latch with two successive `cmp-and-swp` attempts; if they fail, they are put to sleep in the Linux kernel [11]. The Pthread latch's latency distribution shows the effect of backing-off (Figure 3.2b). About 75% of serial phase executions occur without threads backing off, while the other 25% are executed by threads that are put to sleep in the kernel — indicated by the two distinct latency profiles of requests. The variance in latency of serial phases executed without backoff (at the left-hand side of the graph) occurs because of competing `cmp-and-swp` requests.

The MCS latch scales perfectly when the number of threads does not exceed the number of CPU cores. When acquiring the latch, threads perform a single `xchgq` instruction on the latch word and then spin on a local cache line [127], avoiding the overheads associated with transient floods of spurious `cmp-and-swp` instructions and cache invalidations (Section 3.2.1). The MCS latch's latency distribution has a much smaller mean and variance than other synchronization algorithms. The differences in latency across the last 20% of requests is due to queuing delay.

High contention

Figure 3.3 shows the results of the high contention experiment. We set the size of the serial phase to 5% of the parallel phase. The MCS latch's throughput increases until 20 cores and then plateaus. The reason is a lack of parallelism in the workload (1/20th of each transaction is serial).

The TATAS and latch-free algorithms exhibit the same behavior as in the medium contention experiment. Throughput increases until 20 cores and then begins to decrease. The difference between both lines occurs because the TATAS latch's transient behavior when the latch changes ownership *does not* depend on the length of the critical section. This is because cores using the TATAS latch spin in a tight loop. In contrast, cores using the latch-free algorithm speculatively execute the

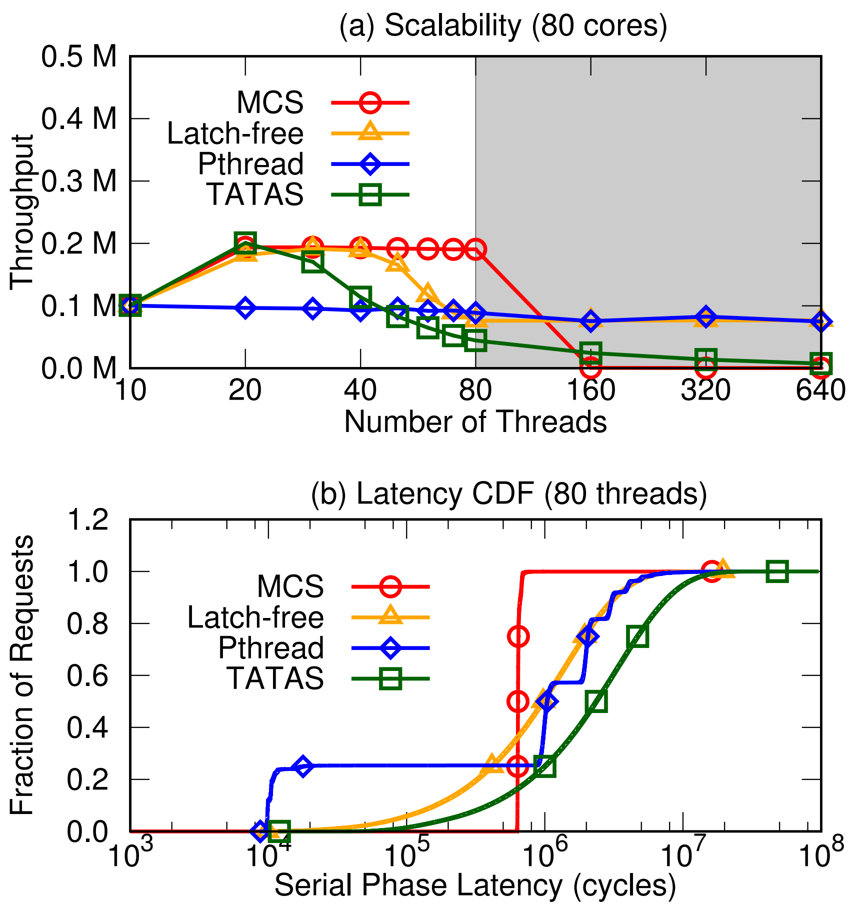


Figure 3.3: Performance of synchronization algorithms under high contention. Serial phase = 10,000 cycles. Parallel phase = 200,000 cycles. Serial phase = 5% Parallel phase.

serial phase between attempts to commit via `cmp-and-swp` instructions.

The Pthread latch does not outperform the latch-free algorithm's throughput in this experiment. As Figure 3.3b indicates, a much larger fraction of threads are put to sleep in the Linux kernel (about 75%).

Figure 3.3b also shows that the MCS latch provides more reliable performance than any other algorithm. This is because MCS latches determine the priority of threads prior to the execution of the serial phase. The latency trend has an important implication for concurrent applications; if the progress of a system depends on stragglers (for instance, algorithms based on barrier synchronization [101]), then non-scalable synchronization algorithms can cause serious degradation in performance.

3.4.2 Queuing experiment

This section shows the effect of avoiding repeated updates against a single shared memory location on a concrete example. We built two versions of a concurrent queue, one latch-based, the other latch-free. Both versions have been used in recently published systems. Jung et al., and Wang and Kimura used the latch-based concurrent queue to construct lists of logical locks in a multi-core optimized lock manager [112,163]. The latch-based queue is also used to determine thread priorities in the MCS latch [127]. Levandoski et al. use the latch-free version of the queue to construct linked-lists of delta updates for Bw-tree nodes (Section 3.3.2) [121].

```
1 xchg_enq(Node **tail, Node *qnode):
2   qnode->prev = INVALID
3   old_tail = xchgq(tail, qnode)
4   qnode->prev = old_tail
```

Listing 3.1: Pseudocode for latch-based enqueue operations using the `xcghq` instruction.

Listing 3.1 shows pseudo-code for the latch-based enqueue algorithm. The algorithm takes two arguments, a reference to the tail of the list (which is itself a pointer to a node), and a reference to the node to be inserted. The new node's `prev` pointer is first marked as `INVALID` (line 2). The algorithm then atomically changes the tail to point to new node using the `xchgq` instruction (line 3). The `xchgq` instruction returns the prior value of the tail, and the new node's `prev` pointer is then changed to reference the old tail value (line 4). The list is temporarily rendered inconsistent between lines 3 and 4 — after atomically changing the tail to reference the new node (line 3), the node's `prev` pointer does not yet point to the valid prior node. To prevent threads concurrently traversing the list from observing this inconsistency, the new node's `prev` pointer is marked `INVALID` on line 2. Traversing threads spin on any node's `INVALID` `prev` pointer until it is changed by the inserting thread on line 4. A new node's `prev` pointer effectively serves as an exclusive latch to prevent traversing threads from observing inconsistent state.

```
1  cmpswp_enq(Node **tail, Node *qnode):
2    while True:
3        qnode->prev = *tail
4        if cmpswp(tail, qnode->prev, qnode):
5          break
```

Listing 3.2: Pseudocode for latch-free enqueue operations using the `cmp-and-swp` instruction.

Listing 3.2 shows pseudo-code for the latch-free enqueue algorithm. The latch-free enqueue algorithm takes a reference to the list tail and a reference to the new node as input. The algorithm first and optimistically sets the new node's `prev` pointer to the value of the tail. The tail's value is obtained by performing a read from memory (line 3). The algorithm then attempts to atomically insert the new node into the list by using an atomic `cmp-and-swp` instruction. The `cmp-and-swp` atomically compares the latest value of the tail with the new node's `prev` pointer and sets the tail's value to reference the new node if the comparison succeeds (line 4). If the comparison fails, the `cmp-and-swp` does not write the tail, and the algorithm retries the steps above.

We compare the performance of these two algorithms using a simple multi-threaded experiment. Each thread repeatedly enqueues new nodes to the tail of a shared list using one of the algorithms above. On successfully performing an enqueue, each thread waits for a specified duration before attempting the next enqueue. We vary contention in the experiment by varying this duration between enqueue requests. We measure the overall throughput of each algorithm (as the number of enqueues performed per second) under low and high contention.

Figure 3.4a shows the result of the high contention experiment. In this experiment, the duration between a successful enqueue and the next enqueue on every thread is set to 10,000 cycles. Both algorithms suffer from the scalability bottleneck of frequently updating the tail — `xchgq` and `cmp-and-swp` instructions are executed sequentially against the tail. However, the latch-based algorithm outperforms the latch-free algorithm by nearly 3x at 80 threads. The difference arises because the latch-free algorithm executes both successful and unsuccessful `cmp-and-swp` operations against the tail. There are two sources of unsuccessful `cmp-and-swp` operations. First, several threads may read the latest value of the tail and subsequently attempt to atomically insert their new nodes into the list, but

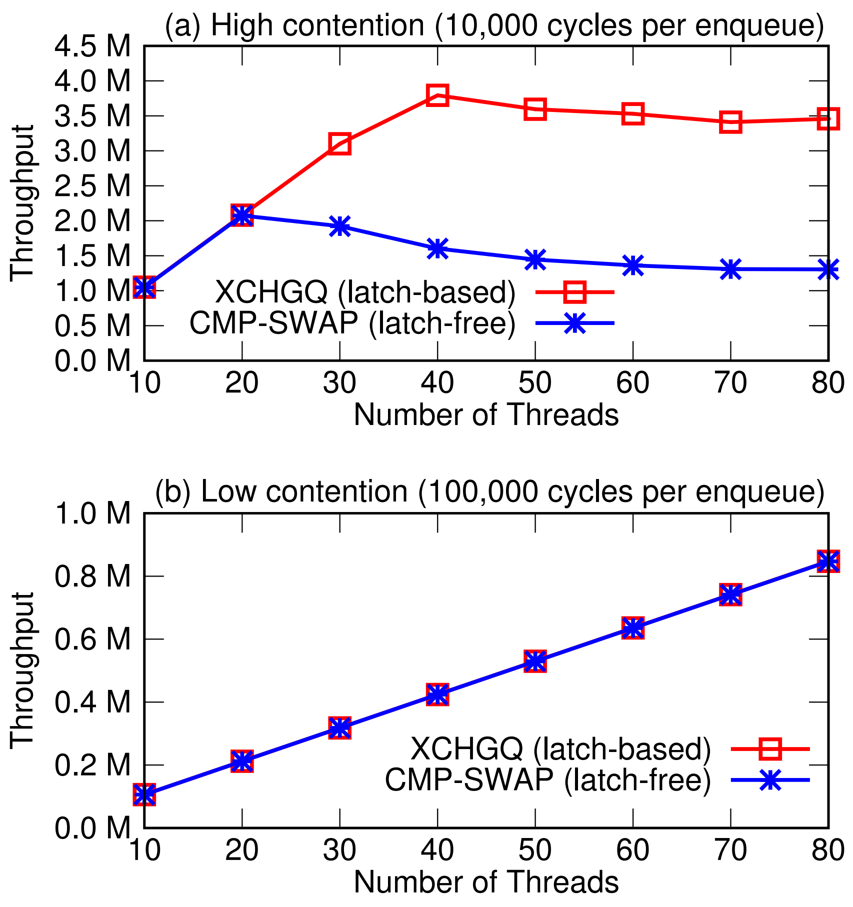


Figure 3.4: Scalability of latch-free and latch-based queues under high and low contention. Throughput is measured as number of successful enqueues per second.

only one will succeed. Second, due to hardware delays in propagating changes in the tail's value (Section 3.2.1), some threads may read a stale tail value and perform a spurious `cmp-and-swp` which will never succeed. The latch-free algorithm in our microbenchmark evaluation experienced similar sources of overhead (Section 3.4.1). In contrast, the latch-based algorithm performs strictly as many `xchgq` instructions as there are enqueues, and therefore experiences significantly less contention for the tail of the linked-list.

Note that in the latch-based `xchgq` algorithm, threads traversing the list must sometimes pay the cost of waiting for a newly inserted node's `prev` pointer to transition from `INVALID` to a valid reference. However, this cost is minimal because a newly inserted node's `INVALID` pointer is changed in the instruction following the `xchgq` (Listing 3.1). Furthermore, since a node's `prev` pointer can only be updated by the corresponding inserting thread, the thread experiences no contention while changing a node's `prev` pointer.

Figure 3.4b shows the result of the low contention experiment. The duration between a successful enqueue and the next enqueue in this case is set to 100,000 cycles. There is no difference in the throughput of the algorithms under low contention.

The experiments in this section show the importance of designing synchronization mechanisms that minimize repeated updates on contended shared memory locations. Repeated updates can cause scalability issues because they are processed sequentially, and therefore increase the amount of sequential execution in a concurrent program. A synchronization algorithm's ability to reduce the number of these sequential operations on shared memory locations is the single biggest factor that influences its scalability.

3.5 Implications

The results of our experimental evaluation indicate that, at the hardware level, the only factor that affects the scalability of a synchronization mechanism is its ability to avoid repeatedly reading or writing a particular location in memory. This can be achieved by designing synchronization algorithms in which threads spin on different memory locations (as in the MCS latch) or use backoff-based contention management mechanisms (as in the Pthread latch). However, we also found that when a system is over-subscribed, pre-emptive scheduling can impact the performance of user-space synchronization mechanisms. Based on these observations, this section outlines implications for the design of multi-core database systems.

Fundamentally, synchronization constrains the schedules of conflicting operations on shared data. The precise nature of a valid schedule depends on the application and algorithm. For example, networking algorithms are resilient to stale information by design, and can hence tolerate reading stale information to make routing decisions [126]. Other applications, such as key-value stores may guarantee linearizability for single key operations [125]. The latter imposes more constraints on schedules of conflicting operations, and this difference is in turn reflected in the synchronization mechanisms used by each algorithm. Synchronization *dynamically* constrains the execution of conflicting operations by forcing their corresponding contexts to coordinate using latching or latch-free mechanisms.

A system could alternatively determine valid schedules *prior* to executing operations. Pre-determining schedules eliminates or reduces synchronization overhead when running operations because a valid schedule has already been determined. At a high level, a pre-determined schedule effectively partitions operations into sets, such that operations in two different sets do not conflict. The operations in different sets can therefore be executed concurrently without the need for any synchronization. The PALM B-tree index is an example of an algorithm that employs such a scheduling mechanism [150]. PALM supports normal B-tree index operations, such as search, delete, and insert. PALM accumulates batches of index operation requests, and performs an analysis on the operations in each batch. During its analysis, PALM divides operations into non-conflicting sets, and then assigns a single thread to execute the operations in a particular set. Threads are thus assigned independent pieces of work to obviate any synchronization during insert, delete, and lookup operations. In contrast, a conventional B-tree implementation requires threads to perform some form of synchronization in order to correctly perform updates and lookups [66,118,119,121,125,152].

The deterministic transaction processing mechanisms described in thesis are designed around the principle of advanced planning. The key insight underlying our research is that pre-determining schedules can avoid the overhead of concurrency control mechanisms based on locking or optimistic validation. These mechanisms totally order transactions prior to their execution, and then relax the total order into a partial order based on actual conflicts between transactions. The partial ordering relationship between transactions is represented using an explicit dependency graph constructed during an analysis phase *prior* to transactions' execution. These systems use an event-driven task-parallel execution model, where ordering constraints between tasks are encoded via the explicit dependency graphs above.

While advanced planning can eliminate or reduce the need for synchronization, its mileage varies depending on the application for two reasons. First, it introduces a tradeoff between scalability and latency — advanced planning improves scalability by reducing synchronization, but creates schedules by *batching* pending operations, which increases latency. This increased latency may hurt overall system performance. For instance, in the B-tree example above, increased latency of index operations may cause logical locks on the corresponding data items to be held for longer or may increase the chances for optimistic validation errors to manifest [144]. Advanced planning therefore typically requires an end-to-end understanding of the impact of increased latency on other unrelated components or higher level abstractions.

Second, in order to construct schedules with opportunities for concurrent execution, advanced planning requires that conflicts between operations can be deduced *prior* to their execution. It should be noted however, that conflicts between operations need not always be *precise*. Advanced planning can only assign concurrent work to tens or hundreds of physical CPUs. The number of non-conflicting sets of operations therefore does not necessarily need to be very large.

The transaction processing mechanisms in this thesis are explicitly designed to work around these two limitations. First, to avoid the deleterious impact of increased latency, transaction schedules are created *prior* to their execution. This extra latency involved in creating schedules does not increase the duration for which conflicting transactions are blocked due to conflicts. On the contrary, it permits significantly more concurrency between conflicting transactions than corresponding state-of-the-art concurrency control protocols by permitting the construction of aggressive serializable transaction schedules [80, 81]. Second, to determine whether transactions conflict, these transaction processing mechanisms speculatively execute a subset of transactions' logic or instead employ coarse-grained conflict information, at the granularity of partitions or even entire tables, which can be obtained via static analysis of transactions [81].

Chapter 4

Lazy transaction evaluation

4.1 Introduction

For decades, transactional database systems have worked as follows: upon receiving a transaction request, the database system performs the reads, writes, and transactional logic associated with the transaction, and then commits (or aborts). Upon receiving a query request, the database system performs the reads associated with the query and returns these results to the user or application that made the request. In both cases, the order is fixed: the database first performs the work associated with the transaction or query, and only afterwards does the database returns the results — read results, and/or the commit/abort decision.

In this chapter, we explore the design of a database system that flips this traditional model on its head. For transactions that return only a commit/abort decision, the database first returns this decision and afterwards performs the work associated with that transaction. The meaning of "commit" and "abort" still maintain the full set of ACID guarantees: if the user is told that the transaction has committed, this means that the effects of the transaction are guaranteed to be durably reflected in the state of the database, and any subsequent reads of data that this transaction wrote will include the updates made by the committed transaction.

The key observation that makes this possible is inspired by the lazy evaluation research performed by the programming language community: the actual state of the database can be allowed to differ from the state of the database that has been promised to a client — it's only if the client makes an explicit request to read the state of the database do promises about state changes have to be kept. In particular, writes to the database state can be deferred, and "lazily" performed upon request

— when a client reads the value of the state that was written.

Therefore, when a client issues a transaction and only expects a commit/abort decision as a result, the work involved in processing the transaction can be replaced by a simple “promise” to the client that it was done. Only when the state that is affected by this transaction needs to be returned to a client does the promise have to be kept, and the work associated with the transaction performed.

However, even if a promise made to a client does not have to be immediately kept, every promise that is made must be theoretically possible to keep. Therefore, in the context of database systems, some amount of work is necessary to return the correct commit/abort decision. There are two classes of scenarios that could cause a transaction to abort: (1) transaction logic can cause a transaction to abort that is dependent on the state of the database and the particular requests made by a transaction (e.g. if the transaction will cause an integrity constraint violation), and (2) the database decides to abort a transaction for nondeterministic reasons that are totally independent of database state (e.g. a deadlock is encountered, or if the database crashes in the middle of processing the transaction).

Given the nondeterministic nature of the second class of scenarios that could cause a transaction to abort, it is impossible in traditional database systems to make promises in advance that a transaction will commit without running the transaction to completion and then actually committing it. Hence, lazy evaluation of transactions has not been a viable option for traditional database architectures. However, deterministic databases, such as Calvin [157], H-Store [155], and VoltDB [161], completely eliminate nondeterministic aborts. Once these types of aborts are eliminated, transactions can only abort due to client-defined transaction logic. Lazy evaluation therefore becomes possible by performing enough work during transaction initiation to determine if, given the current state of the database, transaction logic will cause an abort. If not, a “commit” decision can be immediately promised to the client, with the actual transaction processing performed lazily. Furthermore, the deterministic guarantees of the database can be leveraged to ensure that when the database eventually gets around to processing the transaction, it will be processed over the snapshot of database state that existed when the transaction was originally submitted (instead of the current state) so that the same database state that was used to determine whether or not the transaction will commit will still exist at the time the transaction is processed.

In this chapter, we describe how deterministic database systems can be extended to allow lazy evaluation of transactions, and explore the tradeoffs involved in this approach. In particular we find that laziness provides the following advantages:

- **Improved overall cache/buffer pool locality.** If record X is modified several times before its value is requested by a client query, it needs to be accessed via an IO operation and brought into memory/cache only once, when the writes and final read all occur together (Section 4.2.7).

- **Temporal load balancing.** Deferring execution of transactions requested at peak load times can reduce workload skew between peak and non-peak hours, lowering resource provisioning requirements (Section 4.2.6).

- **Avoiding unnecessary work.** Transactions need not *ever* run to completion fully if their write-sets are never read. This can happen if the write-set of a transaction is overwritten by a blind write (a write that is not dependent on the current value of a data item).

- **Reduced contention footprint.** Contention can be reduced by only executing high-contention operations within a transaction eagerly (the rest of the work is executed at a later time, lazily). This reduces the size of the critical section around contended data access (Sections 4.2.3 and 4.2.4).

- **Reduced transaction execution latency.** If a client is only expecting a commit/abort decision as a result of submitting a transaction to the database system, lazy evaluation allows this decision to be returned before the execution of most transactional logic, thereby significantly reducing the transactional latency that is observable by the client.

On the other hand, lazy transactions introduce certain hazards, and in parting with traditional transaction processing dogma, they introduce new challenges:

- **Higher read latencies.** A request by a client to read a data item may incur delay while writes to that item from other transactions have to be performed prior to the read.

- **Dependencies between deferred transactions.** If many conflicting transactions have been deferred, then executing a particular transaction involves executing a significant number of other transactions that the writer depends on (Section 4.2.2).

- **Overhead of determining the write set of a transaction.** To know what promises have to be kept before reading a data item, any transaction that is processed lazily must mark in some way all items that it will write, so that the database can ensure that these writes will occur before the data item is read. If the user does not explicitly provide the write set of a transaction, additional work is required to determine the write set before promises can be made (Section 4.2.3).

Given this new set of tradeoffs that lazy processing of transactions introduce, it is clear that some workloads are poorly suited for lazy evaluation, while other workloads will see significantly improved throughput, latency, and provisioning characteristics if transactions are executed lazily.

4.2 Lazy Transactions

To illustrate and motivate our approach to implementing lazy transactions, we first examine a naïve implementation of a lazy database system. For the purposes of this example, let us begin by considering only transactions that do not contain logic that can cause them to abort (this restriction will be lifted shortly).

Our naïve lazy system logs transaction requests as it receives them, and replies to each client with a commit "promise" as soon as the request is durably written to the log. However, no additional action is taken immediately to execute the transaction and apply its effects.

When a client goes to read some record(s) in the database at some later time v, this prompts the log to be played forward, so that all transactions before v in the log are executed—then the client can safely read from a snapshot at time v. This playing-forward of the log may involve executing transactions in the log serially, or it may use a locking mechanism that guarantees equivalence to a serial order in the order specified in the log—e.g., deterministic locking [156, 157] or VLL [145]—to parallelize transaction execution and increase resource utilization.

In fact, existing deterministic database systems such as Calvin and VoltDB already implement exactly the machinery required for this type of lazy execution—but they replay the log eagerly rather than waiting for new read requests to prompt them. This is because very little is gained by this implementation of lazy transactions—exactly the same transactions are executed using exactly the same scheduling mechanisms, but with artificial delays inserted.

The basic problem is that the naïve system has to play forward the *entire* log up to v in order to perform *any* read at timestamp v—even if many of the transactions that were executed had no effect whatsoever on the result of the client's read. Below, we describe an approach to implementing lazy transactions using "stickies" that has considerably more useful properties.

4.2.1 Stickies

We introduce our lazy execution technique with an example. Consider a transaction T that writes out a set of records $\{x, y, z\}$ whose values depend on the current values of a (possibly overlapping) set of records $\{a, b, c\}$, implemented as a stored procedure as follows:

```
T ({a, b, c}, {x, y, z}) {
  Read a.
  Read b.
  Read c.
  Perform local computation.
  Write x.
  Write y.
  Write z.
  Commit.
}
```

Note that T has two useful properties: (a) T's write-set is known at the time the transaction is invoked (since it is provided as an argument), and (b) like the transactions handled in the naïve approach described above, T will always commit if executed to completion. For ease of explanation, we will assume that all transactions have these properties for now, and describe how our lazy execution engine handles lazy transactions that do *not* have these properties in Section 4.2.3. We will also assume a data storage structure that supports full multi-versioning; each write inserts a new record without deleting the previous version of the record.

A traditional (eager) database system executing T at time v would read the latest versions of $\{a, b, c\}$, perform whatever computation is specified by the transaction code, and write the resulting new versions $\{x_v, y_v, z_v\}$ to the storage system.

When T is processed by our lazy execution system, however, records $\{a, b, c\}$ are *not* read, nor is any of T's local computation executed—but new records *are* written out to $\{x_v^s, y_v^s, z_v^s\}$ nonetheless. Since no actual values have been computed, these records are *stickies*—temporary place-holders for the real values that provide the reader just enough information to evaluate the specific record when needed. (The superscript 's' denotes here that a record is a sticky.) For example, the sticky written to x_v^s in our example indicates to future readers of the record:

> `This record is a` **`sticky`** `and does not store an actual value.  To compute this record's value, execute transaction` T_v`.`

After stickies have been created for each element of T's write-set and written out to the database's storage engine (and the transaction request T_v is appended to the transaction input log), T is considered to have been durably committed at time v. We refer to this process as *stickification*, and we refer to a transaction such as T_v that has committed by passing the stickification phase—but whose client-specified transaction logic has not yet been fully executed—as a *deferred* transaction.

4.2.2 Substantiating stickies

If a client subsequently looks up the record x at a timestamp $v' > v$ and finds that x_v^s is the most recent version of x, the server retrieves the transaction request T_v, executes it fully (including performing all reads and computation), and overwrites the stickies $\{x_v^s, y_v^s, z_v^s\}$ with the actual values produced, so that subsequent readers need not re-evaluate T_v again. We refer to this process as *substantiation* of x_v.

Note that T_v's read-set $\{a, b, c\}$ (specifically, the latest versions of these records that precede v) may also contain stickies written by earlier transactions. For example, when T_v attempts to read the latest version of a preceding v, it may find a sticky a_u^s inserted by an earlier transaction at time u (let's call that transaction T_u). In order to substantiate x_v^s, a_u^s must first be substantiated by looking up T_u's entry in the transaction request log and fully executing T_u.

In general, there may be many transactions on whose write-sets T_v (transitively) depends—all of which must be executed in order to finally substantiate x_v^s.

Transactional dependencies can be represented graphically, as shown in Figure 4.1. In Figure 4.1(a), $T1$ is ordered before $T2$ in the log. However, because $T1$'s read and write sets are mutually exclusive from $T2$'s, both may be executed independently. Figure 4.1(b) illustrates the distinction between the transaction ordering as imposed by the log and the actual data dependencies among transactions. The log

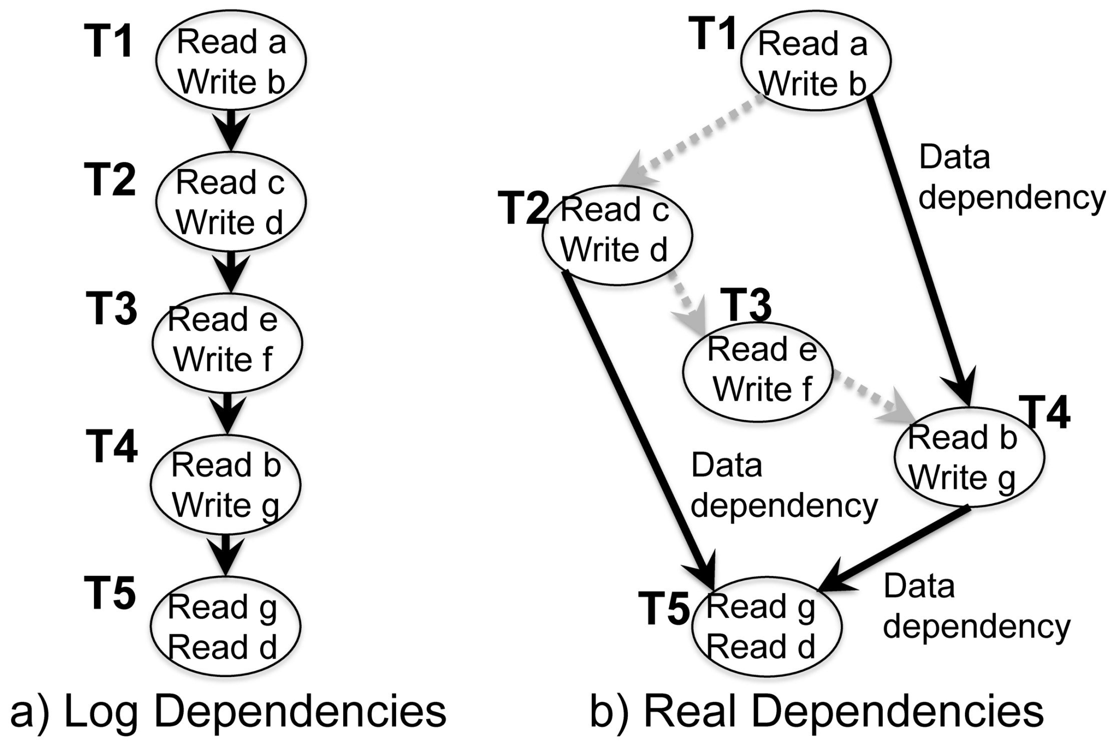


Figure 4.1: Transaction Ordering in the Log

ordering between transactions is shown by dotted lines. The data dependencies between transactions are depicted by solid lines.

Data dependencies between transactions form a *partial order*. The substantiation process needs to obey this partial order, *not* the stricter total order imposed by the stickification log.

A consequence of the partial order among transactions is that substantiation of transactions that *do not* have any data dependencies between them can be completely independent. Referring to Figure 4.1b again, a lazy database system could choose to substantiate the set $\{T1, T2, T4, T5\}$ today, and $T3$ at a later time.

Any partial order can be represented by a directed acyclic graph (DAG). To substantiate a transaction T, we have to first substantiate every transaction T *transitively* depends on. For instance, consider again the data dependencies between transactions as depicted in Fig 4.1(b). $T5$ depends on $T4$ and $T2$, but it transitively depends on $T1$. To substantiate $T5$, we must substantiate $T2$, $T4$ *and* $T1$.

4.2.3 Partially Lazy Transactions

Up until now, we have considered only transactions (a) whose write-sets are known in advance and (b) that cannot abort due to specified transaction logic (e.g., integrity constraint checks). In order to handle the more general class transactions for which these properties need not hold, our execution engine actually executes transactions "partially lazily", dividing each transaction into two phases:

- a **now-phase**, executed immediately, and
- a **later-phase**, which may be deferred until some element of the transaction's write set actually needs to be substantiated.

A transaction's now-phase generally includes:

- All constraint checks and client-specified logic needed to determine commit decisions.
- Any reads from the database state that are necessary to determine a transaction's read/write set. One example of this is a transaction that reads a record via a secondary-index lookup. Without doing the lookup, it is not possible to determine which record must be read. For example, in TPC-C, all $OrderStatus$ transactions must read a customer record, and for some fraction of these transactions the record's primary key must first be determined by a secondary index lookup on the customer's name. If no secondary index is maintained on that field, a full table scan must be performed during the now phase, exactly as it would be with an eager execution mechanism.
- Inserting stickies for each element of the write-set that will be written to during the transaction's later-phase. In addition, secondary index records must be updated to reflect any changes that the transaction will make to indexed fields.

To provide full serializability, transactions' now-phases must be executed in a manner that guarantees equivalence to serial execution in log order. This is one of the places where the use of a deterministic execution, as employed by systems such as Calvin [157], is very helpful — these systems are capable of performing transactions in parallel while still guaranteeing equivalence to a deterministic sequential execution in a specific predetermined order.

Determining a transaction's commit decision in the now-phase is not necessarily straightforward. To examine some common patterns, we examine TPC-C's $NewOrder$ transaction.

$NewOrder$ transactions make up the bulk of the TPC-C benchmark in terms of transaction numbers, work done when running the benchmark, and total contention levels between transactions. Each $NewOrder$ simulates a customer placing an order for between 5 and 15 items from an online retailer, and consists of several steps[1]:

a) **Constraint check.** Abort if a requested item has an invalid ID. The TPC-C specification states that 1% of all $NewOrder$ transactions should fail this constraint check.

b) **One high-contention Read-Modify-Write (RMW) operation.** Increment one of ten Districts' `next_order_id` counters.

c) **8-18 no-contention reads.** Read records from the Warehouse, District, Customer, and Item tables.

d) **5-15 low-contention RMW operations.** Update Stock records for each item purchased.

e) **8-18 blind writes.** Insert records into the Order, OrderLine, NewOrder, and History tables.

Since a TPC-C $NewOrder$ transaction only aborts in the event of an invalid item request, including operation 1 above in the now-phase while leaving the rest of the transaction's logic in the later-phase is sufficient to determine the commit decision. (Performing checks of this kind at the beginning of a transaction's code is a common idiom in transactional applications.)

Uniqueness constraints are also common and can be handled specially. Suppose that $NewOrder$'s blind write into the History table required a uniqueness check on its primary key[2]. Database systems often use Bloom filters for uniqueness checks, and this technique can be applied here. Whenever a record is inserted into a relation, the primary key of this record is checked and inserted into a Bloom

1. For brevity, we examine here a TPC-C deployment consisting of a single warehouse.

2. This is only a hypothetical scenario—History record primary keys are guaranteed to be unique, so this is *not* actually necessary according to the TPC-C specification.

filter. Even unsubstantiated records for which a sticky has been created but not yet substantiated can be included in this Bloom filter since all stickies include the primary key value. In this scenario, the now-phase would first check the Bloom filter for the record in question (let's call it h). If the Bloom filter indicated that no previous version exists, the check passes, and h must be inserted into the Bloom filter (in addition to the sticky h_v^s being inserted into the History table). If the Bloom filter showed that a previous version of h may in fact exist (which might be a false positive)—or if a Bloom filter were not used—the $NewOrder$ transaction then performs a *non-substantiating* read of h. Such a read looks up h in the History table, and if it finds a sticky h_u^s, it does *not* attempt to substantiate it or discern its value by recursively executing the transaction that inserted it, since the precise value is not needed to discern that the uniqueness check has failed.

For uniqueness checks on non-primary-key columns, secondary-indexes on those columns would have to be maintained in the now-phase to avoid full table scans (as mentioned earlier).

A third possibility is that a transaction may violate other types of integrity constraints or have user-specified conditional logic triggering an abort. For example, a system may abort any transaction that would result in a negative stock level. In such a case even more of the transaction logic is forced to execute within the now-phase—limiting the amount of laziness that is achievable for certain classes of transactions.

Specifying now and later-phases

To maximize the benefits of lazy execution, transactions need to be divided into the now and later-phases. There are two options for doing this: user-driven or automatically-driven. In our implementation discussed below, we choose the user-driven approach. Clients specify transaction logic by registering C++ stored procedures. In each procedure, all code is executed in the now-phase up until a special `EndNowPhase()` method is called. When the stored procedure code calls `End-NowPhase()`, stickies are written out to all not-yet-updated elements of the transaction's write set, and control passes back to the calling thread at the call site of the stored procedure. Execution resumes from the same place only when a sticky written by the transaction is substantiated.

Stored procedures whose logic does not contain any call to `EndNowPhase()` execute entirely eagerly and insert no stickies; it is safe for workloads to mix eager and lazy transactions.

Depending on client-provided annotations to determine how much of each transaction to defer to a lazy phase has a clear cost: it imposes an additional burden on database system users, who must reason carefully about data dependencies to safely use lazy transaction evaluation. To ameliorate this, it is possible to introduce automated dependency analysis tools (similar to those commonly used in compilers) that could statically detect the earliest place in the stored procedure logic where it would be safe to call `EndNowPhase()`. For example, in the case of the TPC-C $NewOrder$ stored procedure, it could be statically determined that no operations after the constraint check could lead to an abort decision or modify the write set. By carefully choosing what work to defer, clients can also reduce lock contention as a bottleneck in high-contention workloads, as we discuss in the following section.

4.2.4 Transaction Contention in Lazy Database Systems

A transaction's *contention footprint* is the duration of time that it limits the total concurrency achievable in the system. If transactions are executed serially in a single thread, each one has a contention footprint of its entire active duration, since no other transaction may execute until it completes. In systems that use locking for concurrency control, contention footprint corresponds to the length of time that a transaction holds locks, thereby preventing conflicting transactions from executing. In systems that use optimistic concurrency control, a transaction's contention footprint is the time period from when it starts executing until the critical section of its validation phase—the period during which other transactions may have performed writes that then cause it to fail validation.

The contention footprint of a transaction executing in a lazy database system is more complex, since transactions are not executed all at once. In general, total system throughput can be limited by either now-phase contention or later-phase contention.

Two transactions' now-phases conflict with one another if the two transactions would have conflicted had they been executed eagerly. However, when blocking on another transaction's now-phase, it is only necessary to block until that transaction's now-phase (including stickification) completes. The blocked transaction can begin executing its now-phase before the first transaction's later-phase. Thus, although the likelihood of transactions conflicting is no different, the total contention between now-phases footprint is reduced compared to eager execution.

Later-phase "contention" manifests as dependencies when substantiating stickies: when multiple transactions' later-phases require a single sticky to be substantiated, the amount of parallelism that can be achieved is reduced, since the worker threads processing those later-phases block on that one substantiation operation.This blocking behavior mirrors that which would be observed when executing the same transactions eagerly using a lock-based concurrency control scheme—readers block getting read locks until writers release their exclusive write locks.

Unlike traditional locking protocols, and even MVCC, later-phase contention does not reflect write-write conflicts, but only read-write conflicts, since it is only necessary to substantiate existing stickies when reading a record, not when overwriting it with a new value[3].

Furthermore, later-phase dependencies between transactions only appear for read-write conflicts on records on which the earlier transaction inserted a sticky. If the earlier transaction wrote out a record's actual value (not a sticky) during its now-phase, then later readers of that record need not block on the transaction's later-phase, since the value does not need to be substantiated.

This observation introduces a subtle but powerful opportunity to reduce transactions' contention footprints by pushing high-contention RMW operations into the now-phase and leaving low-contention reads and writes in the later-phase. For example, while TPC-C $NewOrder$'s constraint check is the only step that *must* execute during the now-phase, incrementing the high-contention `District.next_order_id` counter could also be done during the now-phase, while leaving the remaining operations (around 13 contention-free reads, around 10 low-contention RMWs, and around 13 blind writes) in the later-phase. This allows both the now-phase contention and the later-phase contention to be much lower than contention levels observed when executing $NewOrder$ transactions eagerly. Now-phases still conflict with high probability, but are very short, consisting of only the constraint check and incrementing a counter, so blocked transactions need not block for long before running their now-phases. Later-phases do not need to do the RMW operation on any Districts' `next_order_id` counters, and so each transaction only depends on earlier transactions that updated conflicting sets of Stock records, resulting in a much lower contention rate.

3. If a transaction does a RMW operation on a value, it does conflict with earlier writers of that value due to the read part of the RMW operation.

4.2.5 Foundations of Laziness

Our work builds on the theoretical foundations and early implementations of lazy programming languages [98, 104, 105, 162]. In particular, this early work defines a *pure* expression as one whose evaluation neither depends on any external behavior (such as a globally mutable variable being modified by another thread) nor performs any externally visible action (such as printing output or sending messages over a network). A pure expression always evaluates to the same value, regardless of whether eager or lazy evaluation is used. A *function* is considered pure if, when applied to a pure expression, the resulting expression is also pure.

Existing transaction processing systems constrain themselves to eager transaction execution due to an implicit assumption that unpredictable events during transaction evaluation may cause transactions to abort. In other words, transactions are presumed *not* to be pure functions from one database state to the next.

The key observation underlying this work is that database transactions *can* often be formulated as pure functions on database state, introducing the possibility of lazy evaluation. However, this is only possible if deterministic execution is used to ensure non-deterministic aborts do not happen [155–157]. Previous deterministic database systems accomplish this by ordering all transactions into a single serial order before executing them, and writing out this order to a log on stable storage (or across the network). They then use a deadlock-free concurrency control protocol that guarantees equivalence to this serial order that had been defined in advance. A node failure cannot cause a transaction to abort; instead the ordered transaction log is replayed upon a failure (from the most recent checkpoint) to get the database into the same state that it was in at the time of the failure and finishes all in-process transactions from there.

4.2.6 Reducing Peak Provisioning Requirements

Partitioning data across multiple machines is currently the most popular method of servicing high transaction throughput. Repartitioning data on-the-fly to make use of a varying number of machines is challenging, so most practical systems provision a large number of machines, in order to deal with peak traffic. However, these extra machines are not fully utilized most of the time.

A lazy database system can deal with bursty traffic more elegantly. A lazy database can choose to limit the rate of substantiation while dedicating more resources to stickification. When traffic subsides, it can begin substantiating transactions at a higher rate. Since the stickification of a transaction in a lazy database is much less expensive than evaluating a transaction in a conventional system, a lazy database deals with an increased rate of traffic without resorting to adding more machines to the system.

However, this bursty traffic must be mostly transactions that return only commit/abort decisions. If there are many read queries in this burst of traffic, then stickification is not able to get much farther ahead than substantiation, and lazy execution does not help with peak provisioning.

4.2.7 Improving Cache (Buffer Pool) Locality

Consider the case of any two transactions that have a data dependency. We would expect better performance if they were substantiated together, than if their substantiations were separated by a long period of time. Substantiating a transaction involves bringing its records into a processor's cache (and also to the buffer pool for non-main memory systems). If the transactions were substantiated together, then the records the second transaction *shares* with the first will be cache (buffer pool) resident. Thus, evaluating such transactions together yields better cache (and buffer pool) locality. A lazy database system can delay substantiating transactions until the size of a set of *data dependent* transactions is large enough to take advantage of cache locality over large sets of transactions. Since conventional database systems evaluate transactions immediately, they cannot exploit such data sharing between transactions if the difference between the times the transactions enter the system is sufficiently large. In other words, eager databases systems can only expect records to be read from cache if there is temporal *and* spatial locality. Lazy database systems artificially create temporal locality and do not require it to exist naturally in the workload.

4.2.8 Logging and Recovery

As mentioned in Section 4.2.3, to provide full serializability, transactions' now-phases are executed in a manner that *deterministically* guarantees equivalence to serial execution in transaction log order. Given a transaction log, there is only one possible final state of the database system. Therefore, as long as the transaction log is persisted to stable storage, the database system can recover by replaying the log, a concept introduced by other deterministic systems [124, 157]. To avoid replaying the entire history of transactions, checkpointing is necessary. In deterministic database systems, checkpoints must include a complete snapshot of the database as of a particular point in the log where all operations for all transactions before this point are reflected in the database state in the checkpoint, and no operations from future transactions are reflected in the checkpoint — i.e. the checkpoint must be taken at a "point of consistency" with respect to the transaction log[4]. In the case of a lazy database system, a point of consistency cannot be reached unless there are no unsubstantiated transactions present in the system. Although quiescing the database system and finishing the execution of all unsubstantiated transactions before new transactions enter the system is one way to achieve a point of consistency, we use Calvin's mechanism of creating a virtual point of consistency without quiescing the system, and creating a checkpoint over a period of time as transactions that were unsubstantiated at the time of the virtual point of consistency get substantiated [157].

4.3 Implementation

We have taken a clean slate approach to building our lazy database prototype. We have implemented our prototype based on the idea of separating the execution of transactions into two phases; a stickification phase, and a substantiation phase (Section 4.2). Corresponding to the two transaction execution phases, our system architecture is divided into two layers; a stickification layer, and a substantiation layer. The stickification layer is responsible for returning a commit/abort decision to clients and determining the *dependencies* between transactions; once it determines that a transaction can commit, the stickification layer analyzes the transaction's read set, and determines which prior transactions it depends on. The substantiation layer is responsible for executing transactions; it uses the dependency information determined by the stickification layer to batch the execution of a set of dependent transactions. The rest of this section discusses the implementation of the stickification and substantiation layers in detail.

4. Deterministic database systems require consistent checkpoints because they recover state from their input log of transactions. In contrast, conventional non-deterministic database systems can take fuzzy checkpoints [133], which do not need to be transactionally consistent.

4.3.1 Stickification Layer

The stickification layer is the component that receives external input. Processing a transaction involves three steps: first, determining the transaction's commit/abort decision and finding its dependencies; second, maintaining heuristics to process transactions; and third, handing transactions to the substantiation layer.

Dependency Maintenance

When the stickification layer takes a transaction, T, from its input queue, it first executes the transaction's now-phase in order to determine the transaction's commit/abort decision. If the transaction commits, the stickification layer must determine the prior transactions T depends on. It determines T's dependencies based on T's read set; for each record in T's read set, T depends on the last transaction to have the record in its write set. We keep track of the last transaction to have written each record in the system using auxiliary tables mapping primary keys to a 128-bit pair of two 64-bit values: first, a 64-bit transaction identifier corresponding to the record's last writer, and second, a 64-bit counter (Section 4.3.1). For each record in T's read set, we look up the auxiliary table and record a reference to the record's last writer within T. Finally, for each record in T's write set, we update the auxiliary table to identify T as the record's last writer.

In addition to writing the auxiliary table, the stickification thread uses an array local to T to keep track of T's dependencies. As a consequence of tracking dependencies within transactions, the stickification thread maintains an implicit *dependency graph* of transactions. Figure 4.2 shows the workflow of transaction processing in a lazy database system. It shows the stickification layer processing transactions from an input queue, analyzing the transactions, and maintaining the dependency graph. The transactions that make up the graph correspond to as yet *unevaluated* transactions. In order to evaluate a particular transaction T, we have to first recursively evaluate all the transactions T depends on. For instance, if we wish to evaluate transaction T_5 in Fig 4.2, we need to have evaluated T_1, T_2 and T_4.

Heuristics

If the dependency graph is allowed to grow arbitrarily, then the latency of an external read (a read that must be returned to a database user) is adversely affected. External read latency increases with the size of the dependency graph because the

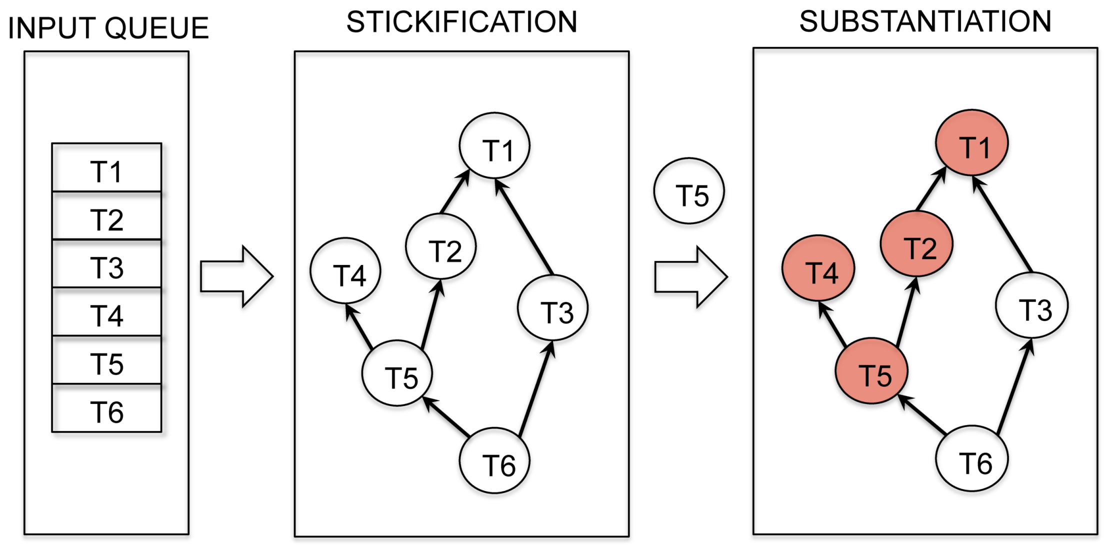


Figure 4.2: Work flow of Lazy Transaction Execution. White circles correspond to Stickified transactions. Red circles correspond to Substantiated transactions.

external read might have a very long chain of transactions it depends on, and this chain of transactions must be executed *before* the external read can be executed. As a consequence, an external read incurs the latency cost of executing every transaction it depends on.

To ensure that the dependency graph does not grow unreasonably large, we keep track of the total number of unexecuted transactions that access each record. We store this information in the second counter field of the value in the auxiliary last-writer table (Section 4.3.1). Whenever a transaction looks up the value keyed by a particular record in the last-writer table, we update the counter field of the value. When we update the counter, we check to see if it exceeds a certain user-defined threshold; if it does, we hand the current transaction to the substantiation layer and reset the counter to 0. Intuitively, the larger the value of the threshold, the longer the chain of transactions that access a particular record.

4.3.2 Substantiation Layer

The substantiation layer takes transactions that are handed to it by the stickification layer and executes them to completion. Our implementation of the substantiation layer consists of multiple worker threads evaluating transactions in parallel. The execution of a transaction on the worker thread proceeds in two steps:

a) **Recursive Dependency Evaluation.** Before executing a transaction's logic, we first need to ensure that its dependencies have been evaluated. The stickification layer ensures that every transaction maintains a *reference* to each of its dependencies (Section 4.3.1). The worker thread looks up each of a transaction's dependencies and checks to see if the dependency has been evaluated; if it has not, then the dependency itself must be recursively evaluated. Figure 4.2 shows the set of transactions that are executed when the substantiation layer is handed transaction T_5. The worker thread must first recursively evaluate T_5's dependencies before it can evaluate T_5.

Two different substantiation layer worker threads working in parallel on different transactions may have an overlapping dependency graph. In order to ensure that transactions within the overlapping subgraph are not processed more than once, each transaction structure is augmented with a single bit which serves as a spinlock. The spinlock protects internal transaction state by ensuring that only one worker thread may substantiate the transaction.

b) **Transaction Logic Evaluation.** Once all of a transaction's dependencies have been evaluated, the thread can proceed with evaluating the transaction's logic; the worker thread reads the records in the transaction's read set, and writes out the records in the transaction's write set.

Executing dependent transactions immediately one after another in the two steps outlined above allows the worker thread to *amortize* the cost of bringing a particular record into on-chip cache across all the transactions that access the record. For instance in Figure 4.2, when the worker thread executes T_5, it can reuse the records brought into cache when it executed T_1, T_2, and T_4 (assuming all their records together fit in cache).

4.4 Experimental Evaluation

Our prototype lazy database consists of a single-threaded stickification layer and a multi-threaded substantiation layer (Section 4.3). As a comparison point, we implemented a system which uses a traditional two-phase locking concurrency control mechanism. Our two-phase locking prototype is built by replacing our lazy database prototype's concurrency control module.

Our experimental evaluation is conducted on a 10 core Intel Xeon E7-8850 processor using 64GB of memory. Our operating system is Linux 3.9.2. We dedicate 8 out of the 10 cores to the transaction processing engine for both 2PL and the lazy system in each of our experiments; both systems use the *same* number of cores. In the 2PL system, each of these 8 cores is utilized by a worker thread. In the lazy system, we dedicate 1 core to the stickification layer, and 7 cores to the substantiation layer. Of the two cores that remain on the system, one is used to drive the database input, while the other is used to measure performance. The measurements for all of our throughput experiments are averaged over 10 runs, the variance across runs in each of our experiments is negligible. Each line in our CDF plots shows a distribution over at least 500,000 points.

This section is organized as follows. Section 4.4.1 describes a set of microbenchmarks designed to evaluate the basic tradeoffs of lazy transaction processing relative to conventional eager processing. Section 4.4.2 evaluates the benefit of deferring transaction execution in the presence of load spikes. In Section 4.4.3 we explore the benefits of eliminating the need for evaluating transactions in the presence of blind writes. Finally, Section 4.4.4 evaluates the benefits of laziness in a high-contention multithreaded environment using the TPC-C benchmark.

4.4.1 Microbenchmarks

We begin our experimental evaluation using a simple microbenchmark involving transactions that perform read-modify-write operations on 10 distinct records. The database consists of a single table with 1,000,000 records. Each record has a size of 1024 bytes, and is indexed by a 64-bit primary key. We pick unique records in each transaction's read-write sets from two different distributions:

a) **Uniform.** We pick 10 unique records among the 1,000,000 uniformly at random.

b) **Normal.** We first select a single record among the 1,000,000 records uniformly at random. The remaining 9 are selected according to a Gaussian distribution around the first record using a standard deviation of 30. The effect of selecting records in this manner is that if two transactions have conflicting read-write sets, then they often conflict on several records. This workload is designed to model applications for which there are correlations in data access; users who buy item X tend to also buy Y in an online shopping scenario, or friends (followers) of X tend to also be friends (followers) of Y in a social networking platform.

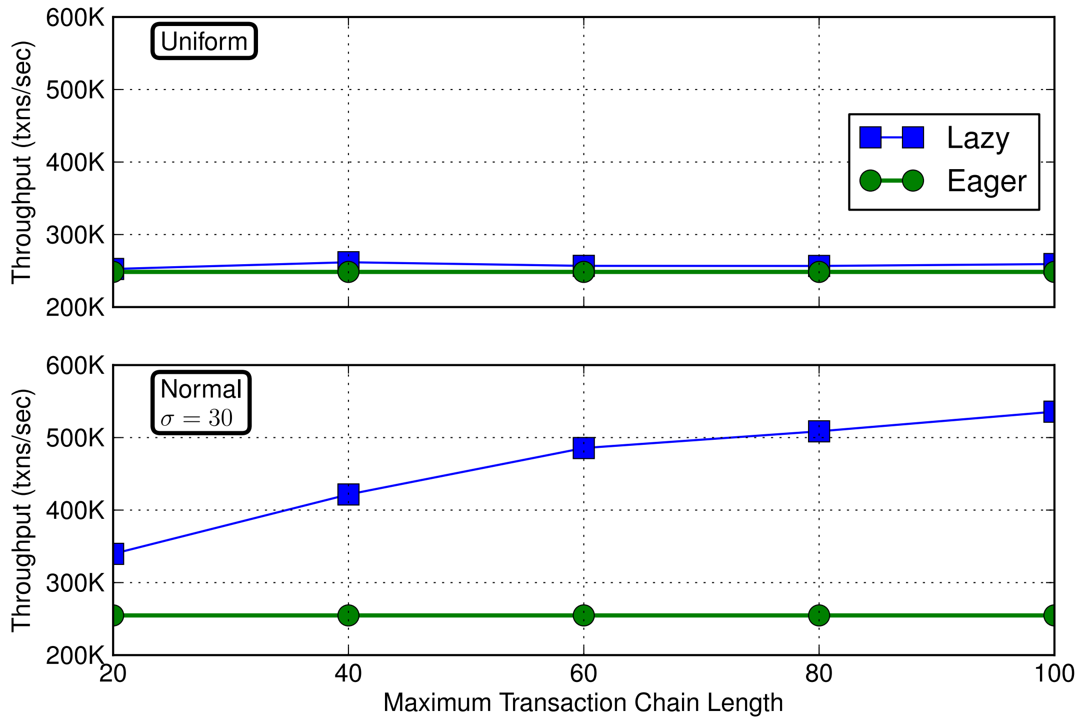


Figure 4.3: Microbenchmark Throughput

Throughput

We first compare the transactional throughput of the lazy and eager systems. Throughput is defined as the rate at which transactions are fully processed. For the lazy scheme, only transactions that have finished both the stickification and substantiation phases (i.e., are completely finished) count towards throughput. The lazy execution scheme is parameterized by the length of the longest chain of unsubstantiated transactions that are allowed to exist before they get automatically substantiated (Section 4.3).

Figure 4.3 shows how the throughput of lazy transaction processing varies with the bound on the longest chain of unexecuted transactions. We plot two graphs, one for the Uniform workload and the second for the Normal workload.

For the Normal workload, we see that the lazy system gets significantly better throughput than the eager system. This is due to the improved cache locality of the lazy system — when substantiating a chain of transactions, the later transactions in the chain find many of the records they access already in cache, having been brought into cache by earlier transactions in the chain. The cost of the initial access of a record (to bring it into cache) is amortized over the number of subsequent ac-

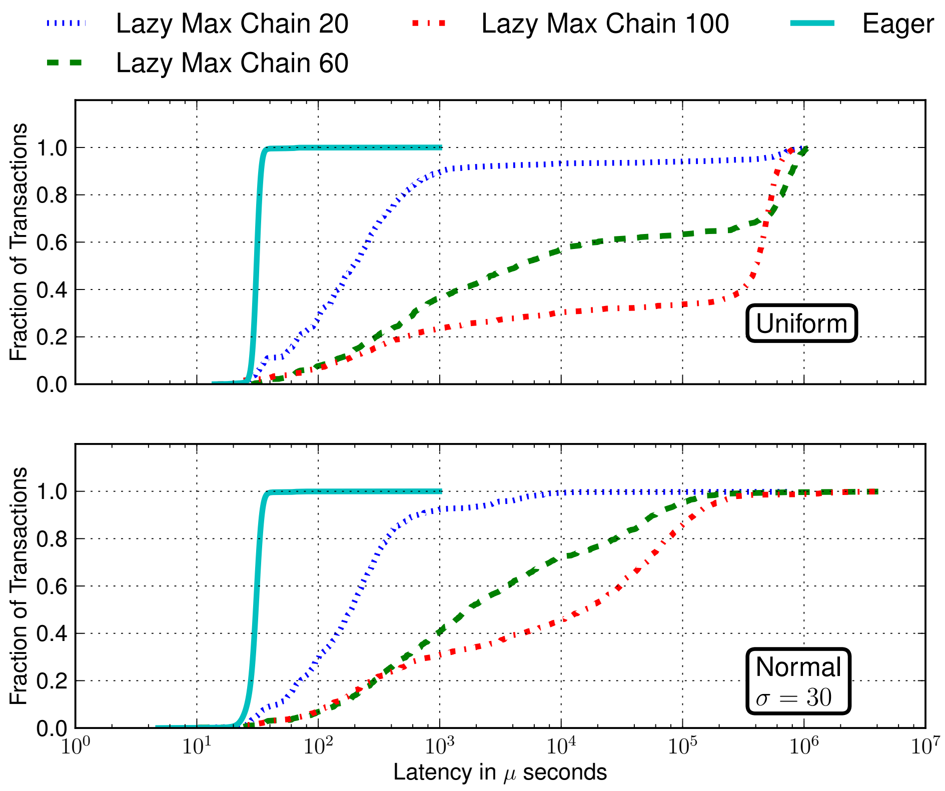


Figure 4.4: External Read Latency CDF

cesses in the chain. Therefore, the throughput difference between the lazy system and eager system increases as the maximum chain size increases.

In contrast, for the Uniform workload, we see that the throughput of the lazy and eager systems is very similar; and furthermore, the chain bound has no effect on the throughput of the lazy system. This is because in the Normal workload, when two transactions overlap in their data access, they tend to overlap on multiple records. Therefore, substantiating a chain of transactions will lead to cache benefits for multiple records. However, in the Uniform workload, two transactions almost never overlap on more than one record. Therefore, the only cache benefits from substantiating a chain of transactions together is just for the one shared record in the chain. This cache benefit is almost completely offset by the additional overhead of maintaining the dependency graph.

End-to-End Latency of Queries

Fig 4.4 shows the latency incurred when the lazy and eager systems receive a query whose evaluation cannot be delayed, because it contains a read request that must be returned to the user. In particular, the query reads a single record from the database. We measure end-to-end latency from the time it begins execution to the time the query result is generated. We generate such "external read" queries at a rate of 1 in every 1000 transactions. In order to execute the logic of an external read query in the lazy system, we have to first execute all unevaluated dependencies of the records that the query reads (Section 4.3.2). Thus, the latencies in this experiment include overhead that is involved in maintaining and traversing the dependency graph of transactions, as well as the latency of executing an entire batch of transactions.

As shown in Fig 4.4, the end-to-end query latency of the lazy system improves with lower limits on transaction chain length, because it forces batches of transactions to be evaluated before they get too large. Meanwhile, the eager system *always* outperforms the lazy system, because when the eager system takes the query off its input queue, it can immediately begin executing it. In contrast, the lazy system first needs to execute the query's unevaluated dependencies before it can begin processing the query. For the Normal workload, which has better cache locality, the cost of evaluating these dependency chains is smaller, which results in a better latency relative to the uniform workload. However, the eager system, which doesn't have any unevaluated dependencies at all, still yields smaller latencies than the lazy system on the Normal workload.

The main conclusion we draw from from this experiment is that when external read queries are rare, the latency to execute such queries can sometimes be much slower in lazy systems than in eager systems, due to the need to process unsubstantiated transactions before beginning query execution (note that the latency graphs used a log scale on the x-axis). In our experimental evaluation of TPC-C, we show the differences between lazy and eager systems when external read queries are more frequent (Section 4.4.4).

4.4.2 Temporal Load Balancing

In this section, we experiment with a lazy database system's ability to deal with bursty traffic. We set up the experiment in the same manner as those in the previous section. We experiment with the Uniform workload so that the lazy and eager databases will have the same baseline throughput, and changes to the baseline as a result of the load burst will be easier to observe. The lazy database's bound on maximum length of a chain of transactions is set to 100. During the course of the experiment, we sample the number of stickified transactions and the number of transactions that have been executed to completion. We sample these statistics every 100 milliseconds, and our plots show how these statistics vary over the course of time.

The experiment lasts for 300 seconds. During the time interval of 0-60 seconds, we warm up the lazy and eager databases with a load of about 100,000 transactions per second. At time t=60s, we simulate a load spike by suddenly increasing the input load to 660,000 transactions per second. We maintain this load during the time interval of 60-120 seconds. At time t=120s, we decrease the load down to 100,000 transactions per second, and maintain it during the time interval of 120-300s. The experiment ends at t=300s.

Figure 4.5 shows the results of the experiment. The topmost graph shows how the input load varies over the course of the experiment. The second graph shows the throughput of the systems as a function of time. For the lazy system, we measure two different types of throughput: (1) throughput observed by the user in terms transactions that have been committed (but in reality have only passed the stickification step) and (2) actual throughput of the system in terms of transactions that complete execution of both the stickification and substantiations phases. We label the former throughput "stickification" in the figure, and the latter "lazy".

During the time interval of 0-60 seconds (when the offered load is 100,000 transactions per second), we see that both databases are able to keep up with the input load. The lazy database's stickification throughput (i.e. the throughput observed by the user) and the eager database's execution throughput both mirror that of the input load. In contrast, the actual throughput of the lazy database is bursty. The reason for this behavior is that the lazy database does not substantiate transactions as soon as they enter the system. Instead, it accumulates batches of transactions until one of two scenarios occurs: either a chain of dependent transactions gets too long and needs to be pruned (so as to adhere to the bound on the maximum length of a chain), or it receives a transaction/query that must be immediately evaluated (and its dependencies must therefore also be immediately evaluated).

During the time interval of 60-120 seconds, we increase the offered load to about 650,000 transactions per second. The throughput of the eager database increases to its maximum, but it is not enough to keep up with the offered load. As a consequence, the eager database begins dropping transactions as its internal queues begin to fill up. This can been seen in the bottommost graph, which shows the percentage of transactions that were able to be successfully processed. We see

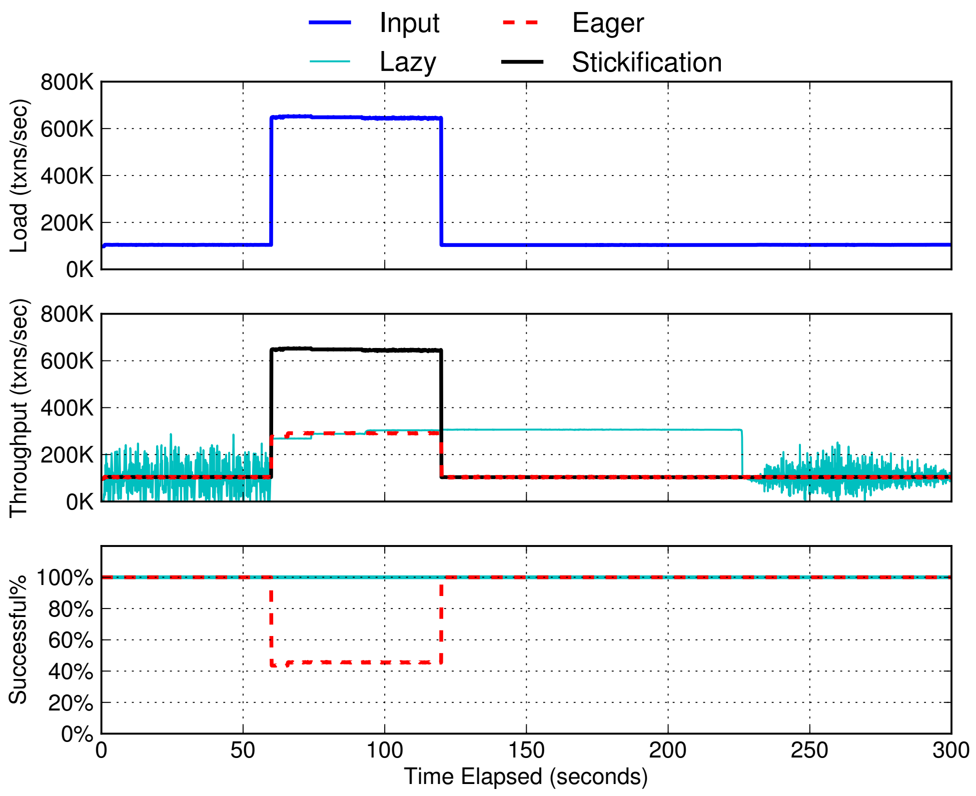


Figure 4.5: Peak Load

that the eager database can only successfully handle about 45% of the offered load. The second graph indicates that the substantiation throughput is *identical* to that of the eager system. However, despite substantiation being unable to keep up with the load, the lazy system does not drop transactions; it is able to handle 100% of the offered load. This is because the *stickification* layer is able to keep up with the offered load.

During the time interval of 120-300 seconds, we lower the offered load back to 100,000 transactions per second. Figure 4.5 shows that the eager system's throughput mirrors that of the offered load; it is now able to keep up with the offered load. In contrast, the throughput of the lazy system remains higher than the offered load. This occurs because the transactions that were processed by the stickification layer but not processed by the substantiation layer result in a backlog. This backlog of transactions to be substantiated is processed concurrently with incoming transactions. The substantiation layer finishes processing the backlog at around t=240 seconds. Once the backlog is processed, the throughput of the lazy system once again becomes bursty.

This experiment demonstrates that in a situation where the substantiation layer is not able to keep up with the load, substantiation can gracefully fall behind stickification and wait until the load burst is complete before catching up (this is why the "actual" throughput of the lazy system remains temporarily high after the load burst is complete). Unlike the eager system, the lazy system is able to *defer* the processing of transactions during resource constrained execution to a time when there are more resources available.

4.4.3 Blind Writes

Until this point, we have only experimented with transactions where all writes to a record are preceded by a read to that same record. However, some workloads contain "blind writes" to data (writes that are not preceded by reads to the same records). Blind writes are particularly beneficial to lazy systems, since by delaying writes to an item, these writes may never have to be performed if they are rendered unnecessary by a blind write.

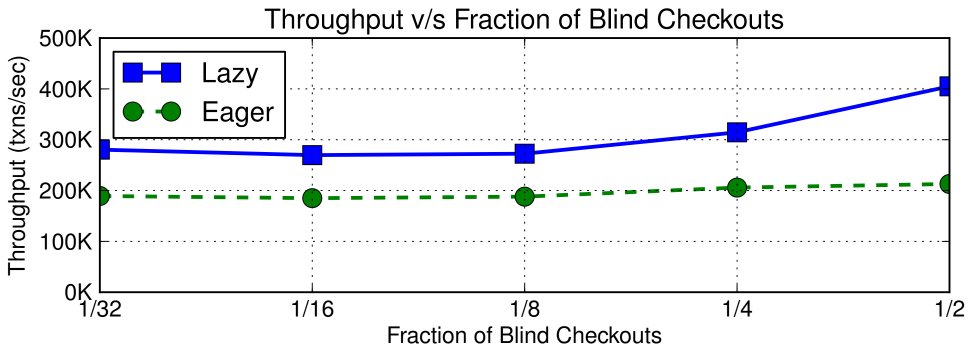
Before getting into a discussion about our experiment, we first describe a scenario where blind writes may occur in practice. Consider the case of a customer using an online shopping portal. The shopping portal's database consists of two base tables – first, an inventory table of items a user can buy online, and second, a shopping cart table, each of whose records corresponds to a particular user's shopping cart. Every customer with an account on the shopping portal has a *private* shopping cart. The state of the shopping cart reflects the items on the shopping portal's catalog that the user is interested in purchasing. During a typical session on the shopping portal, she browses the portal's online catalog for items of interest (the online catalog may be a view of the inventory table). If she finds an interesting item, she adds it to her shopping cart, which causes a read of the online catalog to find the primary key of the item in order to add it to the shopping cart. Eventually a *check-out* operation is performed on her shopping cart. We define a check-out operation as one in which one of the two following possibilities occurs:

- **Type 1.** The user may decide that she wants to buy the items in her shopping cart. In this case, the shopping portal's database back-end must read all the items in the user's shopping cart, and update the inventory table.

- **Type 2.** The user may decide that she does not want to buy any of the items she just added to her shopping cart. Instead she clears the contents of the cart. Alternatively, her session times out, and her shopping cart is cleared automatically. (We still use the term "checkout" for these two cases, even though many Websites would not call a cart clearing operation a "checkout".) The clearing of a shopping cart is a blind-write – the clear operation resets the state of the cart *without* performing any reads. The transactions that added items to the cart are no longer relevant because their effects were "clobbered" by the clear operation.

Our blind-write experiment is motivated by the scenario described above. The workload for our experiment is as follows: each client adds 20 items to its shopping cart and then proceeds to check-out according to one of the two cases described above. Adding an item to the shopping cart involves reading a particular record in the inventory table, and writing the shopping cart record. For simplicity, in the lazy database, we require check-out transactions to be evaluated immediately (even though in many cases a commit/abort decision may be all that is necessary to be returned to the user, and check-outs could therefore be executed more lazily).

The topmost graph in Figure 4.6 shows throughput as we vary the fraction of blind writes. The lazy database's throughput increases as we increase blind-write fraction because transactions whose results are clobbered by a blind-write do not need to be executed at all. However, these transactions still count towards throughput because they are "committed". The eager database's throughput does



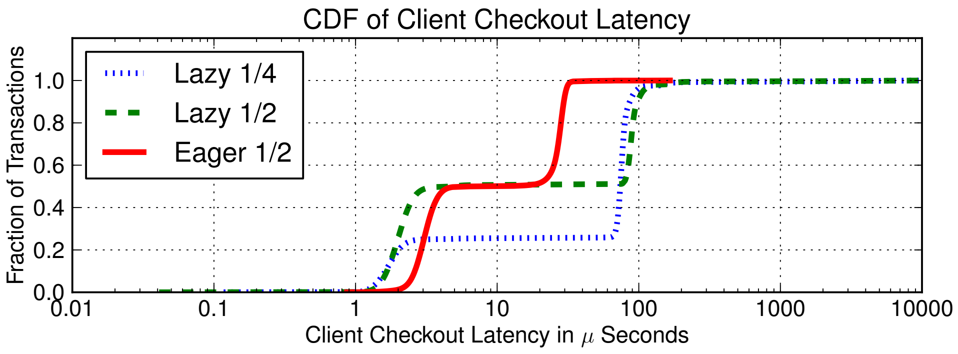


Figure 4.6: Blind Writes

not vary with blind-write fraction, since it is unable to avoid doing work that will eventually be rendered unnecessary.

The plot at the bottom of Figure 4.6 shows a CDF of the end-to-end latency of executing a check-out transaction. In the case of the lazy database, we plot the CDF of check-out latency for two different fractions of blind-writes (1/4 and 1/2), while in the case of the eager database, we plot just for the fraction of 1/2. For the lazy database, the end-to-end latency varies depending on the type of check-out. If the check-out is a blind-write, the lazy database must only evaluate a single transaction (which clears the state of the shopping cart). On the other hand, if the check-out is not a blind-write, it incurs a higher latency penalty because the database must first execute all the transactions the check-out depends on (adding items to the shopping cart). Therefore, the latency distribution is bimodal. As we vary the fraction of blind-writes, the fraction of check-outs with low latency is precisely the same as the fraction of blind-writes.

The latency of a blind-write check-out in the lazy system is comparable to the latency of a check-out in the eager system. This is because both systems perform a similar amount of work — they both evaluate a single transaction. Since the blind-write check-out transaction (clearing the cart) is more lightweight than the non-blind check-out (making a purchase), the distribution of transaction execution latency in the eager system is also bimodal. However, the latency of executing a non-blind checkout (purchase) in the eager system is far less expensive than in the lazy system (the x-axis in the second graph is log-scale) since it has already processed all of the transactions that add items to the shopping cart.

In the scenario mentioned above, the blind-writing "checkout" transaction does not perform any reads. As a consequence, it does not need to wait for the result of any other transaction, and can execute immediately. A lazy database system may not be able to immediately execute transactions in workloads where blind-writing transactions have non-empty read sets. However, even in such a scenario, a lazy database will still never need to process transactions whose effects are clobbered by a blind-write.

4.4.4 TPC-C

Our final set of experiments are designed to evaluate the performance of lazy transaction evaluation on a known benchmark: TPC-C. We found two differences between TPC-C and the other experiments we ran: (1) External read queries appear more frequently (the $StockLevel$, and $OrderStatus$ queries), and (2) TPC-C contains many contended data accesses. In particular, the two transactions that make up the bulk of TPC-C's workload mix, $NewOrder$ (45%) and $Payment$ (43%), write records that are highly contended. Every $NewOrder$ transaction updates a District record, and every $Payment$ transaction updates a District record and its corresponding Warehouse record (each District record contains a foreign key, corresponding to a particular Warehouse's primary key). As a consequence, $NewOrder$ and $Payment$ transactions submitted to the database system will conflict with concurrently executing transactions involving the same District and Warehouse. In a traditional multithreaded database system, such conflicts inhibit scalability; the number of concurrently executing transactions is limited by the number of Warehouses in the system (the number of Districts per Warehouse is limited to 10).

Lazy transaction processing allows for an elegant solution to this scalability problem. Instead of executing writes to highly-contended records in worker threads, highly-contended records can be written solely by the stickification thread(s).

Section 4.2.3 explained that the stickification layer can *partially execute* part of a transaction immediately. We referred to the stickification layer's immediate partial execution of a transaction as a *now-phase*. In Section 4.2.4 we explained that the contention footprint of a transaction can be decreased by moving highly contended accesses to the now-phase, which, for the case of TPC-C, involves moving updates to the district and warehouse records (in the $NewOrder$ and $Payment$ transactions) to the now-phase.

TPC-C Commit Latencies

We now closely analyze the latency distribution of two transactions that are running in the context of the full TPC-C transaction mix: $NewOrder$ and $StockLevel$. We choose these two because they are representative of two fundamental types of transactions: those that only have to immediately return a commit or abort decision ($NewOrder$), and those that need to immediately return the value of database state (an external read) to the user ($StockLevel$).

We plot the client-observed latency of $NewOrder$ and $StockLevel$ transactions while varying the contention in the system. Contention in the TPC-C benchmark is inversely proportional to the number of warehouses in the system. We therefore run each of our systems against two TPC-C warehouse configurations; the first with 1 warehouse (which has high contention), the second with 20 warehouses

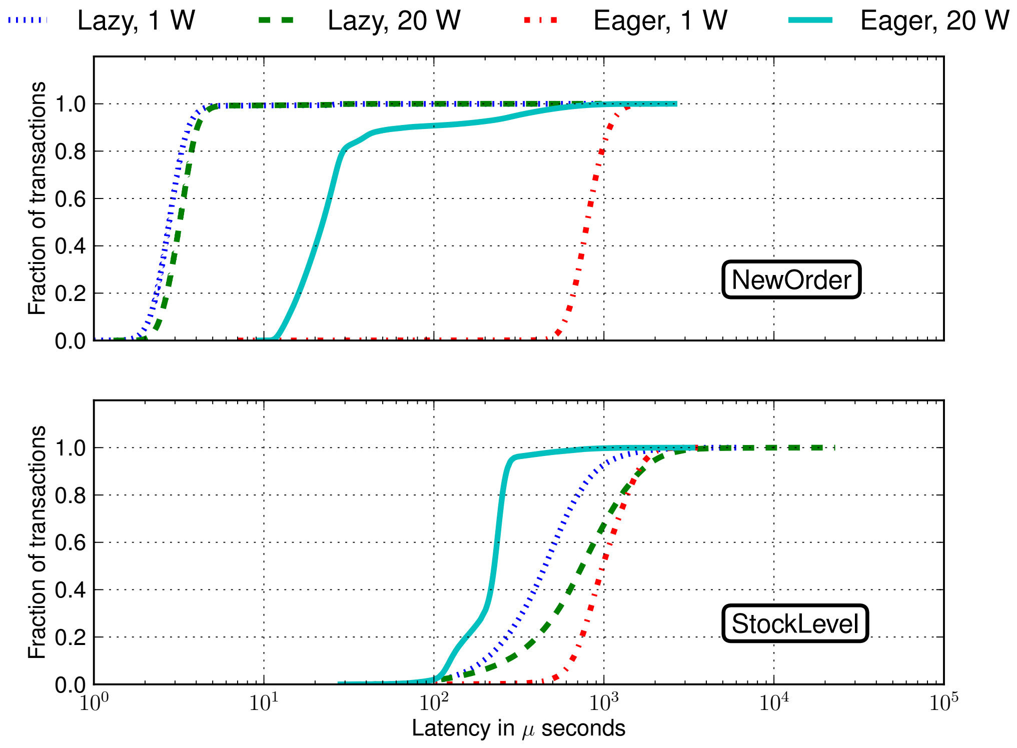


Figure 4.7: Client-observed latency for NewOrder and StockLevel transactions.

(which has low contention).

The top of Figure 4.7 shows the CDF of the client-observed latency of executing *NewOrder* transactions. Lazy transactions clearly have at least an order of magnitude better latencies than eager transactions. This is because the client-observed latency is only the time it takes to receive the commit or abort decision. Since the lazy system does not have to process the entire transaction before returning the decision, it achieves much lower latencies. Meanwhile, the eager system must process the entire transaction before returning the decision to the client.

Furthermore, we see that the latency of eager transactions is significantly impacted by data contention. When there is only 1 warehouse, contention is very high, and most *NewOrder* transactions must wait in lock queues before they can acquire locks and proceed. At 20 warehouses (low contention), these queuing delays are not present.

Since the lazy system also reduces contention by shrinking the contention footprint of the *NewOrder* and *District* transactions, its latencies are not affected by the number of warehouses. Lazy transactions thus have two advantages for transactions that only return commit/abort decisions: (1) they can return to the client without processing the whole transaction and (2) they reduce queuing delays due to contention.

The graph at the bottom of Figure 4.7 shows the CDF of client-observed latency to execute *StockLevel* transactions. Since *StockLevel* is an external read query, the lazy system no longer has the advantage of being able to return early, and therefore no longer outperforms the eager system by an order of magnitude. However, in contrast to the external read latencies for the microbenchmark presented in Section 4.4.1, the latencies for the lazy system are comparable to the eager system. To understand why this is the case, recall from Section 4.4.1 that the reason why lazy systems have high latencies for external reads is that they first need to execute the entire transitive closure of dependencies. In TPC-C, the transitive closure is generally much smaller than in the microbenchmark because TPC-C contains more frequent external-read queries. In particular, the two external-read queries in TPC-C, *StockLevel* and *OrderStatus*, each make up approximately 5% of the workload (recall that the microbenchmarks had one external read per 1000 transactions). These frequent external reads keep the dependency graph from getting large, which reduces the latency of other external reads (keeping them close in cost to reads in the eager system). On the other hand, the smaller dependency graph results in smaller transaction batches and reduced cache locality relative to

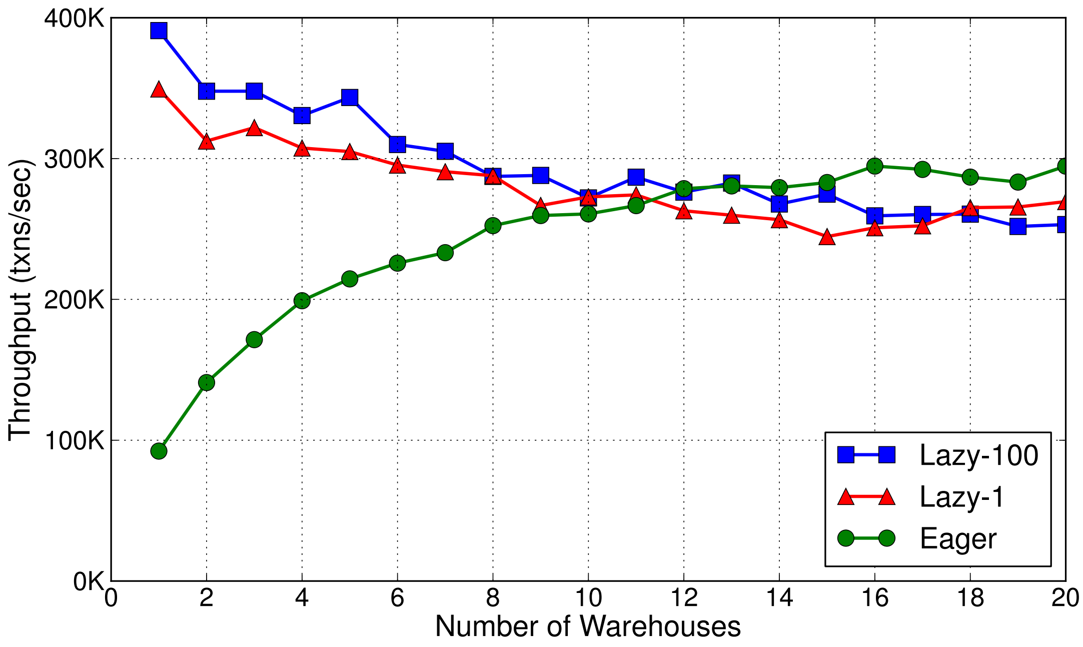


Figure 4.8: TPC-C throughput varying number of warehouses

the microbenchmarks (which we will discuss in more detail for the throughput experiments in the next section).

Unlike the eager system, the latency of the lazy system is *greater* for the 20 warehouse (low contention) case than the 1 warehouse (high contention) case. This is because with more warehouses, the stickification layer experiences larger costs for maintaining the dependency graph, which we explain in detail in the next section.

TPC-C Throughput

In order to analyze the effect of lazy transactions on TPC-C throughput, we run two configurations of the lazy system. The first is the same configuration of the lazy system we have been using in the experiments up until this point, with a bound on the maximum chain length of stickified transactions of 100 (henceforth called "lazy-100"). The second sets the maximum chain length to 1 (henceforth called "lazy-1"). Setting the maximum length chain to 1 forces each transaction's later-phase to be executed as soon as its now-phase completes. This ensures that there is no batching of transactions; when the maximum length is 1, the system is still able to achieve the contention benefits of lazy decomposition of transactions into the now-phase and later-phase, but all cache benefits of laziness are eliminated. By comparing the lazy-100 scheme with the lazy-1 scheme, we are able to distinguish between the cache locality and contention benefits of lazy transactions on TPC-C.

Figure 4.8 shows the results of our experiment. When the system runs with 1 warehouse, we see that both lazy-1 and lazy-100 achieve a substantially higher throughput than the eager system, while the difference between the two lazy systems is more modest. Recall that lazy-100 is able to achieve both the cache benefits and contention benefits of lazy processing, while lazy-1 is only able to achieve the contention benefits. Therefore, the difference between these two lines can be attributed to the cache effects of lazy execution. The rest of the difference of the lazy systems relative to the eager system is due to the contention benefits of lazy execution. We therefore conclude that lazy transaction execution benefits both from improved cache locality and reduced contention, but the benefits of reduced contention are larger for TPC-C. The reason why the cache benefits are not large in this case is explained in the previous section — the large number of external read queries reduces the size of the dependency graph, limiting the amount batching that the lazy system is able to perform.

As the number of warehouses increases, the contention in the system decreases. As a result of the reduced contention, we see that the throughput of the eager system steadily increases until about 12 warehouses, after which contention is no longer the bottleneck, and throughput stabilizes.

In contrast to the eager system's increase in throughput, the throughput of lazy-1 and lazy-100 *decreases* as we add more warehouses. This decrease in throughput occurs because the throughput of both lazy-1 and lazy-100 is limited by the throughput of the stickification layer. The stickification layer performs two tasks for every transaction that enters the system; first, it processes the transaction's now-phase, and second, it maintains a dependency graph of transactions to be processed by the substantiation layer. While the now-phase is short for TPC-C (only reads or writes to records in the Warehouse and District tables are performed), the overhead of dependency graph maintenance is much higher. Maintaining the dependency graph of transactions involves tracking the last transaction to write a particular record by maintaining an *inverted index* from each record to its last writer (Section 4.3.1). Since the inverted index tracks the last writer of *every* record in the database, the size of the inverted index increases as we increase the number of records in the database. As the size of the inverted index increases, a smaller percentage of it remains in cache, and the stickification layer must may a higher cost to update it.

The fact that the stickification layer becomes the lazy system's primary bottleneck as the size of the database scales is specific to our current implementation, and not fundamental to lazy execution. As mentioned above, our implementation of the stickification layer consists of just a single thread executing every transaction's now-phase and maintaining the dependency graph. Multithreading the stickification phase is an important avenue for future work.

4.5 Conclusions

While lazy evaluation has been applied in the past in programming languages, it is interesting and perhaps surprising to note that laziness in the context of database systems has a largely different set of advantages and applications than in programming languages. In particular our experimental implementation shows that lazy evaluation of transactions in database systems can improve cache locality, temporally load balance a workload during spikes of transactional activity, simplify concurrency control, and reduce latency for transactions that return only a commit or abort decision. Not all workloads are well-suited for lazy evaluation, as some queries are delayed as long chains of dependencies are evaluated, but our experimental results show that there is an interesting class of workloads for which lazy evaluation is able to improve throughput and latency.

Chapter 5

BOHM: High performance serializable multi-version concurrency control

5.1 Introduction

Database systems must choose between two alternatives for handling record updates: (1) overwrite the old data with the new data ("update-in-place systems") or (2) write a new copy of the record with the new data, and delete or reduce the visibility of the old record ("multi-versioned systems"). The primary advantage of multi-versioned systems is that transactions that write to a particular record can proceed in parallel with transactions that read the same record; read transactions do not block write transactions since they can read older versions until the write transaction has committed. On the other hand, multi-versioned systems must consume additional space to store the extra versions, and incur additional complexity to maintain them. As space becomes increasingly cheap in modern hardware configurations, the balance is shifting, and the majority of recently architected database systems are choosing the multi-versioned approach.

While concurrency control techniques that guarantee serializability in database systems that use locking to preclude write-write and read-write conflicts are well understood, it is much harder to guarantee serializability in multi-versioned systems that enable reads and writes of the same record to occur concurrently. One popular option that achieves a level of isolation very close to full serializability is "snapshot isolation" [55]. Snapshot isolation guarantees that each transaction, T, reads the database state resulting from all transactions that committed before T began, while also guaranteeing that T is isolated from updates produced by transactions that run concurrently with T. Snapshot isolation comes very close to guaranteeing full serializability, and indeed, highly successful commercial database systems (such as older versions of Oracle) implement snapshot isolation when the user requests the "serializable" isolation level [106]. However, snapshot isolation is vulnerable to serializability violations [55, 85]. For instance, the famous write-skew anomaly can occur when two transactions have an overlapping read set and disjoint write set, where the write set (of each transaction) includes elements from the shared read set [55]. Processing such transactions using snapshot isolation can result in a final state that cannot be produced if the transactions are processed serially.

There has been a significant amount of work on making multi-versioned systems serializable, either by avoiding the write-skew anomaly in snapshot isolation systems [83, 84], or by using alternative concurrency control protocols to snapshot isolation [64, 117]. However, these solutions either severely restrict concurrency in the presence of read-write conflicts (to the extent that they offer almost no additional *logical* concurrency as compared to single-versioned systems) or they require more coordination and book keeping, which results in poorer performance in main memory multi-core settings (Section 5.2).

In this chapter, proposes BOHM, a new concurrency control protocol for multi-versioned database systems. The key insight behind BOHM is that the complexity of determining a valid serialization order of transactions can be eliminated by separating concurrency control and version management from transaction execution. Accordingly, BOHM determines the serialization order of transactions and creates versions corresponding to transactions' writes *prior* to their execution (Section 5.3). As a consequence of this design, BOHM guarantees full serializability while ensuring that reads *never* block writes. Furthermore, BOHM does not require the additional coordination and book keeping introduced by other methods for achieving serializability in multi-versioned systems. The final result is a highly scalable concurrency control protocol across multiple cores — there is no centralized lock manager, almost all data structures are thread-local, no coordination needs to occur across threads except at the end of a large batch of transactions, and the need for latching or any kind of atomic instructions is therefore minimized (Section 5.3.2).

The main disadvantage of our approach is that entire transactions must be submitted to the database system before the system can begin to process them. Traditional cursor-oriented database access, where transactions are submitted to the database in pieces, is therefore not supported. Furthermore, the write set of a transaction must be deducible before the transaction begins — either through explicit write set declaration by the program that submits the transaction, or through analysis of the transaction by the database system, or through optimistic techniques that submit a transaction for a trial run to guess its write set, and abort the transaction if the trial run resulted in an incorrect prediction [146,158].

Although these disadvantages (especially the first one) change the model by which a user submits transactions to a database system, an increasingly large number of performance sensitive applications already utilize stored procedures to submit transactions to database systems to avoid paying round-trip communication costs to the database server. These applications can leverage our multi-versioned concurrency control technique without any modifications.

BOHM thus presents a new, interesting alternative in the space of multi-version concurrency control options — an extremely scalable technique, at the cost of requiring entire transactions with deducible write sets in advance. Experiments show that BOHM achieves linear scalability up to (at least) 20 million record accesses per second with transactions being processed over dozens of cores.

In addition to contributions around multi-versioned serializability and multi-core scalability, a third important contribution of BOHM is a clean, modular design. Whereas traditional database systems use a monolithic approach, with the currency control and transaction processing components of the systems heavily cross-dependent and intertwined, BOHM completely separates these system components, with entirely separate threads performing concurrency control and transaction processing. This modular design is made possible by BOHM's philosophy of planning transaction execution in advance, so that when control is handed over to the execution threads, they can proceed without any concern for other concurrently executing transactions. This architecture greatly improves database engine code maintainability and reduces database administrator complexity.

5.2 Motivation

We now discuss two fundamental issues that limit the performance of current state-of-the-art multi-version concurrency control protocols: *a*) the use of global counters to obtain timestamps, and *b*) the cost of guaranteeing serializable execution.

5.2.1 Centralized Timestamps

When a multi-version database system updates the value of a record, the update creates a new version of the record. Each record may have several versions simultaneously associated with it. Multi-version databases therefore require a way to decide which of a record's versions – if any – are visible to a particular transaction. To determine the record visible to a transaction, the database associates *timestamps* with every transaction, and every version of a record.

Multi-version databases typically use a global counter to obtain unique timestamps. When a transaction needs a timestamp, the database atomically increments the value of the counter using a latch or an atomic *fetch-and-increment* instruction. Using a global counter to obtain timestamps works well when it is shared among a small number of physical CPU cores but does not scale to high core counts [159].

Note that the use of a global counter to assign transactions their timestamps is a pervasive design pattern in multi-version databases. It is not restricted to systems that implement serializable isolation; implementations of weaker isolation levels such as snapshot isolation and read committed also use global counters [64, 117]. These systems are thus subject to the scalability restrictions of using a global counter.

To address this bottleneck, BOHM assigns a total order to transactions prior to their execution. This a priori assignment of transaction timestamps is inspired by timestamp ordering protocols [57, 60, 143], which were originally proposed to minimize coordination in distributed transaction processing. Each transaction is implicitly assigned a timestamp based on its position in the total order. When a transaction is eventually executed, BOHM ensures that the state of the database is identical to a serial execution of the transactions as specified by the total order. Assigning a transaction its timestamp based on its position in the total order allows BOHM to use low-overhead mechanisms for timestamp assignment. For instance, in our implementation, a single thread scans the total order of transactions sequentially and assigns transactions their timestamps (Section 5.3.2).

BOHM's a priori timestamp allocation mechanism bears resemblance to timestamp ordering, a concurrency control technique in which the serialization order of transactions is determined by assigning transactions their timestamps a priori [57, 60, 143]. However, BOHM concurrency control mechanism is still fundamentally different from timestamp ordering in that

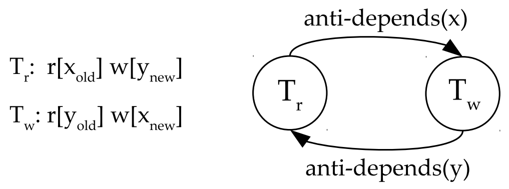

anti-depends(x)
T_r: $r[x_{old}]$ $w[y_{new}]$
T_w: $r[y_{old}]$ $w[x_{new}]$
T_r
T_w
anti-depends(y)

Figure 5.1: Non-serializable interleaving, and corresponding serialization graph of T_r and T_w. $r[x_1]$ denotes to a *read* of version 1 of record x, correspondingly, $w[x_1]$ denotes a *write* to record x, which produces version 1. A record's subscript corresponds to the version read or written by the transaction.

5.2.2 Guaranteeing Serializability

Multi-version database systems can execute transactions with greater concurrency than their single version counterparts. A transaction, T_r, that reads record x need not block a concurrent transaction, T_w, that writes record x. In order to avoid blocking T_w, T_r can read a version of x, x_{old}, that exists *prior* to the version produced by T_w's write, x_{new}. More generally, multiversioning allows transactions with conflicting read and write sets to execute without blocking each other. Unfortunately, if conflicting transactions are processed without restraint, the resulting execution may not be serializable. In our example, if T_r is allowed to read x_{old}, then it must be ordered before T_w in the serialization order.

In the formalism of Adya et al. [42], the serialization graph corresponding to the above execution contains an *anti-dependency edge* from T_r to T_w. In order for an execution of transactions to be serializable, the serialization graph corresponding to the trace of the execution cannot contain cycles. If T_r were to write another record y and T_w read y, then the order of T_r and T_w's operations on y must be the same as the order of their operations on x. In particular, T_w must not read y_{old} (the version of y prior to T_r's write), otherwise the serialization graph would contain an anti-dependency edge from T_w to T_r, leading to a cycle in the resulting serialization graph.

Figure 5.1 shows the interleaved execution of transactions T_r and T_w, and the corresponding serialization graph. The graph contains two anti-dependency edges, one from T_r to T_w, and the other from T_w to T_r; these two edges form a cycle, implying that the interleaving of T_w and T_r as described above is not serializable. This example is a variant of Snapshot Isolation's well known *write-skew* anomaly [55].

In order to avoid non-serializable executions such as the one described above, multi-versioned database systems need to account for anti-dependencies among transactions whose read and write sets conflict. There exist two ways of accounting for anti-dependencies:

- **Track Reads.** Whenever a transaction reads a record, the system associates some meta data with the record that was read. The read meta data is then used to decide on the order of transactions. For instance, the pessimistic version of Hekaton's multi-version concurrency control algorithm associates a counter with every record in the database [117]. The counter reflects the number of in-flight transactions that have read the record. As another example, Cahill et al. modify BerkeleyDB's lock manager to track anti-dependency edges to and from a particular transaction [64].

- **Validate Reads.** A transaction locally keeps track of the version of each record it observed. When the transaction is ready to commit, it validates that the reads it observed are consistent with a serial order. This technique is used by Hekaton's optimistic concurrency control protocol [117], and Multi-version General Validation [43].

While both approaches ensure that all executions are serializable, they come at a cost. Concurrency control protocols track reads in order to *constrain* the execution of concurrent readers and writers. For instance, Hekaton's pessimistic concurrency control protocol does not allow a writer to commit until all concurrent readers have either committed or aborted [117]. In addition to the reduction in concurrency resulting from the concurrency control protocol itself, tracking reads entails writes to shared memory. If a record is popular, then many different threads may attempt to update the same memory words concurrently, leading to contention for access to internal data structures, and subsequent cache coherence slow-downs. Since *reads* are tracked, this contention is present even if the workload is read-only.

The "Validate Reads" approach does not suffer from the problem of requiring reads to write internal data to shared memory. However, validation protocols reduce concurrency among readers and writers by aborting readers. Such a situation runs counter to the original intention of multi-version concurrency control, because allowing multiple versions of a record is supposed to allow for greater concurrency among readers and writers.

To address these limitations, we designed BOHM's concurrency control protocol with the following goals in mind: (1) A transaction, T_r, that reads a particular record should *never* block or abort a concurrent transaction that writes the same record, whether or not T_r is read-only. (2) Reading the value of a record should not require any writes to shared memory.

5.3 Design

BOHM's design philosophy is to eliminate or reduce coordination among database threads due to synchronization based on writes to shared memory. BOHM ensures that threads either make decisions based on local state, or amortize the cost of coordination across several transactions. BOHM achieves this goal by separating concurrency control logic from transaction execution logic. This separation is reflected in BOHM's architecture: a transaction is processed by two different sets of threads in two phases: (1) a concurrency control phase which determines the proper serialization order and creates a data structure that enables the second phase to process transactions without concern for concurrently executing transactions, and (2) an execution phase, which actually executes the transaction's logic.

While the separation of concurrency control logic and transaction execution logic allows BOHM to improve concurrency and avoid scalability bottlenecks, it comes at the cost of extra requirements. In order to plan execution correctly, the concurrency control phase needs advance knowledge of each transaction's write set. This requirement is not unique to BOHM — several prior systems exploit a priori information about transactions' read and write sets [46, 82, 135, 158]. These previous systems have shown that even though they need transactions' write sets, and frequently read sets, in advance, it is not necessary for transactions to pre-declare them. For example, Calvin proposes a speculative technique which predicts each transaction's read and write sets on the fly [158]. Furthermore, Ren et al. show that aborts due to speculative prediction are rare, since the volatility of data used to derive the read and write sets is usually low[1] [146]. BOHM uses this technique if transactions' write sets are not available in advance. Regardless, there is a requirement that the entire transaction be submitted to the system at once. Thus, cursor oriented transaction models that a submit a transaction to the system in pieces cannot be supported.

5.3.1 System Overview

Transactions that enter the system are handed over to a single thread which creates a log in shared-memory containing a list of all transactions that have been input to the system. The position of a transaction in this log is its timestamp. The log is read (in parallel) by m concurrency control threads. These threads own a log-

1. For example, on TPC-C does not abort transactions due to speculative prediction.

ical partition of the records in the database. For each transaction in the log, each concurrency control thread analyzes the write set of the transaction to see if it will write any records in the partition owned by that thread. If so, the thread will create space (a "placeholder") for the new version in the database (the contents remain uninitialized) and link it to the placeholder associated with the previous version of record (which was written by the same thread).

A separate set of n threads (the "transaction execution threads") reads the same log of input transactions, performs the reads associated with the transactions in the log, and fills in the pre-existing allocated space for any writes that they perform. These transaction execution threads do not start working on a batch of transactions until the concurrency control threads have completed that same batch. Therefore, it is guaranteed that placeholders already exist for any writes that these threads perform. Furthermore, reads can determine which version of a record is the correct version to read by navigating placeholders' backward references. If the placeholder associated with the correct version to read is uninitialized, then the read must block until the write is performed. Hence, in BOHM, reads never block writes, but writes can block reads.

The following two subsections give more details on the concurrency control phase and the transaction execution phase, respectively. Furthermore, they explain how our design upholds our philosophy of not allowing contented writes to shared memory, and any thread synchronization at the record or transaction granularity.

5.3.2 Concurrency Control

The concurrency control layer is responsible for (1) determining the serialization order of transactions, and (2) creating a safe environment in which the execution phase can run transactions without concern for other transactions running concurrently.

Timestamp Assignment

The first step of the concurrency control layer is to insert each transaction into a log in main-memory. This is done by a single thread dedicated solely to this task. Because the concurrency control layer is separated from (and run prior to) transaction execution, BOHM can use this log to implicitly assign timestamps to transactions (the timestamp of a transaction is its position in the log). Since a single thread creates the log prior to all other steps in transaction processing, log creation (and thus timestamp assignment) is an uncontended operation. This distinguishes BOHM from other multi-versioned schemes that assign timestamps (which involve updating a shared counter) as part of transaction processing. Thus, timestamp assignment is an example of our design philosophy of avoiding writing and reading from shared data-structures as much as possible.

Several prior multi-version concurrency control mechanisms assign each transaction, T, two timestamps, t_{begin} and t_{end} [43, 55, 64, 117]. t_{begin} determines which versions of pre-existing records are visible to T, while t_{end} determines the logical time at which T's writes become visible to other transactions, and is used to validate whether T can commit. The time between t_{begin} and t_{end} determines the logical interval of time during which T executes. If another transaction's logical interval overlaps with that of T, then the database system needs to ensure that the transactions do not conflict with each other (what exactly constitutes a conflict depends on the isolation level desired).

In contrast, BOHM assigns each transaction a *single timestamp*, ts (determined by the transaction's position in the log). Intuitively, ts "squashes" t_{begin} and t_{end} together; ts determines both the logical time at which T performs its reads, and the logical time at which T's writes are visible to other transactions. As a consequence, each transaction appears to execute atomically at time ts.

Inserting Placeholders

Once a transaction's timestamp has been determined, the concurrency control layer inserts a new version for every record in the transaction's write set. This includes creating new versions for index key-values updated by the transaction. The version inserted by the concurrency control layer contains a placeholder for the value of the version, but the value is uninitialized. The actual value of the version is only produced once the corresponding transaction's logic is executed by the execution layer (Section 5.3.3).

Several threads contribute to the processing of a single transaction's write set. BOHM partitions the responsibility for each record of a table across the set of concurrency control threads. When the concurrency control layer receives a transaction, *every* concurrency control thread examines T's write set in order to determine whether any records belong to the partition for which it is responsible.

Figure 5.2 illustrates how several threads cooperatively process each transac-

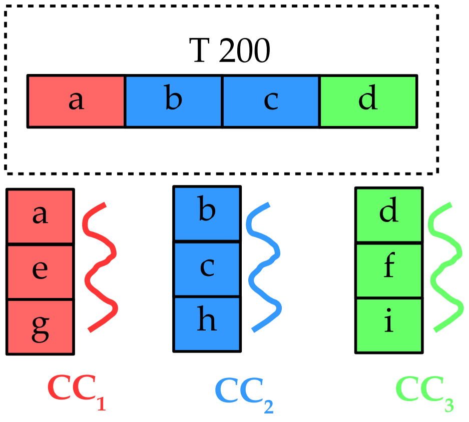


Figure 5.2: Intra-transaction parallelism. Transaction 200, which writes four records is shown in the upper rectangle. The logical partitioning of concurrency control thread responsibility is shown below.

tion. The transaction is assigned a timestamp of 200, and its write set consists of records a, b, c, and d. The concurrency control layer partitions records among three threads, CC_1, CC_2, and CC_3. CC_1's partition contains record a, CC_2's partition contains records b and c, and CC_3's partition contains record d. CC_1 thus inserts a new version for record a, CC_2 does the same for records b and c, and CC_3 for d. BOHM uses several threads to process a *single* transaction, a form of *intra-transaction* parallelism.

Every concurrency control thread must check whether a transaction's write set contains records that belong to its partition. For instance, if record d belonged to CC_1's partition instead of CC_3's, CC_3 would still have to check the transaction's write set in order to determine that no records in the transaction's write set map to its partition.

This design is consistent with our philosophy that concurrency control threads never need to coordinate with each other to process a transaction. Each record is always processed by the same thread (as long as the partitioning is not adjusted); two concurrency control threads will never try to process the same record, even across transaction boundaries. The decision of which records of a transaction's write set to process is a purely thread local decision; a concurrency control thread processes a particular record only if the record's key resides in its partition.

Not only does this lead to reduced cache coherence traffic, but it also leads to multi-core scalability. As we dedicate more concurrency control threads to processing transactions, throughput increases for two reasons. First, each transaction is processed by a greater number of concurrency control threads, which leads to

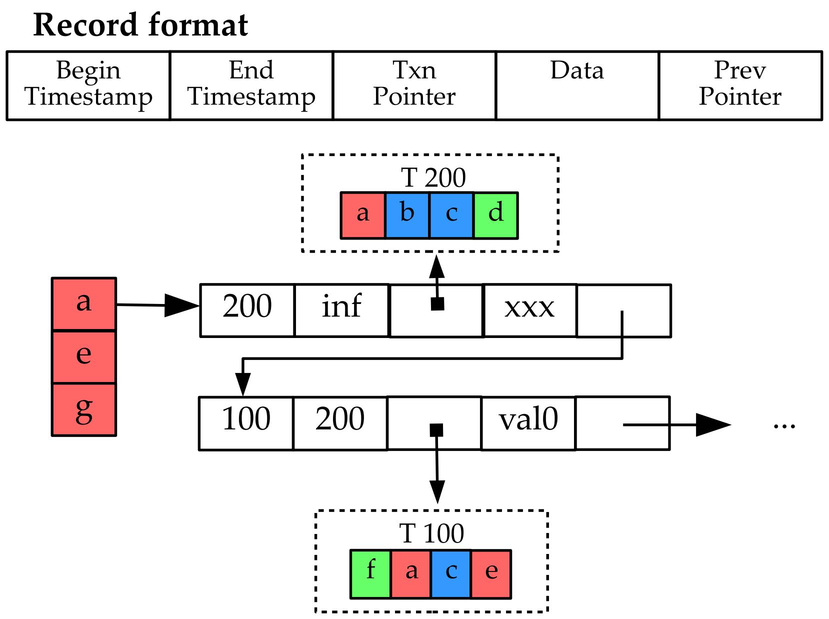


Figure 5.3: Inserting a new version

an increase in intra-transaction parallelism. Since concurrency control threads do not need to coordinate with each other, there is little downside to adding threads as long as there are enough processing resources on which they can run. Second, for a fixed database size, the number of keys assigned to each thread's partition *decreases*. As a consequence, each concurrency control thread will have a smaller cache footprint.

Processing a single transaction's read/write set

For each record in a transaction's **write set**, the concurrency control phase produces a new version to hold the transaction's write. Figure 5.3 shows the format of a record version. Each version consists of the following fields:

- **Begin Timestamp.** The timestamp of the transaction that created the record.
- **End Timestamp.** The timestamp of the transaction that invalidated the record.
- **Txn Pointer.** A reference to the transaction that must be executed in order to obtain the value of the record.
- **Data.** The actual value of the record.
- **Prev Pointer.** A reference to the previous version of the record.

When inserting a new version of a record, the concurrency control thread sets the version's fields as follows: (1) the version's start timestamp is set to the timestamp of the transaction that creates the version, (2) the version's end timestamp is set to infinity, (3) the version's txn pointer is set to the transaction that creates the version, (4) the version's data is left uninitialized, (5) the version's prev pointer is set to the preceding version of the record.

Figure 5.3 shows the thread CC_1 inserting a new version of record a, which is produced by transaction T_{200}. CC_1 sets the new version's begin timestamp to 200, its end timestamp to infinity, and its txn pointer is set to T_{200} (since T_{200} produces the new version). At this point, the version's data has not yet been produced; BOHM needs to execute T_{200} in order to obtain the value of the version.

While inserting a new version of record a, CC_1 finds that a previous version of the record exists. The older version of a was produced by transaction T_{100}. CC_1 sets the new version's prev pointer to the old version, and sets the old version's end timestamp to 200.

In order to create a new version of a record, BOHM does not need to synchronize concurrency control threads. BOHM partitions the database among concurrency control threads such that a record is *always* processed by the same thread, even across transaction boundaries (Section 5.3.2). One consequence of this design is that there is no contention in the concurrency control phase. The maintenance of the pointers to the current version of every record can be done in a thread-local data structure; thus the look-up needed to populate the prev pointer in the new versions is thread-local. Furthermore, if multiple transactions update the same hot record, the corresponding new versions of the record are written by the *same* concurrency control thread, thereby avoiding cache coherence overhead.

For each element in the transaction's **read set**, BOHM needs to identify the corresponding version that the transaction will read. In general, the concurrency control phase does not need to get involved in processing a transaction's read set. When an execution thread that is processing a transaction with timestamp ts wants to read a record in the database, it can find the correct version of the record to read by starting at the latest version of the record, and following the chain of linked versions (via the prev pointer field) until it finds a version whose $t_{begin} \leq ts$ and $t_{end} \geq ts$. If no such version exists, then the record is not visible to the transaction.

While the above-described technique to find which version of a record to read is correct, the cost of the traversal of pointers may be non-trivial if the linked list of versions is long. Such a situation may arise if a record is popular and updated often. An optimization to eliminate this cost is possible if the concurrency control phase has advanced knowledge of the read sets of transactions (in addition to the write set knowledge it already requires). In this case, for every record a transaction will read, concurrency control threads annotate the transaction with a reference to the correct version of the record to read. This is a low-cost operation for the concurrency control threads since the correct version is simply the most recent version at the time the concurrency control thread is running[2].

In particular, if a record in a transaction's read set resides on a concurrency control thread's logical partition, the thread looks up the latest version of the record and writes a reference to the latest version in a memory word reserved in advance within the transaction. The concurrency control thread *does not* track the read in the database, it merely gives the transaction a reference to the latest version of the record as of the transaction's timestamp.

A consequence of BOHM's design is that a transaction's reads do not require any contended writes to shared memory. Even for the read set optimization mentioned above, the write containing the correct version reference for a read is to pre-allocated space for the reference within a transaction, and is uncontended since only one concurrency control thread is responsible for a particular record. In contrast, pessimistic multi-version systems such as Hekaton [117] and Serializable Snapshot Isolation [64] need to coordinate a transaction's reads with concurrent transaction's writes in order to avoid serializability violations.

Batching

Only after a transaction T has been processed by all appropriate concurrency control threads can it be handed off to the transaction execution layer. One naïve way of performing this hand-off is for the concurrency control threads to notify each other after having processed each transaction by using synchronization barriers. After processing T, each concurrency control thread enters a global barrier in order to wait for all other threads to finish processing T. After all threads have entered this barrier, each concurrency control thread can begin processing the next transaction.

Unfortunately, processing transactions this way is extremely inefficient. Threads need to synchronize with each other on every transaction, which forces concur-

2. This is true since concurrency control threads process transactions sequentially (threads derive concurrency by exploiting intra-transaction parallelism).

rency control threads to effectively execute in lock step. Another issue is that some concurrency control threads are needlessly involved in this synchronization process. Consider a scenario where none of the records in T's write set belong to a concurrency control thread, CC's, partition. CC has to wait for every thread in order to move on to the next transaction even though CC contributes nothing to T's processing.

BOHM avoids expensive global coordination on every transaction, and instead amortizes the cost of coordination across large batches of transactions. The concurrency control thread responsible for allotting each transaction a timestamp accumulates transactions in a batch. The concurrency control threads responsible for creating versions receive an ordered batch of transactions, b, as input. Each concurrency control thread processes every transaction in b independently, without coordinating with other threads Once a thread has finished processing every transaction in b, it enters a global barrier, where it waits until all concurrency control threads have finished processing b, amortizing the cost of a single global barrier across every transaction in b.

Coordinating at the granularity of batches means that some threads may outpace others in processing a batch; a particular thread could be processing the 100^{th} transaction in the batch while another is still processing the 50^{th} transaction. Allowing certain concurrency control threads to outpace others is safe for the same reason that intra-transaction parallelism is safe (Section 5.3.2): BOHM partitions the database among concurrency control threads such that a particular record is *always* processed by the same thread, even across transaction boundaries.

5.3.3 Transaction Execution

After having gone through the concurrency control phase, a batch of transactions is handed to the transaction execution layer. The execution layer performs two main functions: it executes transactions' logic, and incrementally garbage collects versions that are no longer visible due to more recent updates.

Evaluating Transaction Logic

The concurrency control layer inserts a new version for every record in a transaction's write set. However, the *data* within the version cannot yet be read because the transaction responsible for producing the data has not yet executed; concurrency control threads merely insert placeholders for the data within each record.

Each version inserted by the concurrency control layer contains a reference to the transaction that needs to be evaluated in order to obtain the data of the version.

Read Dependencies. Consider a transaction T, whose read set consists of $r_1, r_2, ..., r_n$. T needs to read the correct version of each record in its read set using the process described in Section 5.3.2. However, the data stored inside one or more of these correct versions may not yet have been produced because the corresponding transaction has not yet executed. Therefore, an execution thread may not be able to complete the execution of T until the transaction upon which T depends has finished executing.

Write Dependencies. Every time the value of a particular record is updated, the concurrency control layer creates a new version of the record, stored separately from other versions. Consider two transactions T_1 and T_2, such that (1) neither transactions' logic contain aborts, and (2) T_1 is processed before T_2 by the concurrency control layer. Both transactions' write sets consist of a single record, x, while their read sets do not contain x. In this scenario, the concurrency control layer will write out two versions corresponding to x, one each for T_1's and T_2's update. The order of both transaction's updates is already decided by the concurrency control layer; therefore, T_1 and T_2's execution need not be coordinated. In fact, T_2 could execute before T_1, despite the fact that T_1 precedes T_2, and their write sets overlap[3].

We now describe how a set of execution threads execute a batch of transactions handed over from the concurrency control layer. The execution layer receives a batch of transactions in an ordered array $< T_0, T_1, ..., T_n >$. The transactions are partitioned among k execution threads such that thread i is responsible for ensuring transactions T_i, T_{i+k}, T_{i+2k}, and so forth are processed. Thread i does not need to directly execute all transactions that it is responsible for — other threads are allowed to execute transactions assigned to i, and i is allowed to execute transactions assigned to other threads. However, before moving onto a new batch of transactions, thread i must ensure that all transactions that it is responsible for in the current batch have been executed.

3. If T_2 performs a read-modify-write of x, then T_2 must wait for the version of x produced by T_1 before it can proceed with the read.

```
1  def execute_transaction(tx):
2
3    if not cmp_n_swap(tx.state,
4                      Unprocessed,
5                      Executing):
6      wait until tx.state == Complete
7      return
8
9    for dep_tx in tx.read_dependencies:
10     if dep_tx.state != Complete:
11       execute_transaction(dep_tx)
12
13   commit_decision = tx.run()
14   tx.state = Complete
15   return
```

Listing 5.1: Outline of transaction execution in BOHM.

Each transaction can be in one of three states: **Unprocessed**, **Executing**, and **Complete**. All transactions received from the concurrency control layer are in state **Unprocessed** — this state corresponds to transactions whose logic has not yet been evaluated. A transaction is in state **Executing** if an execution thread is in the process of evaluating the transaction. A transaction whose logic has been evaluated is in state **Complete**.

Listing 5.1 describes how execution threads run transactions. To process a transaction, T, an execution thread, E, attempts to atomically change T's state from **Unprocessed** to **Executing** (lines 3-5). E's attempt fails if T is in state **Executing** or **Complete**. If the attempt is successful, then E has exclusive access to T, since subsequent threads that try to change T's state from **Unprocessed** to **Executing** will fail.

If E discovers a read dependency on a version that has yet to be produced, it tries to recursively evaluate the transaction T' that T depends on (lines 9-11). After completing all reads and writes for T, E sets T's state to **Complete**.

Note that execution and concurrency control threads operate on different batches concurrently. Execution threads are responsible for producing the data associated with versions written in a batch, while concurrency control threads create versions and update the appropriate indexes. Logically, a version's data is a field associated with the version (Section 5.3.2, Figure 5.3). Execution threads only write a version's data field; therefore, there are no write-write conflicts between execution and concurrency control threads. However, in order to locate the record whose data must be read or written, execution threads may need to read database indexes. Execution threads need only coordinate with a single writer thread while reading an index – the concurrency control thread responsible for updating the index entry for that record. BOHM uses standard latch-free hash-tables to index data; readers need only spin on inconsistent or stale data [112]. We believe that coordinating structural modifications (SMOs) by a single writer with multiple readers is significantly less complex than coordinating multiple writers and readers. We leave the broader discussion of SMOs in general indexing structures to future work.

Garbage Collection

BOHM can be optionally configured to automatically garbage collect all versions that are no longer visible to any active or future transactions. Records that have been "garbage collected" can be either deleted or archived. This section describes how BOHM decides when a version can be safely garbage collected.

BOHM's execution layer receives transactions in batches. Transactions are naturally ordered across batches; if batch b_0 precedes batch b_1, then every transaction in b_0 precedes every transaction in b_1. Assume that a transaction T belongs to batch b_0. T updates the value of record r, whose version is v_1, and produces a new version v_2. The timestamp of any transaction in batch b_i, where $i \geq 1$, will always exceed v_2, T's timestamp. As a consequence, version v_1, which precedes v_2, will never be visible to transactions in batches that occur after b_0. Section 5.3.3 explained that execution threads always process batches sequentially; that is, each thread will not move onto batch b_{i+1} until the transactions it is assigned in b_i have been executed. Therefore, v_1 can be garbage collected when every execution thread has finished executing batch b_0. This condition holds regardless of which batch v_1 was created in. In fact, v_1 may even have been created in b_0. Whenever a transaction in batch b_i updates the value of a particular record, we can garbage collect the preceding version of the record when every execution thread has finished processing every transaction in batch b_i.

Garbage collection based on the preceding intuition is amenable to an efficient and scalable implementation based on read-copy-update (RCU) [126]. The heart of the technique is maintaining a global low-watermark corresponding to the minimum batch of transactions processed by every execution thread. Each execution thread t_i maintains a globally visible variable $batch_i$, which corresponds to the batch most recently executed by t_i. $batch_i$ is only updated by t_i. We assign one of the execution threads the responsibility of periodically updating a global variable $lowwatermark$ with $min(batch_i)$, for each i.

5.4 Experimental Evaluation

BOHM's primary contribution is a multi-version concurrency control protocol that is able to achieve serializability at lower cost than previous multi-version concurrency control protocols. We compare BOHM's performance to two state-of-the-art multi-versioned protocols: the optimistic variant of Hekaton [117], and Snapshot Isolation (implemented within our Hekaton codebase) [55]. Our Hekaton and Snapshot Isolation (SI) implementations include support for commit dependencies, an optimization that allows a transaction to speculatively read uncommitted data. In order to keep our codebase simple, our Hekaton and SI implementations do not incrementally garbage collect versions from the database and use a simple fixed-size array index to access records (BOHM and its other comparison points discussed below use dynamic hash-tables). The lack of garbage collection does not negatively impact performance; on the contrary, garbage collection was cited as one of the primary contributors to Hekaton's poor performance relative to single-versioned systems. BOHM runs with garbage collection enabled; therefore, any performance gains of BOHM over Hekaton and SI that we see in our experiments are conservative estimates. We would expect an even larger performance difference had garbage collection of these baselines been turned on.

While Hekaton and SI are the main points against which we seek to compare BOHM, our evaluation also includes single-version baselines. We compare BOHM against state-of-the-art optimistic concurrency control (OCC) and two-phase locking (2PL) implementations. Our OCC implementation is a direct implementation of Silo [159] – it validates transactions using decentralized timestamps and avoids *all* shared-memory writes for records that were only read. All our optimistic baselines — single-version OCC, Hekaton, and SI — are configured to retry transactions in the event of an abort induced by concurrency control.

Our 2PL implementation uses a hash-table to store information about the locks acquired by transactions. Our locking implementation has three important properties. 1. **Fine-grained latching.** We use per-bucket latches on the lock table to avoid a centralized latch bottleneck. 2. **Deadlock freedom.** We exploit advance knowledge of transactions' read and write sets to acquire locks in lexicographic order. Acquiring locks in this fashion is guaranteed to avoid deadlocks. Consequently, our locking implementation does not require any deadlock detection logic. 3. **No lock table entry allocations.** We exploit advance knowledge of a transaction's read- and write sets to allocate a sufficient number of lock table entries *prior* to submitting the transaction to the database. The consequence of this design is that the duration for which locks are held is reduced to the bare minimum.

Our experimental evaluation is conducted on a single 40-core machine, consisting of four 10-core Intel E7-8850 processors and 128GB of memory. Our operating system is Linux 3.9.2. All experiments are performed in main-memory, so secondary storage is not utilized for our experiments.

In all our implementations, there is a 1:1 correspondence between threads and cpu cores; we pin long-running threads to cpu cores. Traditional database systems typically assign a transaction to a single physical thread. If the transaction blocks, for instance, while waiting for lock acquisition or disk I/O, the database yields the thread's processor to other threads with non-blocked transactions. In order to adequately utilize processing resources when transactions block, the database ensures that there are enough threads running other non-blocked transactions. The number of active threads is therefore typically larger than the number of physical processors. In contrast, transactions in single-node main memory database systems do not block on I/O. Therefore, some main memory database systems use non-blocking thread implementations such that when a transaction blocks for any reason (such as a failure to acquire a lock), instead of yielding control to another thread, the thread temporarily stops working on that transaction and picks up another transaction to process, eventually returning to the original transaction when it is no longer blocked [145]. We leverage this approach in our implementations, so that all baselines we experiment with do not need to pay thread context switching costs.

5.4.1 Concurrency control scalability

We begin our experimental evaluation by exploring BOHM's separation concurrency control from transaction execution. Concurrency control and transaction ex-

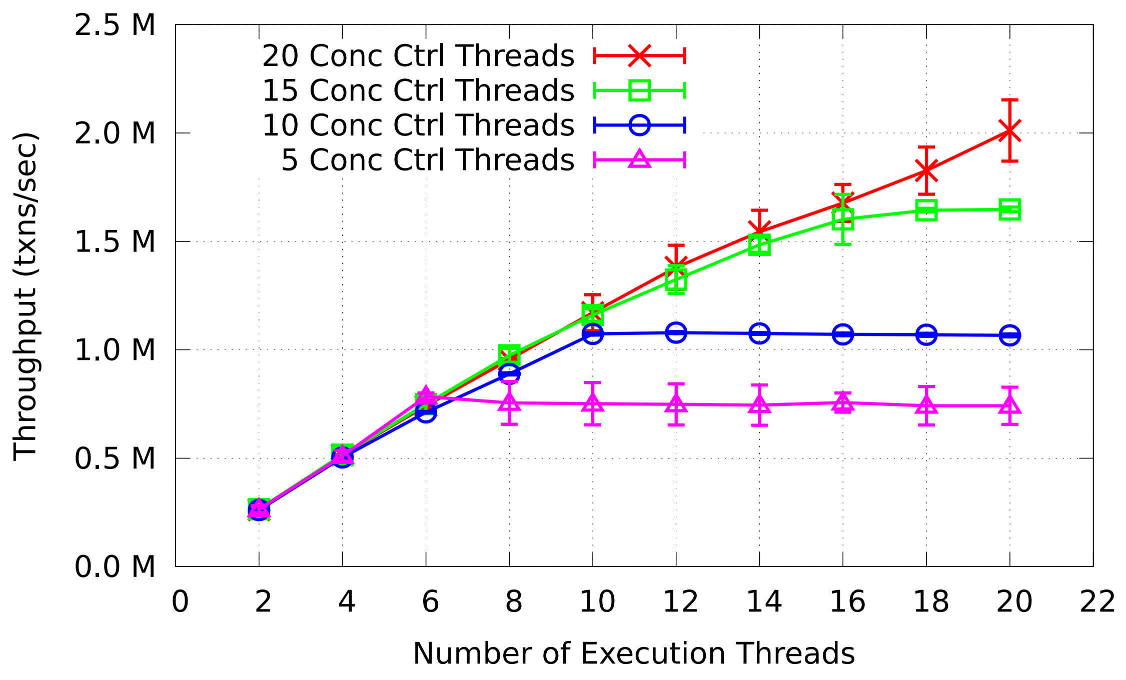


Figure 5.4: Interaction between concurrency control and transaction execution modules.

ecution are handled by two separate modules, each of which is parallelized by a separate group of threads. The number of threads in each module are system parameters that can be varied by a system administrator. We vary both parameters in this experiment.

Our experiment stresses the concurrency control layer in order to test scalability. In particular:

- The workload consists of short, simple transactions, involving only 10 RMWs of different records. Furthermore, each record is very small (it only contains a single 64-bit integer attribute), and the modification that occurs in the transaction consists of a simple increment of this integer. As a consequence, the execution time of each transaction's logic is very small.

- The database consists of 1,000,000 records, and the 10 records involved in the RMWs of each transaction are chosen from a uniform distribution. As a consequence, transactions rarely conflict with each other.

- The entire database resides in main memory, so there are no delays to access secondary storage.

As a result of these three characteristics, there are no delays around contending for data, waiting for storage, or executing transaction logic. This stresses the concurrency control layer as much as possible — it is not able to hide behind other bottlenecks and delays in the system, and must keep up with the transaction execution layer, which consumes very little effort in processing each transaction.

Figure 5.4 shows the results of our experiment. The number of threads devoted to transaction execution is varied on the x-axis, while the number of threads devoted to concurrency control is varied via the 4 separate lines on the graph. Recall that in our experimental setup, there is a 1:1 correspondence between threads and CPU cores. Thus, adding more threads to either concurrency control or transaction execution is equivalent to adding more cores dedicated to these functions.

Despite the extreme stress on the concurrency control layer in this microbenchmark, when the number of concurrency control threads (cores) significantly outnumber the number of execution threads (cores), the system is bottlenecked by transaction execution, not concurrency control. This is why the throughput of each configuration initially increases as more execution threads are added. However, once the throughput of the execution layer matches that of the concurrency control layer, the total throughput plateaus. At this point, the throughput of the system is bottlenecked by the concurrency control layer.

As we increase the number of concurrency control threads (represented by the four separate lines in Figure 5.4), the maximum throughput of the system increases. This indicates that the concurrency control layer's throughput scales with increasing thread (core) counts. This is because the concurrency control layer is able to exploit greater intra-transaction parallelism and has a lower per-thread cache footprint at higher core counts (Section 5.3.2).

While the number of concurrency control and execution threads can be varied by a system administrator, the choice of the optimal division of threads between the concurrency control and execution layers is non-trivial. As Figure 5.4 indicates, using too few concurrency control threads under-utilizes execution threads, while using too many concurrency control threads constrains overall throughput because too few execution threads are available to process transactions.

This problem can be addressed by techniques for dynamic load balancing in high-performance web-servers. BOHM uses a staged event-driven architecture (SEDA) [167]; the concurrency control and execution phases each correspond to a stage. The processing of a single request (in BOHM's case, a transaction) is divided between the concurrency control and execution phases. As advocated by

SEDA, there is a strong separation between the concurrency control and execution phases; threads in the concurrency control phase are unaware of threads in the execution phase (and vice-versa). SEDA's design allows for dynamic allocation of threads to stages based on load. Following SEDA's design principles, BOHM can dynamically allocate resources to the concurrency control and execution phases.

Overall, this initial experiment provides evidence of the scalability of BOHM's design. As we increase the number of concurrency control and execution threads in unison, the overall throughput scales linearly. At its peak in this experiment, BOHM's concurrency control layer is able to handle nearly 2 million transactions a second (which is nearly 20 million RMW operations per second) — a number that (to the best of our knowledge) surpasses any known real-world transactional workload that exists today.

5.4.2 YCSB

We now compare BOHM's throughput against the implemented baselines of Hekaton, Snapshot Isolation (SI), OCC, and locking on the Yahoo! Cloud Serving Benchmark (YCSB) [69].

For this set of experiments, we use a single table consisting of 1,000,000 records, each of size 1,000 bytes (the standard record size in YCSB). We use three kinds of transactions: the first performs 10 read-modify-writes (RMWs) — just like the experiment above, the second performs 2 RMWs and 8 reads (which we call 2RMW-8R), and the third is a read-only transaction which reads 10,000 records.

We use a workload consisting of only 10 RMW transactions to compare the overhead of multiversioning in BOHM compared to a single versioned system. If a workload consists of transactions that perform only RMW operations, we do not expect to obtain any benefits from multiversioning. To understand why, consider two transactions T_1 and T_2 whose read- and write sets consist of a single record, x. Since both transactions perform an RMW on x, their execution must be serialized. Either T_1 will observe T_2's write or vice-versa. This serialization is equivalent to how a single version system would handle such a conflict.

Under high contention multi-versioned systems will execute a workload of 2RMW-8R transactions with greater concurrency than single-versioned systems. The reason is that if a transaction, T, only reads the value of record r, then T does not need to block a transaction T' that writes r (or alternatively, performs an RMW operation on r).

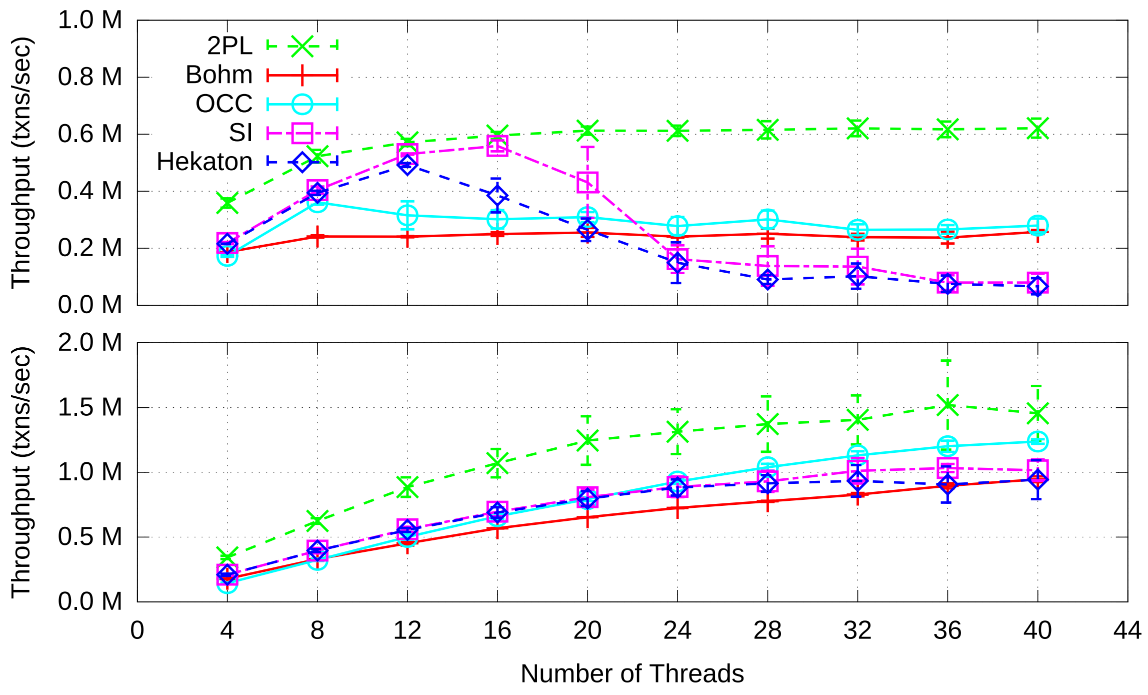


Figure 5.5: YCSB 10 RMW throughput. Top: High Contention (theta = 0.9). Bottom: Low Contention (theta = 0.0).

Finally, we use a workload consisting of a combination of 10RMW and read-only transactions to demonstrate the impact of long running read-only transactions on each of our baselines. We expect such a workload to favor multi-version systems because multi-version systems ensure that read-only transactions execute without blocking conflicting update transactions (and vice-versa).

10 RMW Workload

Our first experiment compares the throughput of each system on YCSB transactions which perform 10 RMW operations, where each element of a transaction's read and write set is unique. We run the experiment under both low and high contention. We use a zipfian distribution to generate the elements in a transaction's read and write sets. We vary the contention in the workload by changing the value of the zipfian parameter theta [90]. The low contention experiment sets theta to 0, while the high contention experiments sets theta to 0.9.

The graph at the top of Figure 5.5 shows the result of our experiment under high contention. The throughput of every system does not scale beyond a certain threshold due to the high contention in the workload — there are simply not enough non-conflicting transactions that can be run in parallel. Hekaton and SI perform particularly poorly because are prone to large numbers of aborts under high contention. Optimistic systems run transactions concurrently, regardless of the presence of conflicts, and validate that transactions executed in a serializable fashion (or in the case of SI, that write-write conflicts are absent and that transactions read a consistent snapshot of the database). A transaction is aborted if its validation step fails, and the work performed by the transaction is effectively wasted. Note, however, that while OCC is also optimistic and suffers from aborts; it does not suffer from the same drop in throughput as Hekaton and SI. This is because Silo (the version of OCC we use in these experiments) uses a back-off scheme to slow down threads when there is high write-write contention.

The locking system outperforms BOHM because individual transactions are subject to greater overhead. When a multi-version database system performs an RMW operation on a particular record (say, x), the corresponding execution thread must bring the memory words corresponding to x's version into cache, and write a *different* set of words corresponding to the new version of x. In contrast, when a single-version system performs an RMW operation, it writes to the same set of memory words it reads. The overhead of creating new versions must be paid during a transaction's contention period [4]. Thus, version creation has a greater negative impact on throughput in high contention workloads (as compared to low contention workloads). This effect is magnified for YCSB, since the size of each YCSB record is fairly large (1,000 bytes), and each transaction must therefore pay the overhead of writing ten new 1,000-byte records. Thus, all the multi-versioned systems (including BOHM) have a disadvantage on this workload — they pay the overhead of multi-versioning without getting any benefit of increase in concurrency for this 100% RMW benchmark.

Nonetheless, BOHM's lack of aborts allows it to achieve over twice the throughput of the other multi-versioned systems (Hekaton and SI) when there is a large number of concurrently running threads. Under low thread counts there is less contention and the optimistic systems do not suffer from many aborts. Furthermore, the high theta increases the number of versions created and ultimately garbage collected for the "hot" records, and our configuration of Hekaton and SI to not have to garbage collect give them a small advantage over BOHM.

We find that OCC's throughput begins to degrade between 8 and 12 threads,

4. A transaction's contention period is the time period during which concurrently running conflicting transactions must either block or abort.

while Hekaton and SI can sustain higher throughput for slightly higher thread counts (12 and 16 threads respectively). The reason is that Hekaton and SI use an optimization that allows transactions to speculatively read uncommitted values (commit dependencies) [117].

The graph at the bottom of Figure 5.5 shows the same experiment under low contention. Locking once again outperforms the other concurrency control protocols; however, the difference is much smaller. Locking still outperforms OCC because most OCC implementations, including our implementation of Silo's OCC [159], require threads to buffer their writes locally prior to making writes visible in the database. Locking does not pay the overhead of copying buffered writes to database records. While OCC's write buffering is similar to the multi-version systems' requirement of creation of new versions, it has lower overhead because the same local write buffer can be re-used by a single execution thread across many different transactions (leading to better cache locality of the local write buffers). In contrast, the multi-version systems need to write *different* locations on every update.

Under low contention, the multi-version systems – BOHM, Hekaton, and SI – have similar performance. Hekaton and SI marginally outperform BOHM, since our implementations of Hekaton and SI do not include garbage collection and use array-based indices to access records.

2RMW-8R Workload

This section compares BOHM's throughput with each of our baselines on a workload where each YCSB transaction performs two RMWs and eight reads (2RMW-8R). In a high contention setting, we expect that the multi-versioned systems will obtain more concurrency than the single-versioned systems, since the reads and writes of the same data items often do not conflict. In particular, under SI, reads and writes never conflict. Therefore, it is theoretically able to achieve more concurrency than any of the other systems that guarantee serializability since they have to restrict, to some degree, reads and writes of the same data items to avoid the write-skew anomaly (see Section 5.2). In particular, BOHM allows writes to block reads, but reads never block writes. In Hekaton, reads also never block writes, but writes can cause transactions that read the same data items to abort. In the single-version systems, reads and writes always conflict either via blocking (2PL) or aborting (OCC).

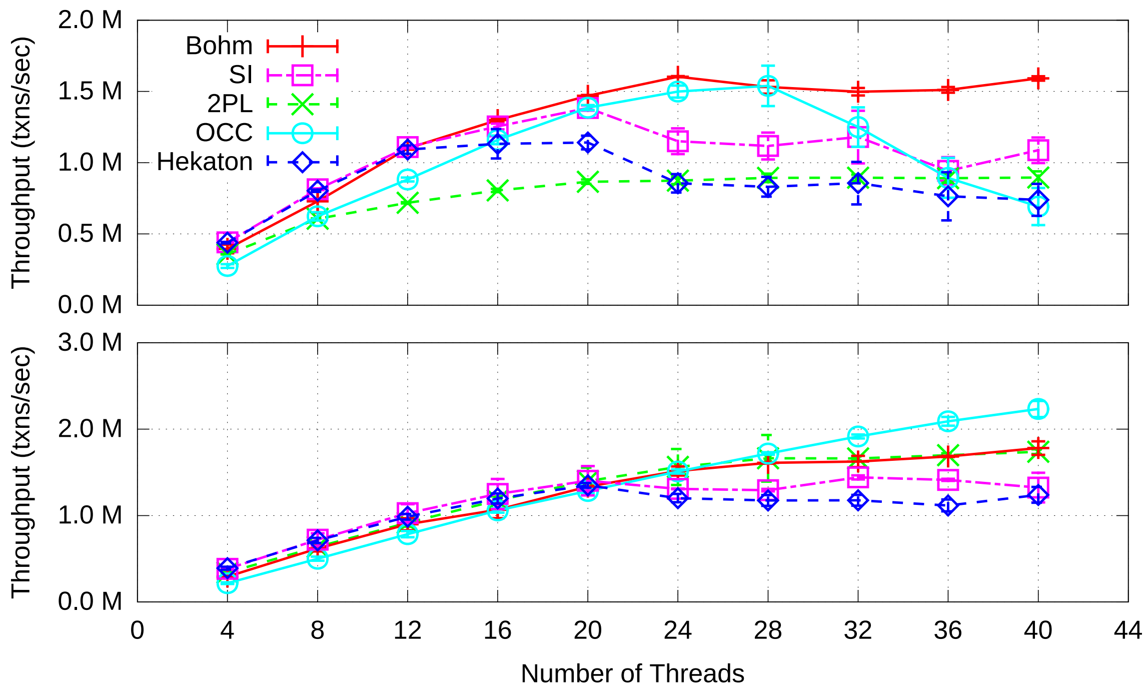


Figure 5.6: YCSB 2RMW-8R throughput. Top: High Contention (Theta = 0.9). Bottom: Low Contention (Theta = 0.0).

The graph at the top of Figure 5.6 shows the results of this experiment under high contention. As expected, the multi-versioned implementations outperform the single-versioned implementations due to their ability to achieve higher concurrency, and SI outperforms most of the other systems due to the larger amount of concurrency possible when serializable isolation is not required.

Surprisingly, however, BOHM significantly outperforms SI. We attribute this difference to aborts induced by write-write conflicts in SI. Under high contention this can lead to many aborts and wasted work. Meanwhile, BOHM specifies the correct ordering of writes to the same record across transactions in the concurrency control layer, so that the transaction processing layer simply needs to fill in placeholders and never needs to abort transactions due to write-write conflicts (Section 5.3.3). Like SI, Hekaton also suffers from aborts and wasted work under high contention. Interestingly, the Hekaton paper also implements a pessimistic version of its concurrency control protocol, but finds that it performs worse than the optimistic version, even under high contention. This is because in the pessimistic version, reads acquire read locks, and thus conflict with writes to the same record, thereby reducing concurrency. Thus, a major contribution of BOHM relative to Hekaton is a solution for allowing reads to avoid blocking writes without

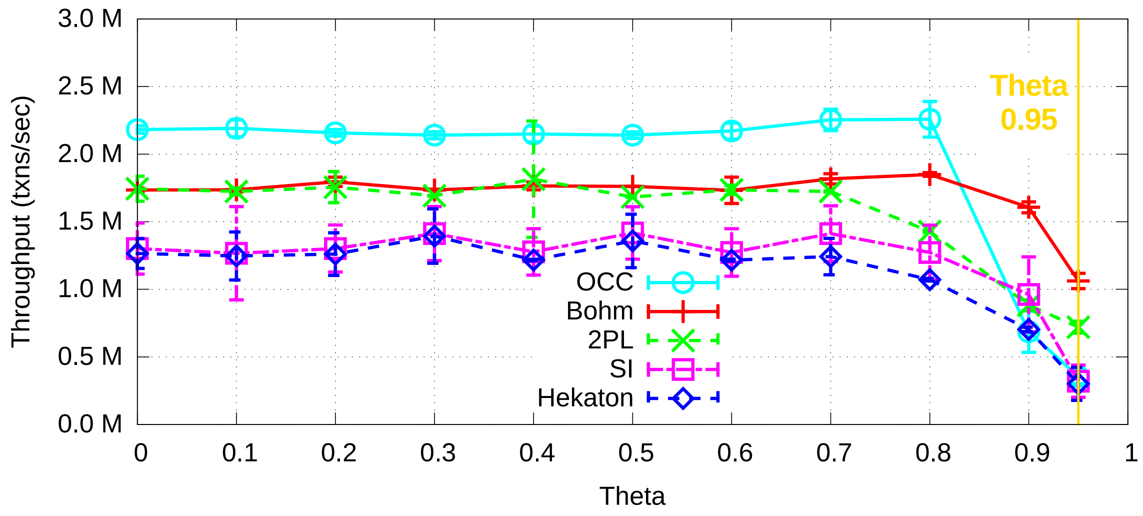


Figure 5.7: YCSB 2RMW-8R throughput varying contention

resorting to optimistic mechanisms.

The graph at the bottom of Figure 5.6 shows the same experiment under low contention. OCC outperforms both BOHM and locking as it employs a light-weight concurrency control protocol, and does not suffer from aborts under low contention. In particular, this workload contains a significant number of reads, and read validation is very cheap in Silo (which is our OCC implementation) [159]. Note however, that BOHM is very close in performance to OCC, despite the additional overhead of maintaining multiple versions.

The slope of the OCC, locking, and BOHM lines all decrease at higher thread counts. We attribute this to the fact that our database tables span multiple NUMA sockets.

The most interesting part of the bottom of Figure 5.6 is the comparison of the three multi-versioning implementations. With no contention, there are very few aborts in optimistic schemes, nor any significant differences between the amount of concurrency between the three schemes. Therefore, one might expect all three implementations to perform the same. However, we find that this is not the case; Hekaton and SI are unable to scale beyond 20 cores. We attribute Hekaton and SI's limited scalability to contention on global transaction timestamp counter. Hekaton and SI use a global 64-bit counter to assign transactions their begin and end timestamps. In order to obtain a timestamp, both systems atomically increment the value of the counter using an atomic *fetch-and-increment* instruction (`xaddq` on our x86-64 machine). The counter is incremented at least twice for every trans-

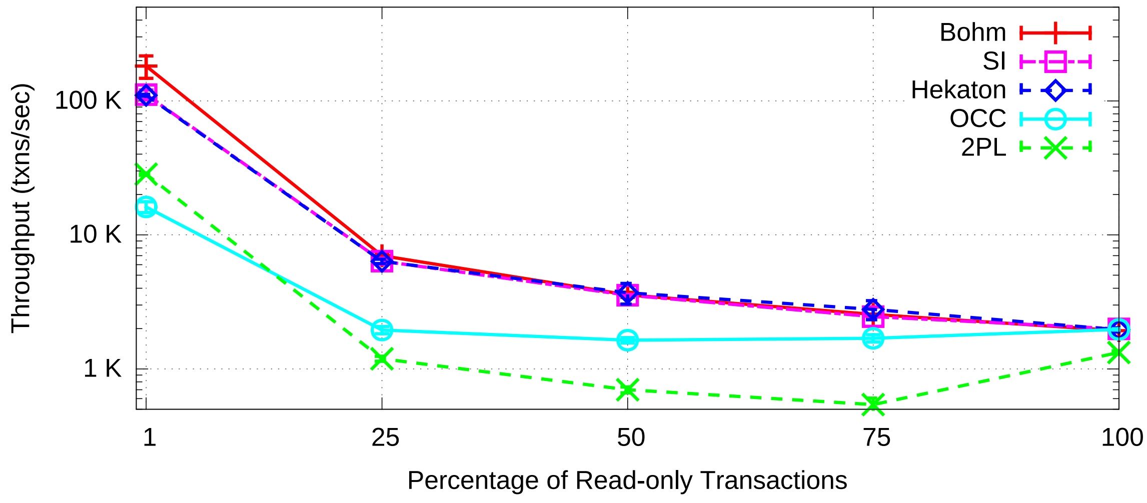


Figure 5.8: YCSB throughput with long running read-only transactions.

action, regardless of the presence of actual conflicts [5]. At high thread counts, SI and Hekaton are bottlenecked by contention on this global counter. This observation indicates that architectures that rely on centralized contended data-structures are fundamentally unscalable. BOHM's avoidance of this prevalent limitation of multi-version concurrency control protocols is thus an important contribution.

Figure 5.7 further illustrates the fundamental issue with Hekaton and SI's inability to scale. The graph shows the throughput of each system (at 40 threads) while varying the degree of contention in the workload. We use the same 2RMW-8R workload. The graph indicates that both Hekaton and SI have identical performance under low to medium contention, as they are both limited by the timestamp counter bottleneck. Only under high contention does a new bottleneck appear, and prevents the timestamp counter from being the primary limitation of performance.

Impact of Long Read-only Transactions

In this section, we measure the effect of long running read-only transactions on each of our baselines. We run each baseline on a workload consisting of a mix of update and read-only transactions. Update transactions are the low contention 10RMW YCSB transactions from Section 5.4.2. Read-only transactions read 10,000 records – chosen uniformly at random – from the database.

Figure 5.8 plots the overall throughput of each system while varying the frac-

5. The counter may be incremented more than twice if a transaction needs to be re-executed due to a concurrency control induced abort.

tion of read-only transactions in the workload. When a small fraction of the transactions are read-only (1%), we find that the multi-version systems outperform OCC and locking by about an order of magnitude (the y-axis uses a log-scale). This is because single-version systems cannot overlap the execution of read-only transactions and update transactions. In the multi-versioned systems, read-only transactions do not block the execution of conflicting update transactions (and vice-versa) because read-only transactions can perform their reads as of a timestamp that precedes the earliest active update transaction. We also find that BOHM significantly outperforms Hekaton and SI. We attribute this difference to BOHM's read set optimization (Section 5.3.2), which ensures that BOHM can obtain a reference to the version of a record required by a transaction without accessing any preceding or succeeding versions. In contrast, in Hekaton and SI, if the version required by a transaction, v_i, has been overwritten, then the system must traverse the list of succeeding versions v_n, v_{n-1}, ..., v_{i+1} (where $n > i$) in order to obtain a reference to v_i. Version traversal overhead is not specific to our implementations of Hekaton and SI – it is inherent in systems that determine the visibility of each transaction's writes after the transaction has finished executing. Thus, version traversal overhead is unavoidable in all conventional multi-version systems.

As the fraction of read-only transactions increases, the throughput of each system drops. This is because each read-only transaction runs significantly longer than update transactions (read-only transactions read 10,000 records, while update transactions perform an RMW operation on 10 records). When the workload consists of 100% read-only transactions, all systems exhibit nearly identical performance, because the workload does not conflicts.

5.4.3 SmallBank Benchmark

Our final set of experiments evaluate BOHM's performance on the SmallBank benchmark [63]. This benchmark was used by Cahill et al. for their research on serializable multi-versioned concurrency control. SmallBank simulates a banking application. The application consists of three tables: (1) Customer, a table which maps a customer's name to a customer identifier; (2) Savings, a table whose rows contain tuples of the form $< CustomerIdentifier, Balance >$; (3) Checking, a table whose rows contain tuples of the form $< CustomerIdentifier, Balance >$. The application consists of five transactions: (1) $Balance$, a read-only transaction that reads a single customer's checking and savings balances; (2) $Deposit$, makes a de-

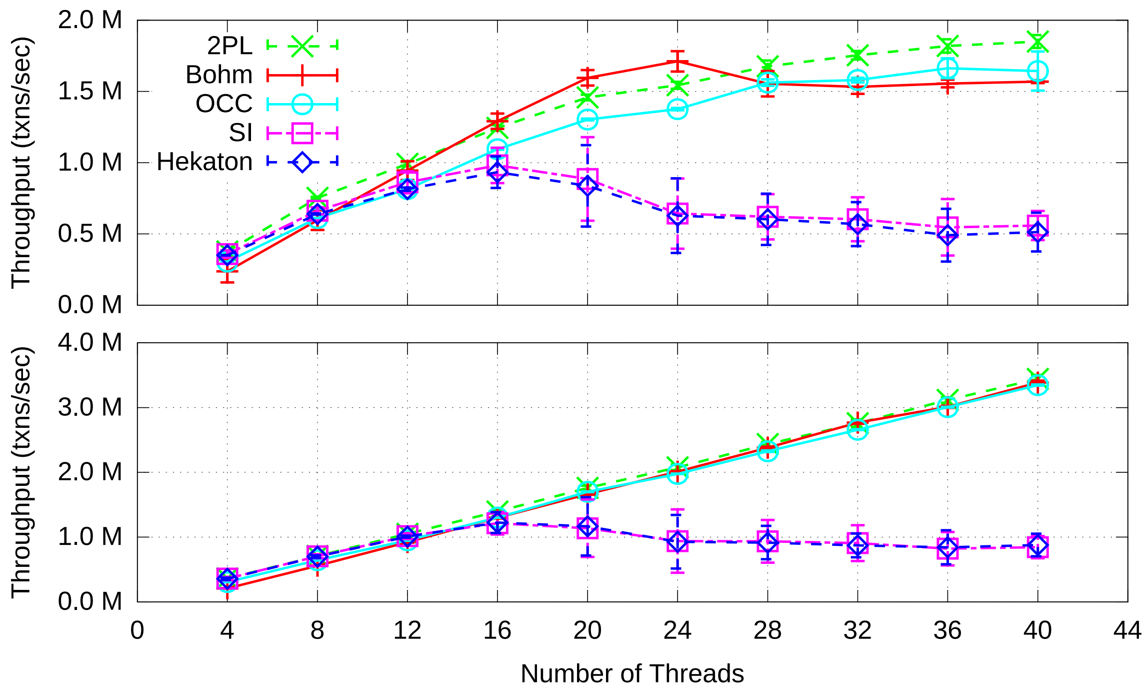


Figure 5.9: Small Bank throughput. Top: High Contention (50 Customers). Bottom: Low Contention (100,000 Customers).

posit into a customer's checking account; (3) $TransactSaving$, makes a deposit or withdrawal on a customer's savings account; (4) $Amalgamate$, moves all funds from one customer to another; (5) $WriteCheck$, which writes a check against an account. None of the transactions update the customer table — only the Savings and Checking tables are updated.

The number of rows in the $Savings$ and $Checking$ tables equals the number of customers in the SmallBank database. We can therefore vary the degree of contention in our experiments by changing the number of customers; decreasing the number of customers increases the degree of contention in the SmallBank workload.

The transactions in the SmallBank workload are much smaller than the transactions in the YCSB workload from the previous section. Every transaction reads and writes between 1 and 3 rows. Each record in the $Savings$ and $Checking$ tables is 8 bytes. In comparison, our configuration of the YCSB workload performs exactly 10 operations for each transaction, and each record is of size 1,000 bytes. In order to make the SmallBank transactions slightly less trivial in size, each transaction spins for 50 microseconds (in addition to performing the logic of the transaction).

Figure 5.9 shows the results of our experiment. The graph at the top of Figure 5.9 shows the results under high contention (the number of SmallBank customers is set to 50). Although locking once again performs best under high contention, the difference between locking and BOHM is not as large as in the contended 10RMW YCSB experiment (Section 5.4.2). There are two reasons for this difference:

First, as explained in Section 5.4.2, BOHM must pay the cost of bringing two *different* sets of memory words into cache on a read-modify-write operation, one corresponding to the version that needs to be read, the second corresponding to the version to be created. Since SmallBank's 8-byte records are smaller than YCSB's 1000-byte records, the cost of this extra memory access is smaller. Hence, the relative difference between BOHM and locking is smaller.

Second, the workload from Section 5.4.2 was 100% RMW transactions. In contrast, a small part of the SmallBank workload (20% of all transactions) consist of read-only $Balance$ transactions. Multi-versioned approaches such as BOHM are thus able to increase the concurrency of these transactions, since reads do not block writes.

Both Hekaton and SI's throughput drop under high contention due to concurrency control induced aborts. At 40 threads, SI outperforms Hekaton by about 50,000 transactions per second because it suffers from fewer aborts while validating transactions. Note that the abort-related drop in performance of Hekaton and SI is greater than OCC. This is because the contention on the timestamp counter for the multi-versioned schemes (Hekaton and SI) increases the time required to get a timestamp. Since the SmallBank transactions are so short, this time to acquire a timestamp is a nontrivial percentage of overall transaction length. Hence, the transactions are effectively longer for Hekaton and SI than they are for OCC, which leads to more conflict during validation, and ultimately more aborts.

The graph at the bottom of Figure 5.9 shows the results of the same experiment under low contention. We find that locking, OCC, and BOHM have similar performance under this configuration. As mentioned previously, the cost of RMW operations on SmallBank's 8-byte records is much smaller than RMW operations on YCSB's 1000-byte records.

As we saw in previous experiments (Section 5.4.2), we find that both Hekaton and SI are bottlenecked by contention on the global timestamp counter. When using 40 threads, BOHM is able to achieve throughput in excess of 3 million transactions per second, while Hekaton and SI achieve about 1 million transactions per second; a difference of more than 3x.

5.5 Conclusions

Most multi-versioned database systems either do not guarantee serializability or only do so at the expense of significant reductions in read-write concurrency. In contrast BOHM is able to achieve serializable concurrency control while still leveraging the multiple versions to ensure that reads do not block writes. Our experiments have shown that this enables BOHM to significantly outperform other multi-versioned systems. Further, for workloads where multi-versioning is particularly helpful (workloads containing a mixture of reads and writes at high contention), BOHM is able to outperform both single-versioned optimistic and pessimistic systems, without giving up serializability. BOHM is the first multi-versioned database system to accomplish this in main-memory multi-core environments.

Chapter 6

Decreasing the cost of serializability via piece-wise visibility

6.1 Introduction

Over the past decade, concurrency control research has seen a renaissance due to the abundance of parallelism in multi-core servers and datacenters. Modern serializable protocols are explicitly designed to exploit this abundant parallelism [76, 112–114, 117, 137, 138, 155, 157, 159]. While these new protocols propose novel isolation mechanisms that address the incompatibility between conventional concurrency control protocols and massively parallel environments, they use ideas for *recoverability* [58] that are decades old. Indeed, the last widely-adopted research on recoverability, group commit [87], was proposed in the 1980s. These conventional recoverability mechanisms limit concurrency control protocols' ability to extract concurrency from a workload.

Recoverability is the property that all of a committed transaction's writes are made durable, and that none of an aborted transaction's writes are made durable or observed by committed transactions [58]. In order to guarantee recoverability, most concurrency control protocols only permit a transaction's writes to be read after it commits or at least finishes executing [71, 87]. These protocols effectively delay making a transaction's writes visible. This *write visibility delay* can adversely impact strong isolation levels such as serializability. This is because serializable isolation requires that transactions always read the latest value of any record; any delay in satisfying a read will delay the corresponding reading transaction.

Recoverability mechanisms employ delayed write visibility because database systems can arbitrarily abort a transaction prior to the point that its commit record is made durable; a database system may abort a transaction due to deadlock handling logic, failures, optimistic validation errors, or simply because the transaction consumes resources that are in short supply. Database systems' ability to arbitrarily abort transactions forces recoverability mechanisms to make extremely pessimistic assumptions about when a transaction's writes are safe from being rolled back.

This chapter makes the case for curtailing database systems' ability to arbitrarily abort transactions. We show that if a database system only aborts transactions under a restricted set of conditions, then it can avoid pessimistic recoverability mechanisms based on delayed write visibility. In particular, if only a subset of a transaction's statements can cause it to abort, then the transaction is guaranteed to commit as soon as every such abortable statement has finished executing, even while one or more "non-abortable" statements remain to be executed. This enables a new write visibility discipline, *early write visibility*, which can safely make transactions' writes visible *prior* to the end of their execution, and, as a consequence, can reduce the duration for which concurrent transactions are disallowed from making progress due to conflicts.

This chapter proposes a new deterministic concurrency control protocol, piecewise visibility (PWV), explicitly designed to enable early write visibility. PWV employs deterministic execution to avoid arbitrarily aborting transactions. To enable early write visibility, PWV decomposes transactions into a set of sub-transactions or *pieces*, such that each piece consists of one or more transaction statements. PWV then schedules pieces such that their corresponding transactions execute in a serializable order. PWV makes a piece's writes visible as soon as its transaction commits, even if one or more pieces of the same transaction have not yet executed.

PWV decomposes transactions by performing a data-flow analysis on their control-flow graphs [47]. PWV's decomposition procedure has three important properties. First, it is modular; a transaction is decomposed based only on the data dependencies between its constituent statements. Second, it places no restrictions on the number of pieces that can potentially abort, while simultaneously preventing cascading aborts. Third, it allows PWV to exploit intra-transaction parallelism by executing multiple pieces belonging to the same transaction in parallel. These three properties address limitations that, to the best of our knowledge, are present in every prior transaction decomposition proposal [136, 151, 164, 170, 171, 173].

Our experimental evaluation shows that PWV's ability to produce aggressive serializable schedules results in significant performance gains. Under high contention workloads, PWV can outperform state-of-the-art serializable protocols, including transaction chopping, by over an order of magnitude. Furthermore, we show that PWV can even outperform a highly optimized implementation of read committed isolation by more than 3X, while still providing the stronger guarantee of serializable isolation.

In summary, this chapter makes the following contributions:

- We identify write visibility delay as a significant impediment to the performance of strong isolation levels due to the *specifications* for strong isolation. This impediment is fundamental and cannot be avoided by designing better concurrency control protocols (Section 6.2).

- We propose a new write visibility discipline, early write visibility, that addresses the limitations of prior write visibility disciplines. We show that early write visibility can enable concurrency control protocols that allow a transaction's writes to be made visible prior to the end of its execution while still guaranteeing serializability and preventing cascading aborts (Section 6.3).

- We design PWV, a concurrency control protocol that exploits early write visibility to obtain greater concurrency than conventional serializable protocols. We prove that if transactions' read- and write-sets are known a priori, it is impossible for any serializable concurrency control protocol that avoids cascading aborts to extract more concurrency from a workload than an ideal implementation of PWV. We also discuss practical issues related to application corner cases (Section 6.4).

- We evaluate a multi-core optimized implementation of PWV against state-of-the-art pessimistic locking, optimistic concurrency control, transaction chopping, and a weak isolation read committed implementation. (Section 6.5).

6.2 Background and motivation

You should say something about early lock release and controlled lock violation, as discussed in Graefe et al SIGMOD 2013.

In order to guarantee serializable and recoverable execution of transactions, every widely deployed concurrency control protocol disallows a transaction's writes from being read until at least the end of the transaction's execution. This write visibility delay is intrinsic to concurrency control protocols such as two-phase locking and optimistic concurrency control due to their use of locks and private writes prior to validation, respectively. Furthermore, the requirement that database systems guarantee recoverability fundamentally limits them from making writes visible early, *regardless of concurrency control protocol*.

6.2.1 Isolation

Variants of two-phase locking [78] and optimistic concurrency control [115] are among the most widely deployed serializable isolation protocols in modern database systems. In order to correctly isolate conflicting transactions, both strict two-phase locking (2PL) and optimistic concurrency control (OCC) restrict interleavings among conflicting transactions. In particular, if a transaction T_2 reads T_1's write to x, then practical implementations of both 2PL and OCC produce schedules in which T_2's read *always* follows T_1's completion. Under 2PL, transactions hold *long-duration* locks on records; any locks acquired by a transaction are only released at the end of its execution [55,88]. This locking discipline constrains the execution of conflicting reads and writes; if transaction T_2 reads T_1's write to record x, and T_1 holds a write lock on x until the time it completes, T_2's read can only be processed *after* T_1 completes.

OCC similarly constrains conflicting transactions. Transactions perform writes in a local buffer, and only copy these writes to the active database after validation [115]. Thus, a transaction's writes are only made visible at the very end of the transaction. Modern variants of OCC actually produce schedules of committed transactions that are provably equivalent to those produced by 2PL [159].

Both 2PL and OCC produce schedules in which there exists a delay between the time that a transaction writes a record, and the time that later transactions can read this write. This delay can significantly limit opportunities for concurrency under high contention.

6.2.2 Recoverability

Every transaction processed by a database system must either commit or abort. If a transaction commits, then all of its writes must be made persistent. In contrast, if a transaction aborts, its writes cannot be made persistent. Furthermore, most widely-used isolation levels – including Read Committed, Snapshot Isolation, and Serializability – require that an aborted transaction's writes must never be observed by a committed transaction [55]. If this is not the case, the committed transaction exhibits an aborted read anomaly [42].

In order to prevent aborted reads, concurrency control protocols must constrain the execution of transactions whose reads and writes conflict. Consider a transaction T_1 that writes record x. If another transaction, T_2, reads T_1's write to x, then T_2 must not be allowed to commit before T_1 commits. This discipline prevents T_1 from aborting after T_2 (which read T_1's data) has already committed. Schedules that satisfy this property are called recoverable [58]. Recoverable scheduling mechanisms must therefore control when a transaction's writes are made visible to other transactions. There exist two general write visibility disciplines:

- **Committed write visibility.** The database delays making a transaction's writes visible until the transaction is guaranteed to commit. Strict two-phase locking is one such strategy [58]. A transaction holds exclusive locks on a record it writes from the time it updates the record to the time the transaction commits. Holding exclusive locks until commit time prevents concurrent transactions from reading uncommitted writes.

- **Speculative write visibility.** Alternatively, the database system can allow transactions to read uncommitted writes (dirty reads), and enforce a commit discipline on transactions that perform dirty reads [44, 58, 93, 111, 142]. If transaction T writes a record and later aborts, then any transaction that read T's write also aborts.

Each of these write visibility disciplines has advantages over the other. Speculative write visibility is susceptible to cascading aborts [58]. If transaction T makes uncommitted writes visible to other transactions and later aborts, then any transaction T' that read T's uncommitted writes must abort. In turn, any transaction that read uncommitted writes by T' must also abort, and so forth. In general, if transaction T aborts, then the transitive closure of transactions linked via dirty-read dependencies to T must also abort. Cascaded aborts can severely hurt performance because the database wastes cycles executing transactions that are later aborted.

Committed write visibility avoids cascading aborts by disallowing dirty reads; transaction T is never allowed to make uncommitted writes visible to other transactions. On the other hand, committed write visibility can inhibit performance by forcing transactions to wait for prior transactions' commit records to be made durable. This delay can lead to unacceptable performance when transactions' runtimes are much shorter than the time it takes to make commit records durable, for instance, in main-memory database systems which maintain durable state on disk.

Due to these tradeoffs, modern database systems employ *hybrid* disciplines that combine committed and speculative write visibility [110, 117, 138, 159]. The best known example of such a hybrid write visibility discipline is group commit [71,87]. As originally proposed by Gawlick and Kinkade, transactions follow a locking protocol in which they hold locks for the duration of their execution, and release locks after their execution completes but before their commit records are made durable. Prior to releasing their locks, transactions write their commit records to an in-memory sequential log. The in-memory log is asynchronously flushed to disk in batches. Since a transaction T holds its locks until its commit record is written to the in-memory log, if transaction T' reads T's writes, then its commit record will be logged after T's commit record. This logging discipline guarantees recoverability; if transaction T' commits, then all transactions whose commit records were logged prior to T''s also commit, including those transactions whose writes were read by T'.

Most modern concurrency control protocols either use a form of committed write visibility or a variant of group commit based on delayed write visibility. In Section 6.2.3, we show that even the short delays in making writes visible in recoverability mechanisms such as group commit – compared to, for instance, delays with disk I/O on the critical path – can adversely impact serializable concurrency control protocols under contended workloads.

6.2.3 Interaction of write visibility and isolation

Database systems allow users to assign transactions an *isolation level*, which abstractly specifies the permissible set of interleavings among conflicting transactions [42]. Isolation levels allow each individual transaction to tradeoff consistency for performance. Strong isolation levels, such as serializability, permit fewer interleavings among conflicting transactions, which provides strong consistency at the expense of concurrency. In contrast, weak isolation levels, such as read committed, permit more interleavings among conflicting transactions, allowing transactions to observe inconsistent database states in order to improve performance.

One important restriction on interleavings is that serializability requires that transactions always observe the *latest* committed value of any record that they read. In contrast, read committed allows transactions to read *any* previously committed values of a record (reads can be arbitrarily stale). As a consequence, serializable concurrency control protocols must carefully constrain the execution of transactions whose reads and writes conflict, while read committed protocols can decouple conflicting reads and writes. Consider a scenario where record x is first written by transaction T_0, and next written by transaction T_1 (T_0 precedes T_1). If a later transaction T_2 reads x, then under serializability, T_2's read *must* return the value written by T_1. In contrast, read committed allows T_2 to read either of T_0 or T_1's writes.

As Section 6.2.2 discussed, recoverability mechanisms based on group commit only permit transactions' writes to be observed at the end of their execution. Serializability's requirement that transactions observe the latest committed values of records interacts poorly with the delayed write visibility discipline employed by these recoverability mechanisms. In the above example, if T_1's write to x is followed by additional writes to records y and z, then for recoverability purposes, T_1's write to x can only be read by T_2 after T_1's additional writes to y and z complete. In contrast, under read committed, the database can allow T_2 to read T_0's write to x even as T_1's writes are in progress. In order to guarantee recoverability, read committed must also delay making T_1's write to x visible until the end of its execution. However, this delay has no effect on T_2 because T_2 is permitted to read *earlier* transactions' writes to x.

Serializability's requirement that transactions observe the latest committed values of records is part of its *specification*. Therefore, every protocol that correctly implements this specification, that is, *every* serializable concurrency control protocol, is subject to the reduction in concurrency due to delayed write visibility. The fact that delayed write visibility limits concurrency cannot be circumvented by designing better protocols or more efficient implementations.

In order to substantiate this argument, we conducted an experiment to measure the interaction between write visibility delay and isolation levels. The experiment runs a workload consisting of transactions that perform 10 read-modify-write operations. The database consists of 1,000,000 records. We designate one record in the database as "hot", and force every transaction to update this hot record. As a con-

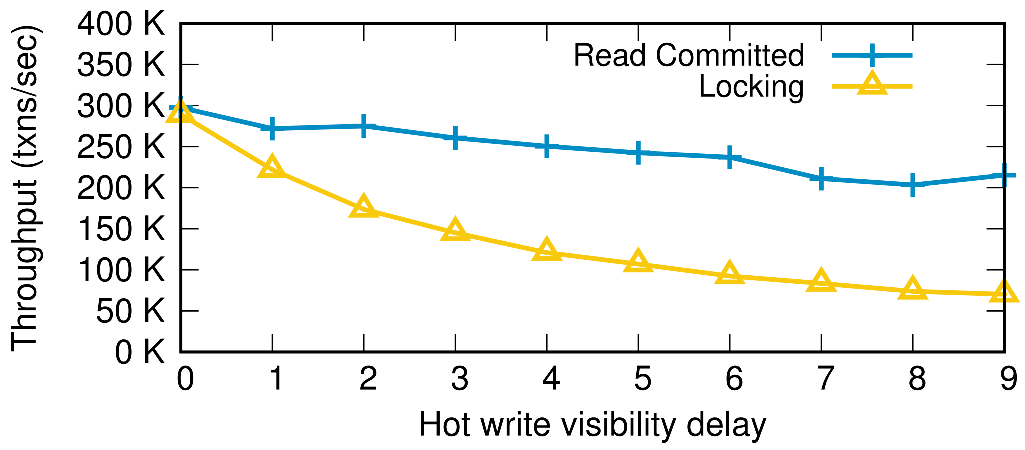


Figure 6.1: Effect of write visibility delay of hot record updates on transaction throughput using 40 threads.

sequence, every pair of transactions conflicts. The 9 remaining records updated by a transaction are chosen uniformly at random from the remaining 999,999 records. We compare the performance of a multi-core optimized implementation of serializable locking and read committed (Section 6.5 provides a detailed discussion of these algorithms and experimental setup).

In order to measure the impact of write visibility delay on each system, we vary the point at which each transaction updates the hot record. The earlier a transaction updates the hot record, the higher the write visibility delay. We measure write visibility delay as the number of updates that transactions must perform *after* updating the hot record. Figure 6.1 shows the results of the experiment. We plot the throughput of read committed and serializable locking as a function of increasing write visibility delay.

Strange way of explaining. The usual way is in terms of locking holding time. Delay access to hot data items at the end of txn to minimize lock hold time; e.g., Gawlick and Kinkade, or the perf section of 2PL in Bernstein and Newcomer. Figure 6.1 shows that locking's throughput decreases dramatically as visibility delay increases. The locking algorithm acquires an exclusive lock on a record prior to updating the record, and holds all acquired locks until the end of its execution. When the hot record update is performed at the end of each transaction (the left-most point on the x-axis), the lock on the hot record is only held while the transaction updates the hot record. In contrast, when the hot record update is performed at the beginning of each transaction (the right-most point on the x-axis), the lock on the hot record is acquired at the beginning of each transaction, and is held while the transaction updates every record in its write-set. Locking's throughput drops by nearly a factor 6 at maximum write visibility delay. In contrast, read committed's throughput drops by a more modest 30%. Increasing write visibility delay does not have as adverse an impact on read committed as serializability because read committed allows transactions to read stale values of records. The modest drop in throughput occurs because read committed acquires write locks on records at commit time in order to consistently order transactions' writes [42] (see Section 6.5). These commit time write locks are acquired in the same order as serializable locking, but held are held for a much shorter duration.

6.3 Early write visibility

We now describe a new recoverability mechanism, early write visibility, that addresses the limitations of delayed write visibility. Early write visibility constrains a database system's ability to arbitrarily abort transactions. Early write visibility can be tailored to any database system which sufficiently constrains transaction aborts to a specific set of circumstances, such as explicit abort statements and constraint violations. Examples of such systems include deadlock-free locking systems [144] and deterministic database systems [80,155,157]. In this chapter, we focus on deterministic database systems (although the ideas can be generalized to other systems).

6.3.1 Deterministic execution background

Transaction aborts can broadly be classified into logic- and system-induced aborts. Logic-induced aborts occur in order to prevent a transaction from writing state that violates application invariants. For example, a transaction may include an explicit abort statement that is conditionally triggered after reading a database record, or the transaction may be aborted if its updates cause a constraint violation. System-induced aborts are triggered by the database system, and occur independently of transactions' logic. Examples of system-induced aborts include aborts due to deadlock handling logic, failures, and validation errors in optimistic protocols.

Deterministic systems employ scheduling techniques that eliminate the vast majority of system-induced aborts in conventional systems. A deterministic system processes a transaction in the following three steps. First, any calls to non-deterministic functions, such as a random number generator or system clock, are evaluated in order to be used at execution time. Second, the transaction's logic, its input parameters, and all non-deterministic input are logged. Note that *all* transactions are logged, regardless of whether they eventually commit or abort. Third, the transaction is processed after its existence has been successfully (stably) logged. We next describe how deterministic systems execute transactions during normal case and recovery processing.

Normal case processing. Deterministic systems process transactions in an order that is equivalent to the order in which they are logged (as described above). Serializability is guaranteed by the fact that the log is totally ordered. A class of deterministic systems, exemplified by Calvin and Bohm [80, 157], use knowledge of transaction conflicts to relax the total order into an equivalent partial order. If transactions T_1 and T_2 conflict, such that T_1 precedes T_2 in the log, then T_1 will always be executed before T_2. The execution of non-conflicting transactions is not constrained.

These systems determine transactions' conflicts using a priori knowledge of transactions' read and write sets. The read and write sets are determined either via a static analysis of each transaction's logic, or via speculative execution of a subset of each transaction's logic (see below). These systems also use a priori knowledge of read and write sets to implement a deadlock avoidance strategy. For example, Calvin isolates transactions using a modified version of logical locking [157]. The scheduler acquires transactions' locks by sequentially scanning the input log. For every transaction in the log, the scheduler requests locks on *every* record in the transaction's read and write sets prior to its execution. A transaction is only permitted to execute when all of its locks have been acquired. This lock acquisition protocol avoids deadlocks because the set of locks required by transactions are known a priori and can be acquired in lexicographic order.

In certain applications, transactions' read and write sets are deducible from their input parameters, such as when all records involved in a transaction are accessed by their primary keys. In other applications, read and write sets depend on database state (such as secondary indexes). In the latter case, deterministic systems determine transactions' read and write sets using speculative execution. Speculative execution occurs as part of non-deterministic input processing prior to logging transactions (as described in the first step of transaction processing above). The obtained read and write sets are then logged along with transactions' other input parameters. At execution time, deterministic systems check that the speculatively obtained read and write sets are correct. This is done by adding a logical condition as early as possible in the transaction code to validate the speculatively generated read and write sets. If this condition fails, a deterministic logical abort results.

Recovery processing. Deterministic systems execute transactions only when they are guaranteed to be stable on the log [124]. Therefore, each in-progress transaction during a failure is guaranteed to be in the log. Furthermore, *all* transactions are logged, regardless of whether they eventually commit or abort. At recovery time, a deterministic system can play the log forward from the time of the last checkpoint. As mentioned previously, the only information contained in a log record is the transaction's logic, its input parameters, and any non-deterministic input. The log is played back in a serial-equivalent fashion using *the same* mechanism used during normal case processing; there is no difference between recovery and normal case processing. Since the non-deterministic inputs to a transaction are also logged, each transaction is guaranteed to deterministically write the same record values and arrive at the same commit decision during recovery.

6.3.2 A new write visibility discipline

We observe that the reduced scope of aborts in deterministic systems can be exploited to obtain a far more aggressive write visibility discipline than those used by conventional systems. Since deterministic systems only abort transactions due to logic- and speculation-induced aborts, a transaction is *guaranteed* to commit once all the operations that can cause logic- and speculation-induced aborts have finished executing. Importantly, this "commit point" can occur before the transaction has finished executing in its entirety. In other words, a transaction can have several operations pending after its commit point.

Early write visibility prescribes two write visibility rules; each applicable to writes that precede or follow a transaction's commit point:

- **Writes preceding the commit point.** Such writes can only be made visible once every other operation that precedes the transaction's commit point has finished executing. Delaying the visibility of these writes until a transaction's commit point ensures that they are never read by another transaction only to be later rolled back.

- **Writes following the commit point.** Such writes can be made visible immediately after their completion. A write that follows a transaction's commit point is guaranteed to never rollback because a transaction can never abort beyond its commit point.

The two rules above ensure that a transaction's writes are only made visible if it commits. Early write visibility therefore guarantees that a transaction never reads dirty data, which eliminates the possibility of a transaction reading a write that is later rolled back.

6.4 Piece-wise Visibility

Although early write visibility makes a transaction's writes visible as soon as it is guaranteed to commit, conventional concurrency control protocols such as 2PL and OCC cannot simply replace their recoverability mechanisms with early write visibility, for two reasons. First, delayed write visibility is intrinsic to both 2PL and OCC due to their respective use of locks and validation (Section 6.2.1). Second, existing concurrency control protocols use arbitrary transaction aborts pervasively; dynamic locking aborts transactions due to deadlocks, and OCC aborts transactions on validation failures. These arbitrary aborts preclude early write visibility, which requires that transactions are only aborted under a limited set of conditions (Section 6.3). While existing deterministic concurrency control protocols do not arbitrarily abort transactions (Section 6.3.1), they cannot exploit early write visibility because they schedule each transaction's logic as a single atomic unit [80,155,157].

This section presents piece-wise visibility, or PWV, a new deterministic serializable concurrency control protocol that schedules work at the granularity of subsets of transactions' individual reads and writes. This fine-grained scheduling allows PWV to fully exploit early write visibility. PWV *decomposes* the totally ordered set of statements that constitute a straight-line transaction into a partially ordered set of statements based on the transaction's data-flow and commit point. PWV then schedules each decomposed transaction's constituent statements using a deterministic scheduler. Intuitively, PWV can produce schedules that are similar to those produced by a locking-based concurrency control protocol that is *not two-phase*; that is, a protocol that releases locks on records, and then goes on to acquire more locks on different records later on, but nonetheless guarantees serializability.

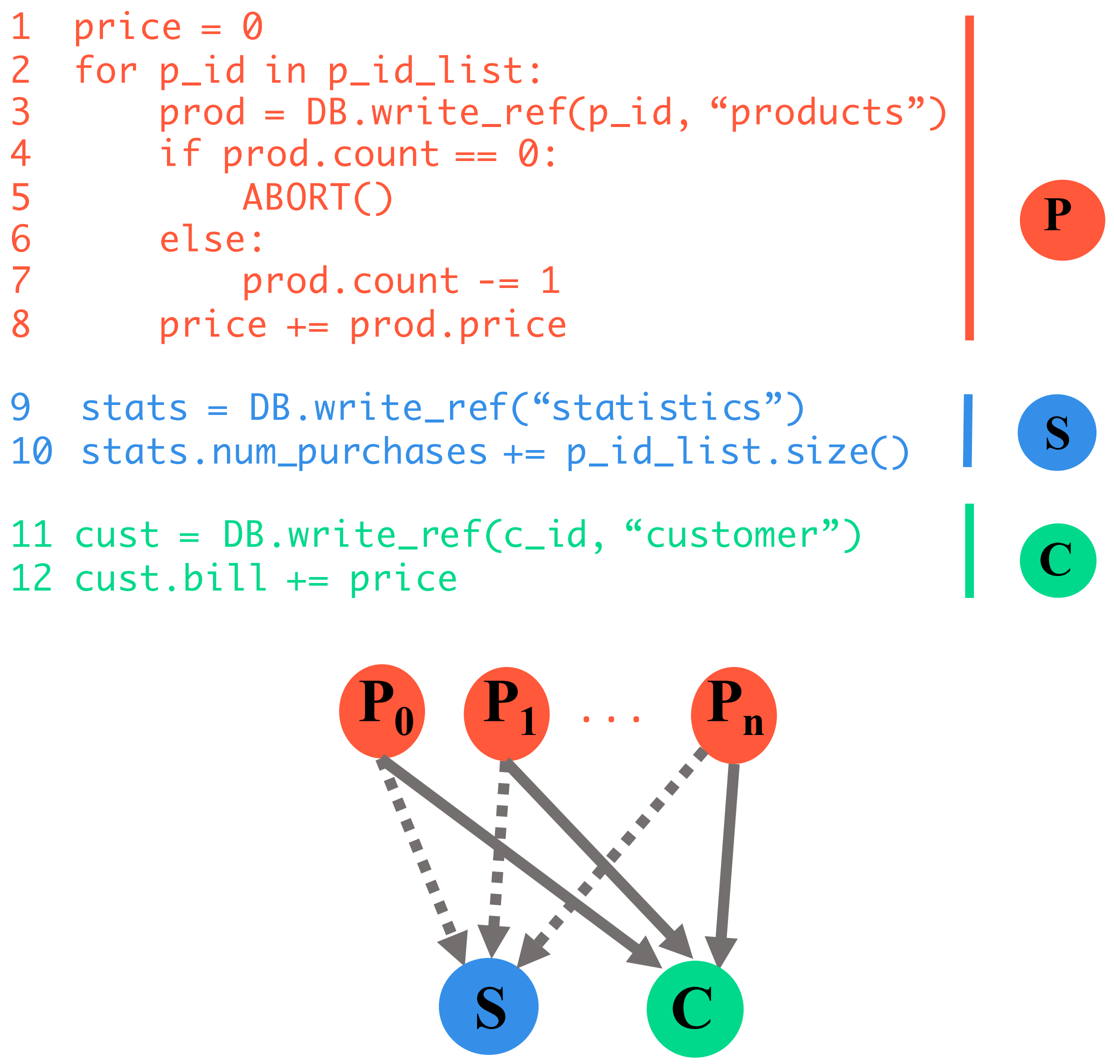
6.4.1 Transaction decomposition

The input to PWV's scheduler is a totally ordered set of decomposed transactions. A decomposed transaction is a partially ordered set of the transaction's constituent statements. This partially ordered set can be represented by a directed acyclic graph (DAG) whose nodes we refer to as *pieces*. The edges of the DAG define the order in which its pieces can execute.

There exist two situations under which an edge is created from piece p_1 to p_2. First, the input of p_2 depends on the output of p_1 (data dependencies). Second, p_2 contains an update statement, and follows its transaction's commit point, while p_1 precedes the commit point (commit dependencies). Intuitively, commit dependencies prevent pieces that follow a transaction's commit point from performing updates until the transaction is guaranteed to commit.

Figure 6.2 shows an example of transaction decomposition. The transaction shows logic that is invoked when a customer attempts to purchase a set of items from hypothetical online shopping portal. The transaction first tries to decrement the count of each requested item (lines 1-8). The transaction aborts if any of the item's counts is zero (lines 4-5). The transaction then updates some application-specific statistics, in this case, the total number of items sold (lines 9-10). Finally, the transaction updates the outstanding amount due from the customer (lines 11-12).

The transaction's decomposition is shown below the straight-line code (in Figure 6.2). Edges corresponding to data dependencies are represented by solid arrows. Edges corresponding to commit dependencies are represented by dashed arrows. The transaction can safely commit after the count of every item requested by the customer is successfully decremented. Each item count decrement is represented by a piece P_i. Piece S updates the total number of items sold, and only depends on the number of items the customer purchases. Importantly, the write in piece S does not depend on the output of any other piece. However, because it follows the transaction's commit point, S has a commit dependency on each P_i. In contrast, piece C, which updates the customer's outstanding bill, depends on the output of every piece P_i (note that C also has a commit dependency on each P_i). In particular, C's write depends on the sum of the price of each item. An item's price is only obtained after the execution of the corresponding piece P_i. Pieces that are not ordered via an edge or path in the graph can be executed concurrently. In particular, each item update piece P_i does not depend on any other item update



```
price = 0
for p_id in p_id_list:
    prod = DB.write_ref(p_id, “products”)
    if prod.count == 0:
        ABORT()
    else:
        prod.count -= 1
    price += prod.price

stats = DB.write_ref(“statistics”)
stats.num_purchases += p_id_list.size()

cust = DB.write_ref(c_id, “customer”)
cust.bill += price
```

Figure 6.2: A straight-line transaction decomposed into a partially ordered set of pieces. Solid edges represent data dependencies. Dashed edges represent commit dependencies.

piece.

In our current implementation, transactions are decomposed by hand. However, PWV's decomposition procedure can be made fully automatic, and thus would not require any developer effort. In its full generality, the algorithm for automatically decomposing transactions is beyond the scope of this thesis, but we describe a simplified algorithm that creates a piece for each transaction statement: First create one piece per unique record that is read or written by the transaction. Data dependencies between pieces can be created using the following three steps. First, construct a transaction's statement-level control flow graph (where each statement corresponds to a single piece). Second, perform a reaching-definitions analysis on the control flow graph [47]. Third, for every definition that reaches a particular statement, construct an edge to the piece from the piece that creates the definition. For commit dependencies, create an edge to a writing piece that follows a transaction's commit point from every abortable piece.

PWV's decomposition algorithm is *modular*; a particular transaction's decomposition does not depend on any other transaction. As a consequence, transactions can be decomposed on clients or by admission control prior to being submitted to PWV. For the remainder of this section, we assume that PWV takes transactions' decomposed pieces as input.

6.4.2 Rendezvous points

PWV must execute the pieces of a decomposed transaction in an order that is consistent with the DAG constructed via the analysis of a transaction's data and commit dependencies. PWV coordinates the execution of pieces whose execution must be ordered using *rendezvous points*, a mechanism for synchronizing a partially ordered set of transaction statements originally proposed by Pandis et al. in their work on data oriented transactions [140]. In addition PWV re-purposes rendezvous points (RVPs) to implement a lightweight transaction commit protocol [58].

Coordinating dependent pieces. PWV associates a single rendezvous point (RVP) with every piece that has at least one dependency. For instance, in the decomposed transaction described in Figure 6.2, PWV associates a RVP with the customer update piece C, since it depends on each product update piece P_i. Furthermore, two or more pieces can share a single RVP if they share the same set of parent pieces.

In Figure 6.2, both C and S have the same set of parent pieces; $P_0, ..., P_n$. C and S can therefore share a RVP.

A RVP is used to determine when *all* of a dependent piece's parents have finished executing. For this purpose, a RVP uses a counter whose value is initialized to the number of parent pieces of a particular dependent piece. Each parent piece contains a reference to this RVP, and decrements the RVP's counter when it completes executing. When the value of the counter reaches zero, the downstream pieces associated with the RVP are ready to execute. In Figure 6.2's decomposed transaction example, the RVP counter associated with C and S is initialized to $n+1$ (corresponding to its parent pieces pieces $P_0, ..., P_n$).

Committing transactions. PWV associates a single RVP with every abortable piece in a transaction. We refer to this RVP as the transaction's *commit RVP*. The commit RVP's counter is initialized to the number of abortable pieces. When an abortable piece finishes its execution and determines that it can commit, it decrements the counter. However, if a piece must abort, it atomically sets the counter to -1. The final value of the counter is either 0 or negative. If the value is 0, the transaction can commit. If the counter's value is negative, the transaction must abort.

6.4.3 Piece ordering constraints

In order to guarantee serializable and recoverable execution, PWV must appropriately order pieces corresponding to *different* transactions. PWV must deal with write-read, write-write, and read-write conflicts between pieces. Among these classes of conflicts, only write-read and write-write conflicts can impact recoverability. Read-write conflicts cannot impact recoverability because the abort of a reader has no impact on a later writer.

If transaction T_1 is serialized before T_2, and one or more pairs of their constituent pieces conflict, then PWV imposes constraints on the order in which T_1 and T_2's pieces can execute. Assume that pieces P_1 and P_2 conflict (where P_1 and P_2 correspond to T_1 and T_2, respectively). PWV orders the execution of P_1 and P_2 based on **Constraints 1 and 2** below. **Constraint 1** captures ordering due to write-read and write-write conflicts, and is divided into two cases depending on whether the preceding writer can abort. **Constraint 2** captures ordering due to read-write conflicts.

- **Constraint 1.** There exists a write-read or write-write conflict between P_1 and

P_2.

a) $\mathbf{P_1}$ can abort because it precedes T_1's commit point. In this case, *all* of T_1's abortable pieces (including P_1) must execute before P_2.

b) $\mathbf{P_1}$ cannot abort because it follows T_1's commit point. In this case, P_1 must execute before P_2.

- **Constraint 2.** There exists a read-write conflict between P_1 and P_2. In this case, P_1 must execute before P_2.

The constraints above ensure that PWV produces only serializable and recoverable schedules. For serializability, PWV ensures that if T_1 is serialized before T_2, then P_1 is always executed before P_2. For recoverability, PWV ensures that transactions never observe dirty reads due to **Constraint 1a**.

6.4.4 Executing pieces

This section describes a multi-core optimized implementation of PWV which respects the ordering constraints from Section 6.4.3.

System model and assumptions

PWV divides the records in the database across a set of mutually exclusive partitions. Each partition processes pieces that read and write records in its partition. PWV can guarantee that a piece always writes records in a single partition by assigning each read or update statement its own piece. The techniques described in this section assume that PWV is deployed on a single multi-core server, such that a single CPU core is assigned a partition of the database.

Intuitively, PWV imposes two total orders; first, a total order on each transaction's pieces, second, a total order on transactions themselves. PWV processes transactions in batches, ordered as follows: Each transaction's pieces are ordered according to a topological sort of the decomposed transaction's DAG (Section 6.4.1). Pieces from different transactions are ordered according to transaction order in the input log; if transaction T_1 precedes transaction T_2 in the log, then *all* pieces of T_1 precede *all* pieces of T_2.

What if a piece spans partitions? Given a batch of totally ordered pieces (as described above), each core only processes pieces that read or write records on its partition. In certain cases, it may be impossible to deduce in advance which partition must execute a piece. Such an ambiguous piece is replicated and processed by every partition. Upon ascertaining the correct partition at runtime, irrelevant replicas immediately commit without executing any logic, while the relevant piece is executed as usual (as Section 6.4.4 will describe).

Partition local concurrency control

PWV must ensure that it executes pieces such that the constraints in Section 6.4.3 are satisfied. PWV ensures that the serialization order of transactions in a particular batch is *exactly the same* as the total order in which the transactions are received as input.

When a partition receives a batch of pieces, it first constructs a dependency graph whose edges represent conflicts among pieces within the partition. PWV constructs this dependency graph in its entirety (for a particular batch), before it executes the first piece in the batch. This partition-local dependency graph captures read-write, write-read, and write-write conflicts among pieces. PWV's dependency graphs are similar to those used in prior deterministic concurrency control algorithms [80, 82, 168]. These prior algorithms use dependency graphs in a *shared-everything* context, while PWV's dependency graphs are partition-local.

In order to construct a batch's dependency graph, PWV needs to determine piece-wise conflicts. This either requires determination of read and write sets as is done in other deterministic systems [80, 82, 157], or alternatively, a piece can conservatively request to access arbitrary ranges of records, such as a partition or an entire table (Section 6.4.5 discusses this issue in detail).

A piece can be in one of three states, **Unexecuted**, **Executed**, or **Completed**. Non-abortable pieces can either be in state **Unexecuted** or **Completed**. After executing, abortable pieces first transition to **Executed**, and eventually transition to **Completed** after their corresponding transactions' commit decisions are determined. All pieces are initially **Unexecuted**.

```
1  def check_ready(piece, tx):
2
3    for dep in piece.local_dependencies:
4      if dep.state != Completed
5        return False
6
7    if piece.rvp != 0
8      return False
9
10   if piece.non_abortable and tx.commit_rvp != 0:
11     return False
12
13   return True
```

Listing 6.1: Outline of conditions required to begin executing a piece.

Once a partition core constructs a batch's dependency graph, it progresses through the total order of pieces generated in Section 6.4.4 and performs three checks to see if it can immediately execute that piece. The first check ensures that a piece P is correctly ordered with respect to pieces from *other* transactions. The second and third checks ensure that P is correctly ordered with respect to pieces in its own transaction. Listing 6.1 outlines the three conditions in pseudocode.

- First, for every piece P' in the partition-local dependency graph which P depends on, the partition core checks whether P' is **Completed**. This step ensures that conflicting pieces execute according to the pre-determined total order of transactions (lines 3-5).

- Second, the partition core checks that P's RVP counter is zero. This step ensures that P's data dependencies have been satisfied (lines 7 and 8).

- Third, if P is not abortable then the partition core checks whether P's corresponding transaction has obtained a commit decision (by checking the corresponding commit RVP). This step ensures that non-abortable pieces only execute if their corresponding transactions commit. This check always holds if P is abortable (lines 10 and 11).

If all three of the above checks hold, there exist two cases depending on whether P is abortable. If P is abortable, then it is executed. If P is non-abortable and the third check determines that P's transaction committed, then P is executed. However, if the third check determines that P's transaction aborted, then P can be ignored and its state directly transitioned to **Completed**. If any of the above three checks does not hold, P is added to a list of pending pieces, and the core moves on to the next piece in the batch.[1]

Listing 6.2 shows the postprocessing that must happen after a piece executes. On executing piece P, its core decrements the count on each of P's children's RVPs (lines 5 and 6)). If P is non-abortable, its state transitions to **Completed** (lines 8 and 9). If P is abortable and can commit, the partition core decrements the corresponding commit RVP's counter and P's state transitions to **Executed** (lines 13 - 14). If P's commit decrements the RVP's counter to zero, then it means P's transaction has committed, and the partition core proceeds to transition every abortable piece's state to **Completed** (including P's, lines 16-18)). If P is the first piece to abort, the partition core undoes the writes of the pieces that committed before P (even if the pieces reside on remote partitions), and mark their state as well as P's as **Completed** (lines 21-26). Note that remote undo is safe because the abortable pieces are still in state **Executed**, and as a consequence, later conflicting pieces from different transactions are blocked because of the first check above. Finally, if P is not the first piece to abort, then it simply undoes its own effects (lines 28-29).

1. As an optimization, our implementation performs the third check even if P is abortable. If its transaction aborted, P does not need to execute, whether or not it is abortable, and can directly transition to **Completed**, even if the first two checks do not hold.

```
def execute(piece, tx):

  piece.run()

  for child_rvp in piece.child_rvps:
    child_rvp.atomic_decrement()

  if piece.non_abortable:
    piece.state = Completed
  else:

    if piece.committed:
      tx.commit_rvp.atomic_decrement()
      tx.state = Executed

      if tx.commit_rvp == 0:
        for abortable in tx.abortable_pieces:
          abortable.state = Completed

    else:
      if atomic_xchg(tx.commit_rvp, -1) != -1:
        piece.state = Executed
        for abortable in tx.abortable_pieces:
          if abortable.state == Executed:
            abortable.undo()
            abortable.state = Completed
      else:
        piece.undo()
        piece.state = Completed
```

Listing 6.2: Outline of piece postprocessing.

The above partition-local mechanisms guarantee that the ordering constraints of Section 6.4.3 hold. The first step, which checks whether conflicting pieces have finished executing, ensures **Constraint 1b** and **Constraint 2** hold, while the first step and the commit protocol above together ensure that **Constraint 1a** holds.

PWV's constraints enable implementations that exploit both intra-transaction parallelism and early write visibility. Our implementation uses the a priori total ordering of transactions and their pieces to correctly order pieces across different transactions. Prior decomposition algorithms cannot exploit intra-transaction parallelism. This limitation is fundamental because these algorithms cannot make any a priori ordering guarantees across multiple pieces corresponding to a pair of conflicting transactions [136,151,164,170,173].

We initially experimented with a design based on partition-local lock managers as in DORA [140]. However, for the workloads evaluated in this chapter, we found that partition-local lock managers came with significant overhead. Most problematic that a single partition core would need to execute lock-manager logic between individual transaction executions. By constructing a dependency graph of transactions up-front, PWV avoids polluting CPU caches by mixing instructions and data associated with concurrency control and transaction logic [144].

6.4.5 Discussion

Deferred constraint checks

Database constraints on records are usually checked as soon as update statements that could invalidate the constraints are evaluated. However, certain types of constraints (such as those involving the values of two or more records) are rendered temporarily inconsistent if evaluated after a single update statement. Transactions typically fix these constraints with later updates. Database systems therefore allow certain constraint checks to be deferred to the end of transactions' execution [21].

If applications use deferred integrity constraints, then PWV must place the commit point of any transaction that triggers the deferred constraint check at the end of its execution. Importantly, this *does not* mean that PWV cannot be used by such applications, only that certain transactions' writes cannot be made visible early. Furthermore, because PWV uses a modular decomposition procedure, deferred constraint checks in one transaction do not affect the commit point of other transactions, even if they conflict. It should be noted that practitioners have proposed that applications should avoid deferred constraint checks when possible; for example, by grouping updates together using multiple assignment operators [12,17].

Comparison to prior deterministic systems

Deterministic database systems determine the serialization order of transactions prior to their execution, and execute transactions in an order that is equivalent to the pre-determined serialization order. Depending on the mechanisms they use to ensure equivalence to this pre-determined serialization order, deterministic systems can be divided into two categories: those that sequentially execute transactions according to the serialization order, and those that re-order non-conflicting transactions in a manner that preserves the serialization order.

Systems in the first category rely on partition parallelism to scale transaction execution. However, they can leave partitions severely under-utilized if even a small fraction of transactions span multiple partitions [144,159]. Systems in the second category identify transactions that conflict, and ensure that conflicting transactions are executed according to the pre-determined serialization order. These systems impose no restrictions on transactions that do not conflict. Therefore, non-conflicting transactions can be executed in parallel. These systems determine conflicting pairs of transactions by requiring that each transaction's read and write sets be deduced prior to its execution. If they cannot be deduced by inspection, additional work must be done prior to transaction execution. One example of a protocol to perform this additional work is OLLP, which performs this preparatory work via a reconnaissance step [146,157].

Similar to this second category of deterministic systems, PWV needs to determine conflict information prior to transaction execution. In particular, PWV leverages piece-wise conflict information in order to execute pieces out of order on each partition. However, the early write visibility discipline of PWV enables an alternative mechanism to the extra preprocessing work required by OLLP.

If their read and write sets are unknown, pieces can conservatively specify ranges of records that they may need to access. These ranges can be arbitrarily imprecise; for instance, a piece may request exclusive access to an entire table even though it only updates a handful of records in the table. Such coarse-grained conflict specification can severely limit concurrency in conventional systems, but this limitation is alleviated because PWV only blocks conflicting *pieces*, not entire transactions.

Indeed, even when read and write sets are known in advance, we have found that coarse-grained conflict specification can improve the performance of PWV — especially under lower contention workloads. Coarse-grained conflict information allows PWV to trade off logical concurrency between conflicting pieces for reduced concurrency control overhead, in the spirit of hierarchical locking [88]. This coarse grained conflict specification is similar to the partition-level conflict specification in partition parallel deterministic systems (the first category of deterministic systems above). PWV does not suffer from the deleterious effects of coarse-grained conflict specification because pieces are isolated for shorter *durations* than entire transactions.

While the lock-acqusistion overhead vs. reduced concurrency performance tradeoffs between coarse-grained locks and fine-grained locks are well-known in traditional systems (and hierarchical schemes exist to allow both to exist within the same system), we have found that PWV pushes this tradeoff in favor of course-grained locks, significantly expanding the space in which they provide a performance benefit. We explore this tradeoff in our experimental evaluation (Section 6.5.3).

6.4.6 Proof of optimality

This section proves that if transactions' read and write sets can be deduced a priori, then PWV's piece ordering constraints are necessary and sufficient to guarantee serializability (SR) and avoid cascaded aborts (ACA) in the absence of failures. We term these two properties together as SR ACA. Our proof pertains to transaction histories permitted by PWV's piece ordering constraints (Section 6.4.3), not our specific implementation of these constraints (Section 6.4.4). The implication of this proof is that, when transactions' read and write sets are known a priori, it is impossible for an SR ACA protocol to extract more concurrency from a workload than an ideal implementation of PWV.

Our proof asserts that, under the assumptions above, PWV permits *all* valid serializable transaction histories (unlike, for instance, two-phase locking, which cannot permit certain valid serializable histories [58, 78]). Section 6.4.3 showed that PWV's constraints on pieces are sufficient to guarantee serializability. We prove that these constraints are also *necessary* if transactions' read and write sets are known a priori.

We define a transaction's constituent read, write, and commit/abort statements as its *operations*. We denote read, write, and commit operations of transaction T_i as $r_i[x]$, $w_i[x]$, and c_i, respectively (where reads and writes occur on record x). We denote operation o_1 preceding operation o_2 as $o_1 < o_2$.

If a transactions' read and write sets are known a priori, PWV can assign each read and write operation to a single piece. The proof sketch below therefore describes constraints on individual read and write operations, not pieces. Given two conflicting transactions, T_i and T_j, such that T_i is serialized before T_j, PWV imposes the following constraints on the order in which their operations can execute:

- **Case 1a.** If $w_i[x]$ conflicts with $o_j[x]$, and $w_i[x] < c_i$ (where o_j is either r_j or w_j), then PWV ensures that $c_i < o_j[x]$. No SR ACA concurrency control protocol can produce the order $w_i[x] < o_j[x] < c_i$ because T_i may abort, and $o_j[x]$ would have observed an uncommitted write ($w_i[x]$) which could have rolled back prior to c_i. Furthermore, no serializable concurrency control protocol can produce $o_j[x] < w_i[x]$ because doing so would violate the assumption that T_i is serialized before T_j.

- **Case 1b.** If $w_i[x]$ conflicts with $o_j[x]$, and $c_i < w_i[x]$, then PWV ensures that $w_i[x] < o_j[x]$. No SR ACA concurrency control protocol can produce $o_j[x] < w_i[x]$ because doing so would violate the assumption that T_i is serialized before T_j.

- **Case 2.** If $r_i[x]$ conflicts with $w_j[x]$, then PWV ensures that $r_i[x] < w_j[x]$. No SR ACA concurrency control protocol can produce $w_j[x] < r_i[x]$ because it violates the assumption that T_i is serialized before T_j.

6.4.7 Corner cases

PWV classifies each of a transaction's pieces as abortable based on the transaction's logic (specifically, the location of explicit abort statements) and constraints on database values, such as integrity constraints. Above, we classified all other types of aborts as system-induced, which are eliminated by deterministic database systems. However, there exist corner cases that fall in between these two categories, in which even deterministic systems would abort the transaction, but nonetheless are not caused by abort statements in transaction logic or integrity constraints. Two examples of such corner cases are integer overflows and infinite loops. One naïve corner case handling mechanism is to consider all pieces that modify integers or involve loops (and so on for all corner cases) as abortable.

Obviously, this naïve solution would lead to a large number of pieces being marked as abortable, precluding PWV's ability to make many writes visible early. A better approach is to engineer a solution to deal with each corner case individually. For loops, it is possible to use static analysis to detect loops that will definitely terminate. Only for the case where the static analysis fails to guarantee that a loop will terminate do the pieces corresponding to the loop's logic need to be marked as abortable. For integer overflows, the system could simply allow integers to overflow (as is the default setting in many modern database systems). Alternatively, the size of the integer (or entire column) can be dynamically increased in order to accommodate the overflow.

Although dealing with corner cases on a case-by-case basis using software engineering techniques, such as static analysis and exception handling, is likely the optimal solution, our current implementation uses a more general approach. Our implementation optimizes for the common case where corner cases do not occur, and suffers from reduced performance when they do. In particular, upon encountering a corner case, such as an integer overflow or infinite loop, our implementation treats this as a full system failure, trashes the current database state, reloads state from the most recent checkpoint, expunges the problematic transaction from the log, and replays the log forward from the checkpoint without the problematic transaction. Clearly, optimizations of this algorithm are possible. For example, instead of trashing the entire database state and replaying the entire log from a checkpoint, one could use piece-wise conflict information to selectively re-execute only those pieces which may have read the aborted transaction's writes.

By selectively aborting problematic transactions, the above dynamic error handling mechanism prevents these expunged transactions from affecting stable database state. However, it does not prevent the writes performed by such aborted transactions from being visible to the application running over the database (for instance, via simple read queries). Our current implementation delays returning results of any data to the application until any transactions that contributed to these results have finished execution. We implemented this via an epoch-based external visibility mechanism, similar to the mechanism used in Silo [159], where read results are returned to the user at the end of each batch of transactions.

6.5 Evaluation

This section evaluates PWV against three serializable protocols – locking, transaction chopping, and optimistic concurrency control (OCC) – and a read committed protocol.

Locking. This implementation is based on two-phase locking. The implementation acquires locks in lexicographic order to eliminate deadlocks [144]. To avoid the overhead of maintaining a separate lock table, logical locks are implemented as MCS reader-writer latches co-located with records [128,163].

Transaction chopping. This implementation is based on Wang et al.'s IC3 protocol [164]. IC3 uses a serializable protocol to schedule transactions' constituent pieces, and dynamically enforces causal dependencies across conflicting pieces. Our chopping implementation uses locking to guarantee serializable execution of pieces.

OCC. This implementation is based on Silo [159]. OCC validates transactions using decentralized timestamps, and avoids writing shared-memory for records that are only read.

Read committed. We implemented read committed isolation (RC) by modifying the OCC algorithm above. Our RC implementation provides PL-2 isolation [41], which imposes two constraints on transactions' reads and writes. First, transactions can only read committed values of records. Second, if two transactions perform conflicting writes, then their writes must be consistently ordered [41,42]. Our RC implementation buffers transactions' writes until commit time. A writer will therefore only interact with a reader at commit time. RC uses a record latch – Silo's per-record TID word [159] – to ensure that reads observe only committed values of records. A writer acquires this latch while copying a record's updated value from its local buffer. Readers spin on the latch until it is free. RC deals with write-write conflicts using MCS latches, which, as in our locking implementation, are co-located with records [127,163]. At commit time, a transaction acquires its write latches in lexicographic order, and then copies updated records' values from its local write buffers. Our RC implementation provides PL-2 isolation, which provides more concurrency than the PL-2L isolation provided by most real-world implementations of RC [41,55,88,117].

We conduct our experimental evaluation on a single 40-core machine, consisting of four 10-core Intel E7-8850 processors and 128GB of memory. Our operating system is Linux 3.19.0. All experiments are performed in main-memory, so secondary storage is not utilized for our experiments. Every implementation explicitly pins long running threads to CPU cores.

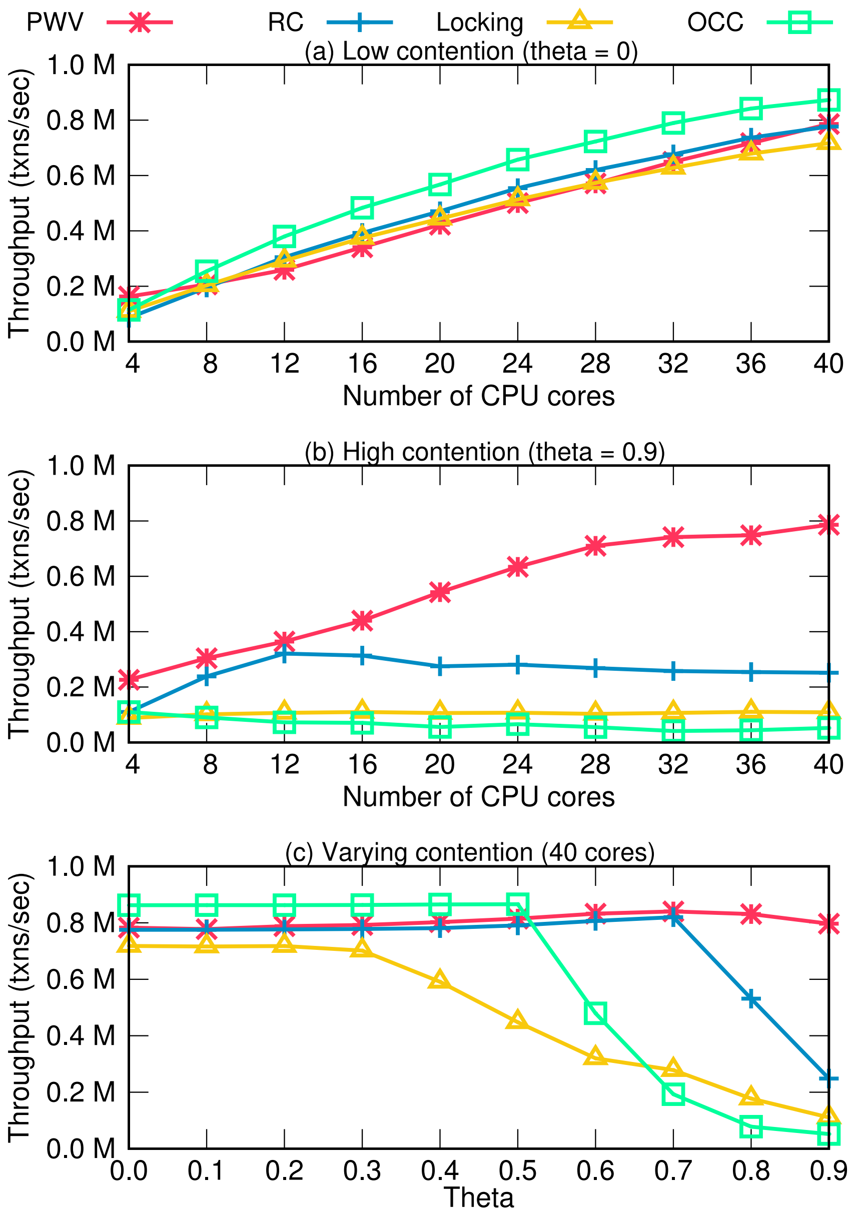


Figure 6.3: Effect of contention on throughput. (a) Low contention. Concurrency control protocols exhibit similar behavior because conflicts among transactions are rare. Furthermore, there is no advantage to running transactions under weak isolation in the absence of contention. (b) High contention. Locking and OCC's throughput drops due to conflicts. RC's throughput drops due to commit ordering of writes. PWV decouples individual writes and thus scales near-linearly. (c) Varying contention. Locking and OCC's throughput decrease at medium levels of contention. RC's throughput decreases at high levels of contention. PWV's throughput remains stable.

6.5.1 Effect of contention

In this experiment, we use the Yahoo! Cloud Serving Benchmark to understand PWV's basic performance characteristics [69]. The database consists of a single table of 1,000,000 records. Each record is 1,000 bytes in size. The workload in this section consists of a single type of transaction; an update transaction that performs 20 read-modify-write (RMW) operations. The records updated by each transaction are chosen from a zipfian distribution. We vary contention by varying the zipfian parameter, *theta* [90]. PWV's batch size is set to 10,000 transactions. We partition data using a random hash function, as a consequence most PWV transactions span more than 10 partitions.

The experiments in this section assume that transactions do not contain any abort statements, that is, they are guaranteed to commit before they begin executing. As a consequence, in PWV, there is no delay from the time that an individual update is performed, and the time it is made visible to other transactions. We perform three sets of experiments, one under low contention, one under high contention, and one under varying contention (Figure 6.3). Transaction chopping does not provide any benefit to our locking implementation in this experiment because it decomposes transactions based on table-level accesses. We thus omit transaction chopping from this set of experiments.

Figure 6.3a shows the results of the low contention experiment. We measure the throughput of each implementation while varying the number of available CPU cores. The zipfian parameter, theta[2], is set to 0. Figure 6.3a indicates that each system scales similarly under low contention because conflicts among transactions are rare.

Figure 6.3a also highlights the relationship between strong and weak isolation levels under low contention — there is no significant performance advantage to running transactions under weak scalability of the conventional serializable algorithms, OCC and Locking, are nearly equivalent to that of Read Committed (RC).

Figure 6.3b shows the results of the same experiment under high contention. In this case, the records updated by each transaction are chosen from a zipfian distribution with theta set to 0.9. Locking and OCC's throughput drops significantly as compared to the low contention experiment. This is because in the high contention experiment, the likelihood that a pair of transactions conflicts is much higher. Un-

2. Theta can take values between 0 and 1. Larger values of theta correspond to higher contention.

der the Locking concurrency control algorithm, transactions will acquire conflicting locks on the same records; Locking will therefore block any transaction that requests a lock on an already-locked record. Under OCC, if several transactions concurrently update the same record(s), then only one of those transactions will be allowed to commit. The rest of the transactions are aborted and later re-tried. In both cases, contention significantly constrains the number of transactions that can execute concurrently. Furthermore, under OCC, the work performed by a transaction is wasted if it aborts, while Locking avoids wasting work by blocking transactions. As a consequence, Locking outperforms OCC under high contention.

Figure 6.3b also shows that RC's throughput significantly decreases under contention. Although RC does not impose any order among conflicting reads and writes, writes across transactions must still be consistently ordered. Accordingly, our RC implementation acquires write locks on records at commit time, before transactions copy updated values from their buffers into the database (Section 6.5). The decrease in RC's throughput under high contention occurs due to transactions acquiring the same write locks at commit time. Importantly, these locks are acquired at commit time, and held for much shorter duration than locks acquired by serializable locking. This explains why RC can attain a much higher peak throughput than locking under high contention. Finally, since transactions perform only updates, locking-based PL-2L implementations that hold long-duration write locks on records would perform the same as serializable locking. [3].

Finally, we find that PWV's throughput trend is completely different from the other concurrency control algorithms. The locking and OCC lines remain nearly flat, while RC peaks at 12 cores and plateaus thereafter. In contrast, PWV's throughput increases with core count without plateauing. Since transactions contain no abort statements, PWV can make an update visible as soon as it completes, without waiting for the corresponding transaction to finish executing in its entirety. This decoupling of a single transaction's constituent writes is the reason that contention does not adversely affect PWV's throughput. At 40 cores, PWV outperforms OCC, Locking, and RC by 15x, 7x, and 3x, respectively.

To better understand the behavior of each algorithm, we measured each algorithm's throughput while varying contention. Figure 6.3c shows the results of the experiment (contention increases with increasing theta). Locking and OCC's

3. Note that since traditional Locking-based RC implementations [55, 88] acquire long-duration write would result in the same performance as Locking for this experiment.

throughputs decrease at medium-low levels of contention (theta range of 0.3 to 0.6). RC and PWV's throughput remain very similar from low to medium contention. RC's throughput decreases when theta increases from 0.7 to 0.8 (medium to medium-high contention). PWV's throughput remains nearly constant despite variations in contention.

These experiments show that PWV is highly robust to contention in the ideal scenario that transactions never experience logic-induced aborts. Under high contention, PWV can outperform conventional serializable protocols by more than an order-of-magnitude. PWV even outperforms our highly-optimized *non-serializable* read committed implementation, indicating that PWV can provide applications that fit this ideal with fast serializable isolation.

6.5.2 Effect of commit point

In this experiment, we limit PWV's ability to make individual writes visible as soon as they complete. We augment transactions' logic with explicit abort statements. By varying the position of transactions' abort statements with respect to its update statements, we can control which writes can be made visible immediately.

We term the point at which a transaction contains an abort statement its *commit position*. The value of a transaction's commit position corresponds to the number of write operations that precede it. We measure the effect of changing a transaction's commit position under low and high contention. As in Section 6.5.1, we use transactions that perform 20 read-modify-writes. The parameters of the low and high contention experiments are the same as Section 6.5.1.

Figure 6.4a shows the result of the low contention experiment. Locking, OCC, and RC's throughputs do not change while varying a transaction's commit position. The fact that a transaction contains an explicit abort statement has no effect on these protocols. PWV's throughput also remains nearly constant while varying transactions' commit position. However, small variations occur because each core needs to perform slightly more work to execute transactions; PWV must execute the RVP-based commit protocol (Section 6.4.2), and writing pieces that follow a transaction's commit point must wait for the commit protocol to complete.

Figure 6.4b shows the result of the high contention experiment. As before, locking, OCC, and RC's throughputs do not change with commit position. In contrast, PWV's throughput decreases significantly with increasing commit position. This is because PWV cannot make a transaction's writes visible until its commit po-

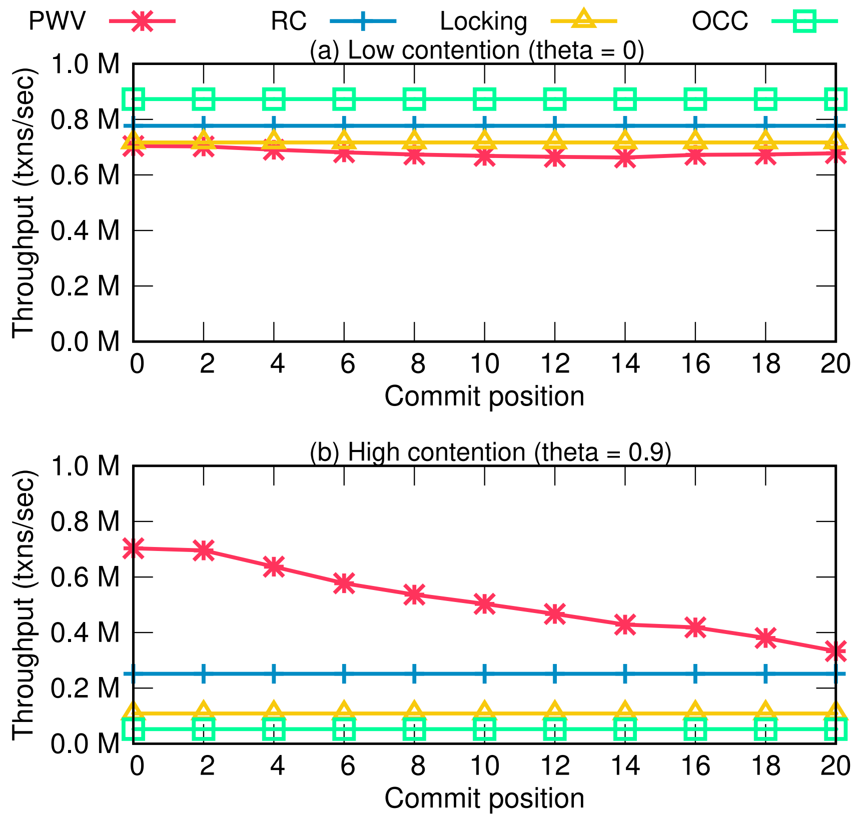


Figure 6.4: Effect of commit point on throughput. Locking, OCC, and RC's throughput is invariant to abort position because they make transactions' writes visible at the end of their execution. (a) Low contention. PWV experiences a slight drop in throughput because of the extra cycles needed to determine a transaction's commit decision (Section 6.4.2). (b) High contention. PWV's throughput decreases with increasing abort position because of increasing write visibility delay. However, PWV still outperforms the other baselines due to its use of intra-transaction parallelism, which minimizes write visibility delay. (Relative to the other baselines.)

sition. Under high contention, this delayed visibility hurts throughput. Unlike the other algorithms, PWV employs intra-transaction parallelism by executing a single transaction's updates in parallel on multiple cores. Intra-transaction parallelism minimizes the execution time of an individual transaction, which reduces write visibility delay. Thus, PWV's throughput remains significantly higher than locking and OCC's, even when every algorithm makes writes visible at the end of each transaction (the right-most point of Figure 6.4b).

6.5.3 Reducing concurrency control overhead

We now examine the performance implications of PWV's coarse-grained conflict isolation mechanisms, whereby transactions can request access to a set of keys that is guaranteed to be a superset of the keys they actually access. We show that coarse-grained isolation can significantly improve PWV's performance under low contention, while preserving its advantages under high contention.

We evaluate the throughput of each system under a workload consisting of an equal mix of TPC-C NewOrder and Payment transactions [40]. We use two versions of PWV for this benchmark: standard PWV, and PWV-coarse. In order to access the District, Customer, NewOrder, OrderLine, Order, and History tables, both PWV and PWV-coarse specify read and write requests at the granularity of (warehouse-id, district-id) foreign-key pairs. We partition these tables by (warehouse-id, district-id) pairs. Pieces that request access to a table via the same (warehouse--id, district-id) foreign-key pair are always processed by the same partition, but records with the same (warehouse-id, district-id) foreign-key from different tables, say Customer and NewOrder, can reside on different partitions. PWV and PWV-coarse differ in their conflict specification mechanisms for Stock records. PWV isolates pieces at the granularity of Stock record primary keys, while PWV-coarse isolates conflicting accesses to the Stock table at the granularity of warehouses. In PWV-coarse, each piece effectively requests exclusive access to the entire set of stock records within a single warehouse. Thus, if two pieces update stock records within the same warehouse, then PWV-coarse determines that they conflict, even if their read and write sets do not overlap.

We also measure the impact of coarse-grained isolation on conventional recoverability mechanisms by implementing a version of coarse-grained locking. Like PWV-coarse, coarse-grained locking protects all the stock records within a warehouse with a single lock. Furthermore, we include two versions of our implementation of IC3's transaction chopping protocol [164]. A standard version of IC3, which isolates pieces at the granularity of reads and writes, and a version that exploits commutativity. We refer to these algorithms as chopping and chopping-comm, respectively. Note that PWV does not make any commutativity assumptions.

We first show the results of a low contention experiment in which we simultaneously vary the number of database cores and warehouses (the number of warehouses is equal to number of cores). The non-PWV algorithms affinitize a core to a particular warehouse; requests that originate at a particular warehouse are always processed by the same core. This experiment therefore represents the *best case* scenario for these systems; locality is maximized and conflicts are minimized because transactions on a particular origin warehouse are always processed by the same core [144, 159].

Figure 6.5a shows the results of the experiment. Every algorithm's throughput scales with increasing core count. However, there exist significant differences in absolute throughput achieved by each algorithm. First, both chopping and chopping-comm are outperformed by locking because of the extra overhead they impose on tracking dependencies between pieces at runtime. Chopping-comm outperforms chopping despite the lack of conflicts because it maintains less chopping-related meta-data corresponding to updates by commuting pieces. Locking outperforms locking-coarse because locking-coarse induces unnecessary conflicts. Since approximately 10% of NewOrder transactions update stock records from remote warehouses, these transactions will block due to spurious stock update conflicts. This reduction in concurrency outweighs any potential benefit in reduced concurrency control overhead.

In contrast, the opposite effect is observed for PWV-coarse, where the reduced overhead greatly outweighs the reduction in concurrency. PWV-coarse *pipelines* the execution of transactions to minimize the impact of coarse-grained isolation on blocking. If two transactions conflict, PWV-coarse only imposes an order between the transactions' conflicting pieces, not entire transactions.

Figure 6.5b shows the result of a high contention experiment in which we fix the number of warehouses to 1, and vary the number of database cores. In non-PWV algorithms, the entire database is shared across all cores of the system. Due to increasing contention, OCC, locking, and locking-coarse's throughput remain mostly stagnant with increasing core count. Surprisingly, chopping is unable to outperform any of these systems. There are two reasons for this. First, if two pieces corresponding to a pair of transactions conflict, then later non-conflicting pieces corresponding to the same pair of transactions are constrained [164]. Second, dynamically tracking dependencies across pieces imposes overhead at runtime, which worsens under increased contention.

In contrast, PWV and PWV-coarse are able to outperform both RC and chopping-comm, even though they provide serializable isolation and make no assumptions about commutativity of pieces. PWV and PWV-coarse outperform chopping-comm because they impose no constraints on the execution of non-conflicting pieces,

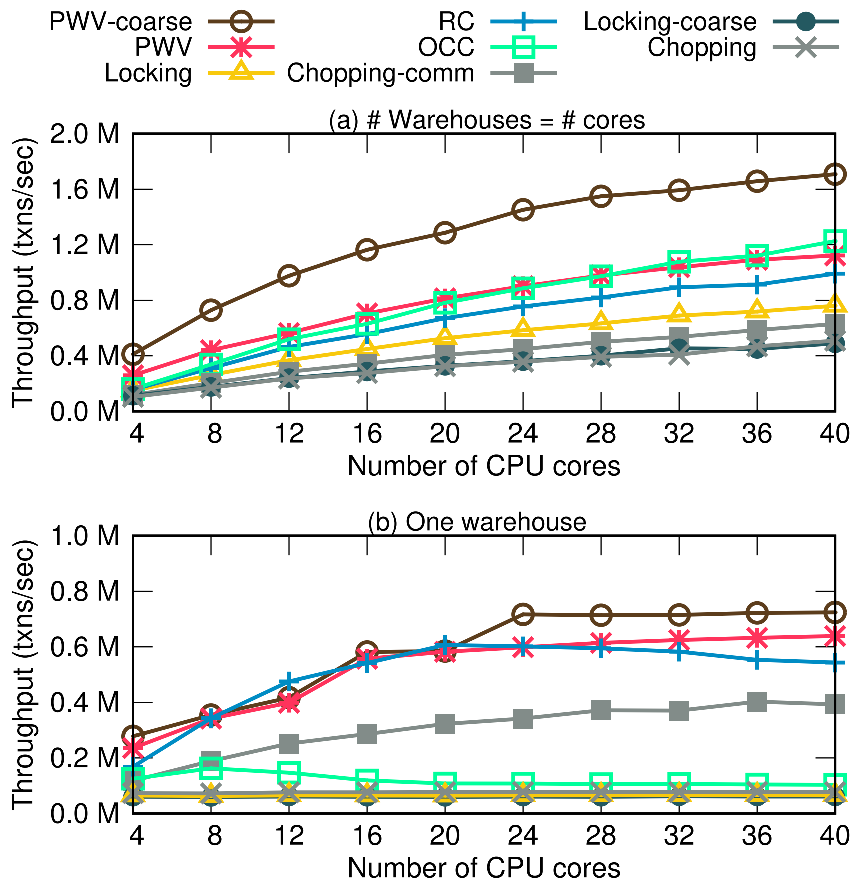


Figure 6.5: TPC-C NewOrder and Payment throughput.

while chopping-comm must constrain non-conflicting pieces (although these constraints are minimized due to commutativity). The difference between RC and PWV is smaller than prior experiments because RC must acquire fewer commit time write locks and consequently holds write locks on the most contended records (Warehouse and District records) for short durations.

	PWV-coarse	Locking	OCC	RC	Chopping-comm
40 warehouses	2ms	8.2ms	3.4ms	4.5ms	42ms
1 warehouse	7.3ms	73ms	534ms	10ms	136ms

Table 6.1: 95^{th}%ile latency for batches of 5000 NewOrder and Payment transactions under maximum throughput (40 cores).

Finally, Table 6.1 shows the 95^{th} percentile latency of processing batches of 5,000 TPC-C transactions. Table 6.1 shows that PWV-coarse's multi-core scalability does not come at the expense of latency; PWV-coarse's 95^{th} percentile latency latency is lower than that of every other protocol under both high and low contention.

6.6 Conclusions

This chapter identifies write visibility delay as an important inhibitor of database concurrency and introduces early write visibility, a recoverability mechanism that enables writes to become visible as soon as a transaction executes any statements that could cause it to abort. To enable early write visibility, we designed PWV, a new concurrency control protocol that leverages database determinism to prevent arbitrary transaction aborts, and found that PWV can significantly outperform modern serializable and non-serializable concurrency control protocols.

Chapter 7

Application survey

Deterministic transaction execution requires conflicts between transactions to parallelize their execution. Deterministic execution processes the input log of transactions, and uses conflict information to decide how transactions need to be constrained. In most work on deterministic transaction processing this conflict information is assumed to be transactions' read and write sets. This chapter examines a set of Ruby on Rails [27] applications to determine how often transactions' read and write sets can be determined prior to their execution in the real world.

Rails applications employ an object relational mapper framework (ORM) called ActiveRecord, which can be called directly via Ruby. We chose Rails applications for two reasons. First, we were interested in analyzing methods that perform database operations when users interact with applications, and Rails made it easy to identify these methods. Rails applications are based on the Model View Controller (MVC) architecture, in which user facing code calls controller methods, which in turn interact with the database via ActiveRecord. Rails applications must clearly define and label controller methods; this clear demarcation of controller methods made it easy for us to identify which methods to analyze. Second, ActiveRecord exposes a simple, narrow interface with which applications can interact with the database. The semantics of the methods in the interface are well documented, and did not require any analysis. In contrast, analyzing applications built over alternative frameworks, such as Java Database Connectivity (JDBC), would have required combining two special purpose analyzers; one to handle SQL statements and another to handle Java.

7.1 Ruby on Rails background

Rails applications employ an object relational mapper framework (ORM), ActiveRecord, to interact with database backends. ActiveRecord provides methods to query, write, and navigate records in the database. Rails applications use the Model View Controller (MVC) architecture; user facing code calls controller methods, which interact with the database via ActiveRecord methods defined on models. Our analysis conservatively assumes transactional intent with every controller method; we treat every database access performed in the context of a controller method call (including its callees) as part of a single transaction context. For performance, Rails applications typically do not employ a transaction per controller call, instead each database access is executed as a separate transaction. Incidentally, prior work has shown that loosening transactional guarantees across database accesses can introduce severe bugs [165]. Furthermore, assuming that all database interactions must occur within the scope of a single transaction assumes worse case application design patterns because it assumes coarse-grained transactional intent, even when it is acceptable to run multiple fine-grained transactions in the context of a single controller method.

Given that every controller method should ideally execute a single transaction, our static analyzer checks for ActiveRecord methods called by the application. The ActiveRecord query interface provides a single method, `find`, which take a list primary keys as input, and returns the records associated with the specified keys. We treat all other ActiveRecord query methods as non-primary key lookups (e.g., `where`, `find-by`, `first`, `last`, `all`, and so forth). Moreover, applications can navigate relationships between records via associations, which can specify foreign key relationships between records. We treat all queries via associations as non-primary key lookups. Finally, our analysis considers writing methods, which modify the database's state via inserts, updates, and deletes (e.g., `save`, `destroy`, `update`, `find-or-create-by`, and so forth).

We assume that applications only use ActiveRecord's explicitly provided mechanisms to identify records whose primary keys are unknown at transaction submission time; e.g., associations, selection via `where` or `find-by`, and so forth. An application could violate this assumption by managing non-primary key lookups itself. Consider the example of an online forum application in which every Post is associated with a corresponding Thread. There exists a one-to-many relationship between Posts and Threads; each Post is associated with a unique Thread. Given a Post's primary key, an application could navigate to the associated Thread via a `thread-id` field in each Post record. Given this schema, a controller method could: (1) Take a Post's primary key as input; (2) Lookup the Post record via `find`; (3) Use the Post record to obtain the corresponding Thread's primary key via the `thread-id` field; (4) Lookup the Thread record via `find`; (5) Update the Thread record. This hypothetical controller method's read and write set are not fixed at submission time; the Thread record, which is read and updated, is only identified after reading the Post record.

However, Rails makes application-specific management of lookups based on foreign key relationships and other relationships between records cumbersome because applications must cede control of primary key generation to ActiveRecord. ActiveRecord automatically generates records' primary keys in an application agnostic fashion. Based on its internal knowledge of primary keys, ActiveRecord transparently and automatically generates schemas based on relationships between records of two or more tables; e.g., foreign key relationships. In the above online forum application, ActiveRecord would transparently embed a `thread-id` field in the Post schema, and automatically generate navigation queries underneath an object-oriented interface. The application can lookup the Post record via `find`, and then obtain its corresponding Thread by navigating via `post.thread`.

In summary, we assume that applications only use ActiveRecord's mechanisms to identify records whose primary keys are unknown at transaction submission time. Thus, explicitly detecting these Active mechanisms in each controller method's call graph can serve as a proxy to check whether its corresponding transaction's read and write sets are unknown at submission time. However, this also means that the resulting analysis can produce false positives; e.g., if a controller method specifies a list of primary keys to retrieve records via the `where` method, then it is misclassified as a non-primary key lookup.

7.2 Static analysis algorithm

This section describes a lightweight static analyzer we built to analyze Rails applications. The analyzer flags controller methods whose interactions with the database have the following two properties; first, the method performs at least one non-primary key access; second, the method performs at least one modification. The first condition conservatively flags a controller method as not knowing its read and write sets prior to invocation because of the assumption outlined in Section 7.1; if a controller method does not know its read and write sets when invoked, then it performs at least one non-primary key database access. When combined with the second condition, it implies that the flagged methods' corresponding transactions might need to be speculatively executed. We ignore read-only transactions because recent work has shown that read-only transactions can efficiently be served via a lightweight checkpointing mechanism that maintains transactionally consistency [159].

Starting from a Rails application, we use the RubyParser library [28] to construct abstract syntax trees (ASTs) of the methods defined in the project. Our core analysis takes these ASTs as input. The analysis performs two passes over the application's methods.

The first pass is an intra-procedural analysis; it walks through each method's AST and generates a list of callees in program order. The first pass was simple to implement, we had to recursively traverse each expression in every method's AST and insert every method call into a list.

```
1  def second_phase(callee_list):
2    output = []
3    for callee in callee_list:
4      if callee.is_active_record_method():
5        output.append(callee)
6      else:
7        resolve_methods = callee.resolve()
8        for method in resolve_methods:
9          output += second_phase(summaries[method])
10   return output
```

Listing 7.1: Outline of the second phase algorithm.

The second pass is an inter-procedural analysis; it outputs a list of ActiveRecord calls for each controller method. Listing 7.1 outlines the second pass algorithm in pseudocode. For each controller method, the analyzer examines each callee obtained in the first phase (Line 3); if the callee is an ActiveRecord method it is added to the output list (Lines 4 and 5), otherwise, the analyzer recursively generates the list of ActiveRecord calls for the callee and concatenates the callee's list to original method's the output list (Lines 7-9). The output is thus an ordered list of ActiveRecord calls made in a controller method's call graph.

Unlike the first pass, the second pass was difficult scale to large applications. Some of our applications were large, on the order of tens of thousands of lines of code. The main scaling challenge was ambiguity in resolving callees and the blowup in space it caused. Ruby is a dynamically typed language, making it difficult to resolve callees statically. If the analyzer cannot resolve a callee to a single method via scoping rules, then it assumes all matching methods might be called. We initially constructed multiple summaries per method, each corresponding to a unique matching method. However, this led to an unmanageable blowup in space for several of our applications because the state is forked for each matching method at a particular callee. With multiple such unmatched callees, the forked state quickly blows up; e.g., if a method has three callees, each of which resolves to M, N, and O different methods, then the analysis would generate up to $M * N * O$ lists for the method. To avoid this space blowup, the analyzer instead concatenates the lists obtained from a callee's resolved methods (Line 9 in Listing 7.1). This leads to a loss in precision: although the methods' ActiveRecord calls may not individually involve both a non-primary key lookup and a write, their concatenation might. E.g., if method A calls `where` (classified as a non-primary key lookup) and method B calls `save` (classified as a write), their concatenated summary will contain calls to both `where` and `save`.

Finally, we outline assumptions we made while resolving callees in the second pass.

Undefined methods. If there exists no definition corresponding to a callee, the analyzer assumes that the callee does not call any ActiveRecord methods.

Resolving callees. Applications often define methods with the same name at different scopes; e.g., two classes might each define a `serialize` method. To resolve a method's callees, we inspect the method's full scope and resolve the callee to the longest prefix that matches the scope. If more than one callee matches the same prefix, we conservatively assume that all of the methods may be called.

ActiveRecord. We resolve callees to ActiveRecord methods if their names match, regardless of scope. We break recursive analysis when a callee resolves to an ActiveRecord method (Lines 4 and 5 in List 7.1).

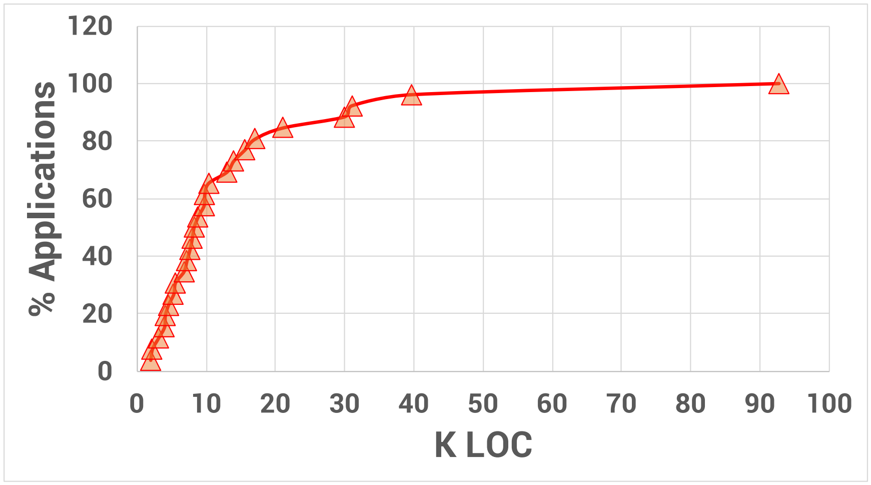


Figure 7.1: CDF of surveyed Rails applications' lines of code counts.

Anonymous methods and super class methods. We ignore anonymous methods defined within the scope of another method. We also ignore calls to a super class-method from the corresponding sub-class method.

7.3 Results

We conducted both, an automated analysis and a manual analysis of Rails applications. The automated analysis simply runs our analyzer over a set of 26 Rails applications. The results of this analysis are coarse-grained statistics which determine the fraction of controller methods that both perform at least one non-primary-key lookup and at least one write. In our manual analysis, we selected six applications (each with fewer than 10 flagged methods) so that we could inspect every flagged method in the application.

7.3.1 Automated analysis

We use the static analyzer to obtain the set of controller methods. Among these methods, we analyze those which perform database accesses (termed DB access methods), and flag them if they perform both a non-primary key access and a write access. Figure 7.1 shows this raw data. Figure 7.1 plots the CDF of applications' lines of code counts. Approximately 40% of the surveyed applications are over 10,000 lines of code (LOC). The set of surveyed applications are spread over a wide range of sizes and thus complexity. In the analysis that follows, we measure the

Application	Lines of Ruby	Controller methods	DB access methods	Flagged methods
amahi-platform [1]	6764	119	64	14
boxroom [3]	1952	70	26	5
Brevidy [4]	9685	107	39	6
calagator [5]	9772	67	34	1
citizenry [6]	7960	53	19	1
diaspora [7]	39704	274	148	55
enki [8]	3961	55	12	0
forem [9]	4534	96	30	9
fulcrum [10]	3096	49	9	0
jobsworth [13]	21134	329	162	98
linuxfr.org [14]	7634	342	153	25
lobsters [15]	8225	163	64	9
locomotivecms [16]	17078	161	19	7
opencongress [18]	29987	654	388	66
opengovernment [19]	8845	84	39	3
openstreetmaps [20]	31032	386	202	67
piggybak [22]	2201	15	5	0
publify_core [25]	12979	179	91	20
rubygems.org [29]	13965	123	25	6
rucksack [30]	5304	156	114	105
shoppe [31]	4058	111	52	4
skyline [32]	10344	146	99	92
snorby [33]	7197	133	89	62
spot.us [34]	92723	273	112	33
tracks [35]	15573	264	118	22
wallgig [37]	5538	140	56	6

Table 7.1: Summary of surveyed Rails applications.

percentage of DB access methods that are flagged by the static analyzer.

Figure 7.2 shows a scatter plot of flagged method ratios and applications' lines of code counts. Application size is often used as a proxy for complexity; this plot shows the distribution of flagged methods across a wide range of application complexity. In most cases, less than 40% of DB access methods are flagged. In four applications, more than 40% of DB access methods are flagged; skyline (93%, 10,344 LOC), rucksack (92%, 5,304 LOC), snorby (70%, 7,197 LOC), and jobsworth (60%, 21,134 LOC). Among the largest applications, those larger than ~30,000 lines of code, fewer than 40% of DB access methods are flagged; spot.us (29%, 92,723 LOC), diaspora (37%, 39,704 LOC), openstreetmaps (33%, 31,032 LOC), and opencongress (17%, 29,987 LOC).

Another measure of application complexity is the number of controller methods it defines. Figure 7.3 shows a scatter plot of flagged DB access methods and the number of controller methods in each application. Seven applications defined more than 200 controller methods, and one had more than ~40% of its DB ac-

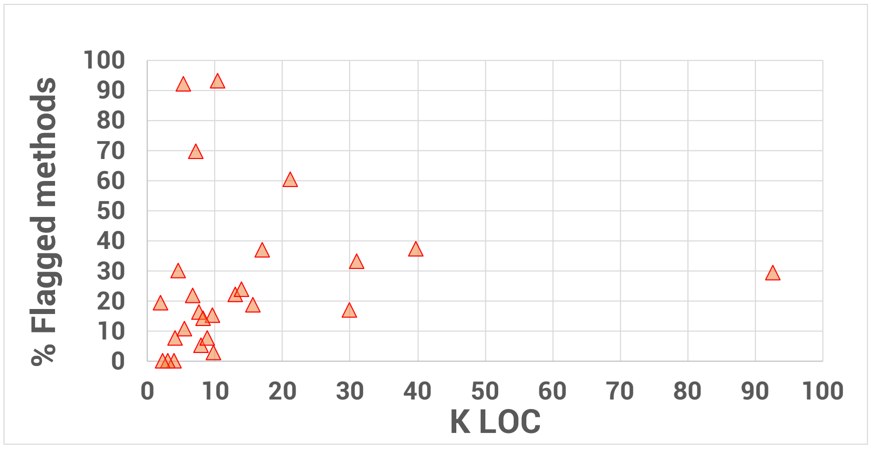


Figure 7.2: Scatter plot of percentage of flagged DB access methods and applications' lines of code counts.

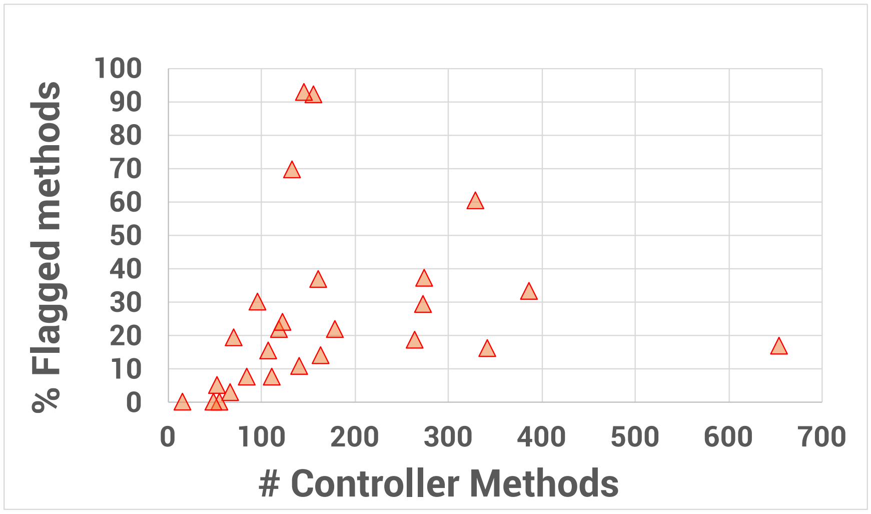


Figure 7.3: Scatter plot of percentage of flagged DB access methods and number of controller methods.

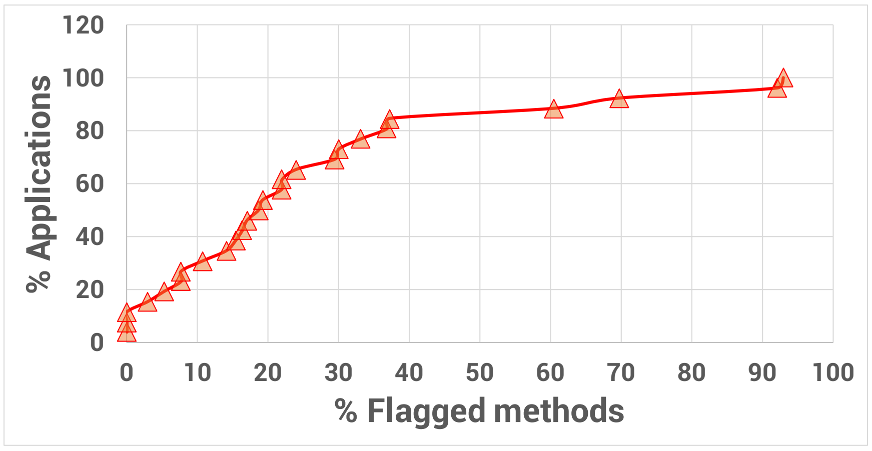


Figure 7.4: CDF of flagged DB access methods.

cess methods flagged (jobsworth, 60%, 329 controller methods). Among the four largest, those with more than 300 controller methods, three had fewer than 35% of their writing methods flagged; opencongress (12%, 654 controller methods), openstreetmaps (34%, 386 controller methods), linuxfr.org (16%, 342 controller methods). The fourth application was jobsworth, with 60% of its DB access methods flagged. Among the applications that had more than 70% of their writing methods flagged in the previous LOC based scatter plot, all defined less than ~150 controller methods.

Finally, Figure 7.4 shows the CDF of flagged DB access methods in the set of analyzed applications. 60% of applications had less than 22% of their DB access methods flagged, while ~80% had less than ~37% of their DB access methods flagged.

7.3.2 Manual analysis

Given that methods flagged by our static analyzer may sometimes be misclassified as performing a non-primary key access, we manually analyzed six applications which had a small number of methods flagged (less than 10). We analyzed all the flagged methods in these applications. Via manual analysis, we determine if flagged methods actually performed a non-primary key access, and also look to find common types of non-primary key lookups or common patterns which lead to misclassification.

Boxroom

Boxroom [3] is an tool for online file sharing. It has a hierarchical structure, in which files are located in folders, and allows users to set permissions on files and folders.

`AdminsController:create` **[Misclassified]** This method was incorrectly flagged. The static analyzer matched a field access of an object with an association (a non-primary key access).

`FoldersController:create` **[Foreign key]** This method looks up a permissions table by folder_id, which is a foreign key associated with each permission.

`GroupsController:create` **[Foreign key]** This method looks up a permissions table by group_id, which is a foreign key associated with each permission.

`UsersController:create` **[Misclassified]** This method was incorrectly flagged. The static analyzer matched a field access of an object with an association.

`PermissionsController:update_multiple` **[Foreign key, Secondary index]** This method updates the permissions associated with a folder; it looks up permissions by folder_id, which is a foreign key access. It also recursively updates the permissions associated with a folder's children. There is a one-to-many relationship from children to parents, and each child has a foreign-key relationship with its parent.

Brevidy

Brevidy is a social media application [4]. Its controller methods thus reflect social-media-centric logic, such as profile updates, blocking users, profile setting updates, and so forth.

`ProfileController:update` **[Unique key]** This method updates a user's profile, and looks up the user's profile by user_id, which is a unique key. It is flagged because the profile is looked up by user_id, which is not a primary key, but still a unique key.

`UsersController:block` **[Unique key]** This method looks up a `Blocking` table, whose entries specify a blocking relationship between two users, where the first user, the "blocking user", blocks a second user, the "blocked user". The blocking user_id and blocked user_id together uniquely identify records in the `Blocking` table. Before inserting a new record into the table, the method checks whether the record exists by specifying a blocking user_id and blocked user_id pair.

`UsersController:update_notifications` **[Unique key]** This method updates a record in the Settings table, which corresponds to a particular user's settings. The setting record is looked up via a user-id. The user-id is a unique key associated with each setting record.

`VideosController:encoder_callback` **[Foreign key]** This method is called when a user uploads a video. The server preprocesses the video, and updates two tables, a Videos and a VideoGraphs table. Video records have a foreign-key relationship with records in the VideoGraph table. The method is provided a video-id, and looks up and updates the corresponding VideoGraph record by navigating an association from Videos to their corresponding VideoGraph.

`VideosController:successful_upload` **[Foreign key]** This method updates a Video's VideoGraph record. It is given a video-id as input, and navigates to the VideoGraph via an association.

`VideosController:upload_error` **[Foreign key]** Same as the two methods above.

Calagator

Calagator is an open source community calendaring platform, primarily used as an aggregator for events in Portland, OR [5].

`Calagator:DuplicateChecking:ControllerActions:squash_many_duplicates` **[Misclassified]** This method was incorrectly flagged. It looks up records using ActiveRecord's `where` method with a list of primary keys. Its lookups are thus by primary key. The method was flagged by our static analyzer because the `where` method is assumed to perform non-primary key lookups.

Open Government

OpenGovernment is an application for aggregating and presenting open government data [19].

`Admin:TaggingsController:create` **[Misclassified]** This method looks up records via a where clause with a list of primary keys, and was thus misclassified as performing a non-primary key lookup.

`Admin:TaggingsController:destroy` **[Misclassified]** Same as above.

`Admin:IssuesController:destroy` **[Select all]** This method deletes a record from a Tag table by specifying its primary key. As its last action, the method retrieves all records from the Tag table via `Tag.all`. This query does not impact the result of the delete. The query could therefore be executed in a separate transaction. Every other method in this controller also retrieves all records from the Tag table and renders them in the browser, indicating that the table is likely small. Accesses to the table could therefore simply take a lock on the entire table.

Rubygems.org

Rubygems is the Ruby community's gem hosting service [29].

`EmailConfirmationsController:create` **[Secondary index]** This method looks up users by their email addresses. The look up is on a non-primary key. However, we expect the corresponding index to not change very often (since users' email addresses do not change very often).

`EmailConfirmationsController:update` **[Foreign key]** This method looks up users by a unique confirmation token, the confirmation token record references a record in the user via a foreign key.

`EmailConfirmationsController:unconfirmed` **[Misclassified]** This method updates a user's profile. It was flagged because one of its callees looks up the last user in the User table via `User.last` to send the user an email. On manual inspection, we found that this was a false positive because the corresponding method is only invoked under preview or test. The actual method called at runtime looks up users by their id, but the analyzer could not resolve the function correctly statically.

`ProfilesController:update` **[Misclassifed]** This method updates a user's profile, and has the same callee as the controller method above. It thus also erroneously flagged.

`UsersController:create` **[Misclassified]** This method creates a new user record, and sends a confirmation email to users. The email is sent via the same callee the two methods above. The method is thus also erroneously flagged.

`Api:V1:RubygemsController:create` **[Secondary index]** This method creates a new Ruby gem record. On its critical path, it looks up the Rubygem table via a name and version number. This is a secondary index look up.

Shoppe

Shoppe is a Rails-based e-commerce platform, that allows users to build a catalog-based online store [31].

The static analyzer flagged four controller methods in Shoppe:

- `UsersController:destroy` **[Misclassified]**
- `OrdersController:accept` **[Misclassified]**
- `OrdersController:reject` **[Misclassified]**
- `OrdersController:ship` **[Misclassified]**

All these methods were flagged because they looked up users by calling User.First in ActiveRecord. On manual inspection, we found that this path was only taken when the applications runs in demo mode. In non-demo-mode, users are always looked up by their ids.

7.3.3 Discussion

We manually analyzed 6 applications and 25 different controller methods. Of these 25 methods, 12 were mistakenly flagged, 6 navigated to records via foreign key relationships, 3 looked up records via unique keys, 2 looked up records via secondary indexes, 1 used both foreign keys and secondary indexes, and 1 retrieved all records from a particular table (Open Government's `Admin:IssuesController:destroy` method).

Seven of the twelve misclassifications were due to demo calls to ActiveRecord's `first` or `last` methods. Three methods were misclassified because they used the `where` method to lookup primary keys. Finally, two methods were misclassified because the analyzer resolved a field access to an association (this can occur when a class attribute has the same name as an association). We expect that several methods in the automatic analysis were misclassified for similar reasons.

Nine different methods looked up records via a foreign key relationship or a secondary index. These records were either found by directly looking up a field in a record whose primary key was specified or by looking up a secondary index via specified arguments. Importantly, no controller logic needed to execute to determine the records to look up and the records could be retrieved via a single extra read from the database (to the corresponding secondary index or the child record of a foreign key).

Finally, we also found that applications would sometimes lookup records via application defined unique keys. This is likely because applications must cede control of primary key generation to ActiveRecord.

7.4 Implications for deterministic database systems

To date, past work on deterministic database systems has assumed that transactions must be speculatively executed in their entirety to determine their read and write sets [146, 157]. A deterministic system processes a transaction whose read and write sets are unknown in the following three steps. First, the transaction is speculatively executed, and its read and write sets are obtained. Second, the transaction's logic, its input parameters, and the speculative read and write sets are logged. Third, the transaction is processed after it is stably logged. As part of processing a transaction, the system deterministically validates its speculatively obtained read and write sets.

Speculative execution of entire transactions is necessary when read and write sets can only be determined via a sequence of data-dependent reads from the database. This chapter's analysis shows applications do not actually ever perform data-dependent reads to determine their read and write sets. This has two important implications for deterministic database systems:

Lightweight speculative execution. Our manual analysis showed that if a transaction's read and write sets unavailable prior to execution, then they can be determined by reading a secondary index, reading a child record of a foreign key relationship, or a unique key look up. We never once encountered any need for a data-dependent sequence of multiple database look ups. To determine such transactions' read and write sets, deterministic database systems could simply speculatively read the appropriate indices and tables, and validate the reads at transaction execution time. These speculative reads would be much lighterweight than full blown speculative execution.

Early commit decisions. Since transaction's read and write sets can be determined via simple one-shot database reads (i.e., without performing a sequence of data-dependent reads), they can also be validated via equally simple one-shot reads. A transaction's read and write sets can thus quickly be validated as soon as it is ready to execute, which enables aggressive early write visibility (Chapter 6) if the transaction never triggers logic-induced aborts.

In conclusion, this chapter's analysis of real-world applications paints a rosy picture for deterministic transaction execution; the assumptions made in this body of work hold in the fairly broad set of applications we analyzed. Indeed, the assumptions in prior work have probably been overly pessimistic; the generality of speculative execution of entire transactions was never necessary because a transaction's read and write sets could easily be determined via one-shot database reads. Finally, our analysis shows that most of the time, even the above lightweight speculative execution is unnecessary. The majority of writing controller methods could determine their read and write sets a priori, even under the pessimistic approximations made by our static analyzer.

Chapter 8

Related work

8.1 Deterministic transaction processing

In 1997, Whitney et al. proposed a deterministic transaction processing mechanism for handling high throughput workloads whose datasets fit entirely in main-memory [168]. Whitney et al.'s transaction processing mechanism used the state machine approach [149] to maintain consistent database replicas. Transactions would be logged in a total order prior to their execution, after which single-threaded database replicas would execute the log of transactions in order, guaranteeing that their state would never diverge. The motivation for their approach was the "Wall Street" setting of high frequency trading, and hence transaction, volume. Single-threaded execution eliminated the need for complex concurrency control and recovery protocols (such as two-phase locking [78] and ARIES [133], respectively), and as a consequence, performed significantly better than conventional database systems. Furthermore, their use of the state machine approach to replication, as opposed to the then widespread two-phase commit for replication, obviated the need for any complex distributed protocols.

In 2007, Stonebraker et al. proposed the H-store transaction processing database system, which extended Whitney et al.'s ideas by *sharding* the database into independent partitions, and using the state machine approach within partitions [155]. H-store was motivated by research (from the same group at MIT) showing that when datasets fit in memory, the then state-of-the-art relational database systems spent the vast majority of their cycles (over 95%) on overhead related to transaction processing [94]. This included time spent on memory/buffer pool management, concurrency control, synchronization, and recovery. H-store's response was to employ single-threaded in-memory execution, as was originally proposed by Whitney et al., to avoid these overheads. At the time, the need for parallelism in transaction processing gained importance due to newly available multi-core hardware and the ability to shard data across servers in a distributed system. H-store therefore proposed parallelizing transaction execution by sharding state into independed single-threaded state machines, and executing cross-shard distributed transactions by blocking the involved shards for the duration of the transaction. While H-store's single-threaded execution would significantly outperform interleaving transactions via concurrency control, distributed cross-shard transactions would leave shards severely under-utilized, even at a small fraction of mostly single-shard transactions.

In their recent research on the Calvin database system, Thomson et al. found that deterministic execution could allow partitioned database systems to reduce the cost of commit protocols on the critical path of transactions [156–158]. In comparison to the two-phase commit protocol [116], which requires two round-trips of communication between participating servers and two synchronous writes to durable storage to commit a transaction, Calvin could commit a transaction with just half a round trip of coordination in the worst case, and often required no coordination whatsoever. To avoid commit protocol related overheads it could rely on database shards to deterministically arrive at the same outcome for a particular transaction, regardless of failure. Two-phase commit's synchronous writes and coordination are used to guard against partitions non-deterministically reneging on a prior decision to commit or abort a transaction due to failures. In addition to decreasing the cost of commit protocols, Thomson et al. proposed exploiting knowledge of conflicts between transactions to avoid sequential execution within a partition. In particular, by only constraining the execution of transactions that actually conflict, Calvin could avoid sequentially executing transactions within a partition. The significantly reduced cost of commit coordination and its ability to avoid sequential execution within partitions enabled Calvin to execute distributed transactions with significantly less overhead than H-store.

A recently proposed class of distributed transaction processing systems relies on the use of a *shared log abstraction* to implement optimistic transactions [54,56,59, 166]. Database or application state resides in the form of in-memory objects backed by a durable, fault-tolerant shared log. In effect, an object exists in two forms: an ordered sequence of transaction updates stored durably in the shared log; and any number of views, which are full or partial copies of the data in its original form.

The shared log acts as a source of strong consistency, durability, failure atomicity, and transactional isolaton.

Comparison with our work. The common theme across this prior work on deterministic transaction execution is that they employ deterministic execution to circumvent overhead in transaction processing. Whitney et al. and H-store circumvent overhead in conventional disk-oriented database architectures by sequentially executing transactions in memory. Calvin circumvents overhead in distributed transaction processing by ensuring that partitons always deterministically arrive at the same state regardless of failures. In addition to circumventing overhead of conventional database systems on modern hardware, we show that deterministic execution can lead to implementations of serializability that are competitive with weak isolation levels, such as read committed and snapshot isolation. Furthermore, prior work does not explore the implications of deterministic transaction execution on large-scale multi-core hardware. Indeed, the proposed mechanisms and architectures are counterproductive to multi-core scalability. For instance, both Whitney et al.'s main-memory system and Calvin perform transaction scheduling within a global critical section, which fundamentally limits scalability on multi-core hardware.

Transaction processing systems based on distributed shared logs use the log as a convenient abstraction. Instead, this thesis is focussed on concurrency control mechanisms *after* transactions after been logged, that is, they are effectively a form of log replay, and can be used by shared log systems to efficiently materialize state.

8.2 Transaction processing on modern hardware

With the democratization of access to servers, both virtual and physical, with large main-memories and core counts has come the need for a rethinking of database architecture and mechanisms on modern hardware. We now discuss related work on database systems explicitly designed for this new hardware.

In their work on Shore-MT, Johnson et al. were among the first to recognize and address the scalability limitations of conventional database architectures on modern hardware [109]. The Shore-MT project addressed scalability limitations in database concurrency control protocols, logging mechanisms, indexes, and high level architecture [107, 108, 110, 140, 141]. Johnson et al. identified latch contention on high level intention locks as a scalability bottleneck in multi-core database systems [107], and proposed a technique to amortize the cost of contended latch acquisitions across a batch of transactions by passing hot locks from transaction to transaction without requiring calls to the lock manager. Pandis et al. proposed a shared-nothing transaction processing architecture to avoid contention on locking meta-data (DORA) [140], and indexes (PLP) [141]. Unlike conventional database architectures in which a single thread performs a transaction's logic, their work proposes that a transaction is collectively processed by the partitions on which it must execute. A transaction's logic is broken into smaller sub-transactions such that each sub-transaction is restricted to a single partition.

Since conventional database systems were designed for environments with low to non-existent hardware parallelism, they employed design patterns that were unsuited to large-scale multi-core hardware. For instance, in several database systems, the lock manager is protected by a single mutex. Jung et al. proposed addressing this limitation in MySQL by synchronizing threads' accesses to the lock table's individual buckets rather than the table in its entirety [112]. Horikawa proposed addressing the same limitation in Postgres's concurrency control metadata [103]. The Silo [159] and Hekaton [117] database systems addressed similar bottlenecks in optimistic concurrency control protocols. As originally proposed by Kung and Robinson [115], optimistic concurrency control must validate transactions in a global critical section, which limits its scalability on modern hardware. Larson et al. mitigated this bottleneck by reducing the global coordination to an atomic increment of a single global counter. Tu et al. removed the bottleneck altogether via the observation that single version database systems do not require monotonically increasing transaction identifiers to ensure serializability. Both Silo and Hekaton addressed logging bottlenecks in database systems via redo-only logging of transactions.

Comparison with our work. While these approaches to transaction processing address limitations of conventional systems via better mechanisms, architectures, and algorithms, our work focusses on a clean-slate approach to multi-core transaction processing. This prior work is focussed on the design of avoiding scalability bottlenecks in non-deterministic database systems. Our research focusses on scalable deterministic execution, while also addressing the gap between strong and weak isolation levels.

8.3 Multi-version concurrency control

In principle, database systems can employ multi-versioning in order to decouple reads and writes to the same record. Database systems employ a timestamping mechanism to identify versions of records; transactions are assigned unique timestamps, and the versions of records created by their writes are marked with this timestamp. A version of a particular record can therefore be identified by its timestamp.

There generally exist two timestamping mechanisms used in database systems; those that assign transactions monotonically increasing timestamps, and those that assign unique, but non-monotonic timestamps. The monotonicity of timestamps (or lack thereof) has a significant impact on the design of multi-version concurrency control protocols. A transaction's timestamp determines the versions of records it is permitted to read, and determines the version number its writes are assigned. With a monotonic timestamp generator, transactions can be assigned two timestamps; (1) a begin timestamp, assigned when it enters the systems and which determines a database snapshot visible to the transaction, and (2) an end timestamp, assigned at the end of its execution, which determines the timestamps assigned to its writes. The begin timestamp effectively fixes the snapshot of the database visible to the transaction, because writes by later transactions will always be assigned greater timestamps. With non-monotonic timestamps, this property no longer holds true; a write by a later transaction might be assigned a *lower* timestamp, and might therefore be visible to earlier transactions. Multi-version concurrency control protocols that employ non-monotonic timestamps therefore require reads to protect themselves from being invalidated by later writes via some read-specific meta-data that must be maintained on every read.

Multi-version protocols based on monotonic timestamps therefore need to maintain less meta-data than those based on non-monotonic timestamps. On the other hand, monotonic timestamp generators are typically implemented via global counters on modern hardware, which requires global synchronization on every transaction and can limit scalability on multi-core hardware.

Multi-version concurrency control was first introduced by Bernstein et al. [60] and Reed [143] in the context of distributed database systems. These protocols were referred to as *timestamp ordering*, and were based on non-monotonic timestamps so that transactions could be assigned timestamps in a decentralized fashion; a transaction could be assigned a timestamp from any one among a collection of servers, which could append a local counter and a unique server identifier to construct timestamps. Since these protocols used non-monotonic timestamps, each record maintained a read timestamp (in addition to its write timestamp) corresponding to the timestamp of the latest reading transaction. Read timestamps would prevent writes with earlier timestamps from invalidating reads by simply aborting the writing transactions. One of the primary disadvantages of read meta-data maintenace in these early protocols was that reads required disk writes to maintain their meta-data [91]. Consequently, real-world database systems, such as PostgreSQL, SQL Server, and Oracle have employed multi-versioning protocols based on monotonic timestamps.

Recent research has adapted multi-version concurrency control for multi-core main-memory databases. Most notably, Microsoft's Hekaton project [72,117] adopted an optimistic protocol based on monotonic timestamps. This protocol was further optimized by Neumann et al. [138] in the HyPer database system [113]. While these modern protocols imposed low overhead on transactions, the monotonicity requirement on timestamps fundamentally limits their scalability [80, 159]. Consequently, there has been a resurgence of interest in protocols that can use non-monotonic timestamps, such as Deuteronomy [120] and Cicada [122]. Unlike their predecessors from the 1980s, these protocols can maintain read meta-data in memory resident data-structures due to the abundance of main-memory, and asynchronous logging protocols that do not perform writes to durable storage on the critical path of transactions [72, 110, 117, 159].

Comparison with our work. Deterministic execution circumvents the above trade-off between timestamp monotonicity and protocol and meta-data complexity. Transactions are assigned monotonically increasing timestamps in a scalable fashion because they are effectively derived outside of their execution. Furthermore, we guarantee that reads never impede writes by implicitly resolving read write conflicts in the order that the corresponding reading and writing transactions occur in the log. In contrast, to the best of our knowledge, non-deterministic multi-version concurrency control protocols cannot decouple reads from writes without one impeding the other. Protocols based on timestamp ordering abort writers in favor of readers [60, 120, 122, 143]. Those based on locking block readers and writes [117], and those based on optimistic conurrency control abort readers in favor of writers [117, 138].

8.4 Write visibility

One of the key themes in transaction processing research has been to increase concurrency between transactions. Increased concurrency directly translates to more parallelism at runtime because more transactions can execute in parallel. Research that aims to increase transaction concurrency can be classified into two categories; approaches that *weaken isolation* between transactions and approaches that *decompose transactions* into smaller sub-transactions. We now discuss each of these two lines of research in more detail.

8.4.1 Weakening isolation

One approach to increase transaction concurrency is to simply run them at weaker levels of isolation. However, this exposes them to anomalies, such as non-repeatable reads and lost updates [55], which can cause subtle application-level bugs. As a consequence, there has been significant research interest in exploiting application semantics to employ weak isolation levels while still preserving application-level correctness.

One of the earliest examples of application-specific use of weak isolation were Escrow transactions, first proposed by Gawlick and Kinkade [87], and O'Neil [139]. Escrow transactions exploit commutativity of operations on contended records to employ short-duration write-locks, thereby making writes to these records immediately visible to other transactions, as opposed to the delayed write visibility mechanisms employed by protocols such as two-phase locking and optimistic concurrency control. Garcia-Molina and Salem proposed the Saga abstraction, which consists of a series of sub-transactions, which are sequentially executed by the database. A sub-transaction's writes are made visible as soon as its finishes executing, allowing Sagas more leeway to interleave as compared to running the sub-transactions within the context of a database transaction. Sagas exploit application semantics to implement *logical undo* to cancel out the effects of sub-transactions in case the Saga must abort. Alonso et al. proposed logical inverse undo operations to achieve a similar effect [49].

More recently, exploiting application semantics for weak isolation has seen a resurgence of interest to avoid coordination on multi-core hardware and distributed systems. Bailis et al. propose an application dependent correctness criterion *I-confluence*, that determines whether a coordination free execution of transactions will preserve application invariants [53]. Conway et al. propose using monotonicity analysis to eliminate coordination in distributed applications [68]. Similar to Escrow transactions, Doppel exploits commutative operations on hot records to replicate such records on a single multi-core server, and allows concurrent commutative updates to each replica [137].

Comparison with our work. Each of these prior techniques exploits application semantics to enable either a new recoverability or isolation mechanism, or both. In contrast, we makes no assumptions about application semantics, while still guaranteeing recoverability and serializability. The novelty of our work lies in its use of deterministic execution guarantees to devise new recoverability and scheduling algorithms, such as early write visibility and piece-wise scheduling (Chapter 6), to reduce the duration of conflict-induced blocking.

8.4.2 Transaction decomposition

Transaction chopping is the most well-known mechanism for serializable transaction decomposition [151]. As proposed by Shasha et al. in 1995, transaction chopping is a static analysis mechanism for decomposing transactions at the application-level, and makes no assumptions about the underlying database system. As a rule of thumb, transaction chopping groups transactions statements that could lead to an abort in a piece, and furthermore does not permit conflicts between transactions on more than one piece. This second implication follows directly from the fact that transaction chopping makes no assumptions about the underlying database system. If two transactions conflict on two different pieces, then the database system could inconsistently order them, leading to a serializability violation.

Transaction chopping has recently seen a resurgence of interest in modern research on transaction processing. Recent systems that employ transaction chopping extend it in two important ways, addressing some of its limitations in the process. First, some systems, such as Lynx [173] and Salt [169], have advocated exploiting application semantics to determine whether any two pieces will conflict, and use this conflict information to inform the static analysis mechanism that is used to decompose transactions. The resulting decompositions can be finer grained than those produced by static analysis based purely on database accesses. Others have advocated transaction-chopping-aware database scheduling, in which the database is no longer treated as a black-box, and actively aids in the scheduling of transaction pieces. This latter extension to transaction chopping alleviates the limitation that transactions can only conflict on a single piece by dynamically ordering pieces in the database. Examples of this extension to transaction chopping include IC3 [164] and Runtime pipelining [170].

Comparison with our work. Unlike transaction chopping and its recent variants, our work on PWV guarantees serializable execution by using the input total order of transactions to schedule conflicting pieces, rather than static analysis. Furthermore, transaction chopping does not address the underlying issue of recoverability of transactions, and consequently only permits application aborts in one piece [151], or handles application aborts via cascading rollbacks [164,170].

Chapter 9

Concluding Remarks

This thesis makes the argument that the non-deterministic transaction processing principles that are the foundation of database systems can severely limit performance on modern hardware. Non-deterministic execution was necessary in the era of uniprocessors, paucity of main memory, and open ended interactive interfaces. But today, non-deterministic execution is fraught with limitations; it imposes overhead due to its unnecessary complexity, it limits scalability due to its inability to avoid contentious synchronization on multi-core hardware, and it limits the performance of serializable consistency as compared to weak isolation levels.

The key insight in deterministic transaction processing is that the above limitations can be addressed by assuming a total order of transactions as input (achieved by serializing transactions into a log prior to their execution), and employing transaction processing mechanisms that guarantee equivalence to this total order. This enables transaction processing mechanisms that are simple, fast, and provide guarantees that are impossible using non-deterministic execution. In particular, deterministic systems can execute transactions lazily, exploit multi-versioning to aggressively parallelize update transactions with strong guarantees, and reduce the gap between serializability and weak isolation levels.

In addition to these high level outcomes, this research proposes a novel scheduling and execution architecture based on dependency graphs. The scheduling and execution components of a database system are typically heavily intertwined; e.g., the execution component must make calls to a database system's lock manager as it processes transactions. This back and forth interaction between database components can limit performance because it destroys cache locality, even if transactions are aggressively interleaved. For instance, our initial implementation of the piece-wise visibility protocol based on core-local lock managers was untenable due to its poor cache locality. Dependency graphs eliminate all interaction between scheduling and execution components; the scheduling component can produce a dependency graph that specifies the order in which transactions must execute, while the execution component treats the graph as an immutable data structure and simply crawls it to find and execute transactions.

Finally, our application survey validates assumptions made by a long line of work on deterministic transaction processing. In particular, we found that when transactions' read and write sets are unknown prior to execution, they are easily deducible via speculative one-shot reads from the database.

We conclude on a philosophical note.

We are accustomed to the logic that more choice is a good thing. More choice means that our systems, and not just computer systems, have more opportunities to optimize for a specific metric. A good example is economics, where its taken for granted that more choice in the market is a good thing because it drives down prices.

This thesis shows that sometimes the opposite can be true; *less* choice can lead to more efficient systems. Deterministic transaction execution is all about less choice. While conventional database systems are free to choose any serial order in which to execute transactions, deterministic execution guarantees equivalence to one and only one serial order. This lack of choice enables transaction processing mechanisms that are faster and provide stronger guarantees than those with unbounded choice. Sometimes less is more.

Bibliography

[1] Amahi platform. https://github.com/amahi/platform.

[2] Bitcoin bank flexcoin closes after hack attack. https://www.theguardian.com/technology/2014/mar/04/bitcoin-bank-flexcoin-closes-after-hack-attack.

[3] Boxroom. https://github.com/mischa78/boxroom.

[4] Brevidy. https://github.com/iwasrobbed/Brevidy.

[5] Calagator. https://github.com/calagator/calagator.

[6] Citizenry. https://github.com/reidab/citizenry.git.

[7] Diaspora. https://github.com/diaspora/diaspora.

[8] Enki. https://github.com/xaviershay/enki/.

[9] Forem. https://github.com/rubysherpas/forem.

[10] Fulcrum. https://github.com/fulcrum-agile/fulcrum.

[11] The GNU C library (glibc). https://www.gnu.org/software/libc/.

[12] How to write correct sql and know it: A relational approach to sql. https://goo.gl/WDpTDU.

[13] Jobsworth. https://github.com/ari/jobsworth.

[14] Linuxfr.org. https://github.com/linuxfrorg/linuxfr.org.

[15] Lobsters. https://github.com/lobsters/lobsters.

[16] Locomotive cms. https://github.com/locomotivecms/engine.

[17] The multiple assignment operator and deferred constraints. https://goo.gl/84aGMR.

[18] Open congress. https://github.com/opencongress/opencongress.git.

[19] Open government. https://github.com/opengovernment/opengovernment.

[20] Open street maps. https://github.com/openstreetmap/openstreetmap-website.

[21] Oracle database online documentation 10g release 2 (10.2). https://goo.gl/g9oFBs.

[22] Piggybak. https://github.com/piggybak/piggybak.

[23] Poloniex loses 12.3% of its bitcoins in latest bitcoin exchange hack. https://www.coindesk.com/poloniex-loses-12-3-bitcoins-latest-bitcoin-exchange-hack/.

[24] Possible typo/bug in the michael and scott queue white paper psuedo-code. https://tinyurl.com/zx7dqsb.

[25] Publify. https://github.com/publify/publify_core.

[26] Reader/writer locks and their (lack of) applicability to fine-grained synchronization. https://tinyurl.com/gv7zfjt.

[27] Ruby on rails. https://rubyonrails.org/.

[28] Ruby parser. https://github.com/whitequark/parser.

[29] Rubygems. https://github.com/rubygems/rubygems.

[30] Rucksack. https://github.com/jamesu/rucksack.git.

[31] Shoppe. https://github.com/tryshoppe/shoppe.git.

[32] Skyline. https://github.com/DigitPaint/skyline.git.

[33] Snorby. https://github.com/Snorby/snorby.git.

[34] Spot.us. https://github.com/Snorby/snorby.gi://github.com/spot-us/spot-us.

[35] Tracks. https://github.com/TracksApp/tracks.

[36] Use-after-free bug in maged m. michael and michael l. scott's non-blocking concurrent queue algorithm. https://tinyurl.com/h4jlutf.

[37] Wallgig. https://github.com/jianyuan/wallgig.

[38] X1 instances for ec2 — ready for your memory-intensive workloads. https://tinyurl.com/hc7bgzz/.

[39] Xv6, a simple Unix-like teaching operating system. https://pdos.csail.mit.edu/6.828/2011/xv6.html.

[40] TPC Council. TPC Benchmark C revision 5.11. 2010.

[41] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, 1999.

[42] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *ICDE*, 2000.

[43] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1), 1987.

[44] D. Agrawal and A. El Abbadi. Locks with constrained sharing. In *PODS*, 1990.

[45] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *TODS*, 12(4), 1987.

[46] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[47] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques*. Addison Wesley, 1986.

[48] J. Alemany and E. W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *PODC*, 1992.

[49] G. Alonso, D. Agrawal, and A. El Abbadi. Reducing recovery constraints on locking based protocols. In *PODS*, 1994.

[50] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *TPDS*, 1(1), 1990.

[51] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *TOCS*, 10(1), 1992.

[52] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, et al. System r: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[53] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *PVLDB*, 8(3), 2014.

[54] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340. ACM, 2013.

[55] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD*, 1995.

[56] P. A. Bernstein and S. Das. Scaling optimistic concurrency control by approximately partitioning the certifier and log. *DE Bull*, 38(1), 2015.

[57] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221.

[58] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[59] P. A. Bernstein, C. W. Reid, and S. Das. Hyder-a transactional record manager for shared flash. In *CIDR*, volume 11, 2011.

[60] P. A. Bernstein, D. W. Shipman, and J. B. Rothnie, Jr. Concurrency control in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 5(1):18–51, 1980.

[61] B. N. Bershad. Practical considerations for lock-free concurrent objects. Technical Report CMU-CS-91-183, CMU Computer Science, 1991.

[62] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *OLS*, 2012.

[63] M. J. Cahill. *Serializable Isolation for Snapshot Databases*. PhD thesis, University of Sydney, 2009.

[64] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *SIGMOD*, 2008.

[65] B. Cantrill and J. Bonwick. Real-world concurrency. *Queue*, 6(5), 2008.

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

[67] D. Comer. The ubiquitous b-tree. *CUSR*, 11(2), 1979.

[68] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SoCC*, 2012.

[69] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[70] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, 2013.

[71] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, 1984.

[72] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *SIGMOD*, 2013.

[73] D. Dice, D. Hendler, and I. Mirsky. Lightweight contention management for efficient compare-and-swap operations. *CoRR*, abs/1305.5800, 2013.

[74] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *DISC*, 2006.

[75] D. Dice and N. Shavit. Tlrw: return of the read-write lock. In *SPAA*, 2010.

[76] B. Ding, L. Kot, A. Demers, and J. Gehrke. Centiman: Elastic, high performance optimistic concurrency control by watermarking. In *SoCC*, 2015.

[77] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge, 2006.

[78] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *CACM*, 19(11), 1976.

[79] J. M. Faleiro and D. J. Abadi. FIT: A distributed database performance tradeoff. *DE Bull*, 38(1), 2015.

[80] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11), 2015.

[81] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein. High performance transactions via early write visibility. *PVLDB*, 10(5), 2017.

[82] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD*, 2014.

[83] A. Fekete. Allocating isolation levels to transactions. In *Proc. of PODS*, 2005.

[84] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *TODS*, 30(2):492–528, 2005.

[85] A. Fekete, E. O'Neil, and P. O'Neil. A read-only transaction anomaly under snapshot isolation. *ACM SIGMOD Record*, 33(3):12–14, 2004.

[86] K. Fraser and T. Harris. Concurrent programming without locks. *TOCS*, 25(2), 2007.

[87] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS fast path. *DE Bull*, 8(2), 1985.

[88] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared database. In *IFIP*, 1976.

[89] J. Gray and A. Reuter. *Transaction processing*. Morgan Kaufmann Publishers, 1992.

[90] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, 1994.

[91] J. N. Gray. Notes on data base operating systems. In *Operating Systems*, pages 393–481. Springer, 1978.

[92] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *OSDI*, 1996.

[93] R. Gupta, J. Haritsa, and K. Ramamritham. Revisiting commit processing in distributed database systems. In *SIGMOD*, 1997.

[94] S. Harizopoulos, D. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. Proc. of SIGMOD, 2008.

[95] S. Harizopoulos and A. Ailamaki. A case for staged database systems. In *CIDR*, 2003.

[96] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, 2001.

[97] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. *Architecture of a database system*. Now Publishers, 2007.

[98] P. Henderson and J. H. Morris, Jr. A lazy evaluator. POPL, 1976.

[99] M. Herlihy. Wait-free synchronization. *TOPLAS*, 13(1), 1991.

[100] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *DISC*, 2003.

[101] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

[102] M. P. Herlihy. A methodology for implementing highly concurrent data structures. In *PPOPP*, 1990.

[103] T. Horikawa. Latch-free data structures for dbms: design, implementation, and evaluation. In *SIGMOD*, 2013.

[104] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, Sept. 1989.

[105] B. C. S. D. Indiana University, D. Friedman, and D. Wise. *Cons should not evaluate its arguments*. 1975.

[106] K. Jacobs, R. Bamford, G. Doherty, K. Haas, M. Holt, F. Putzolu, and B. Quigley. Concurrency control, transaction isolation, and serializability in SQL 92 and Oracle 7. Oracle Whitepaper, Part No. A33745, 1995.

[107] R. Johnson, I. Pandis, and A. Ailamaki. Improving OLTP scalability using speculative lock inheritance. *PVLDB*, 2(1), 2009.

[108] R. Johnson, I. Pandis, and A. Ailamaki. Eliminating unscalable communication in transaction processing. *VLDBJ*, 23(1), 2014.

[109] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A scalable storage manager for the multicore era. In *EDBT*, 2009.

[110] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *PVLDB*, 3(1-2), 2010.

[111] E. P. C. Jones, D. J. Abadi, and S. R. Madden. Concurrency control for partitioned databases. In *SIGMOD*, 2010.

[112] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. A scalable lock manager for multicores. In *SIGMOD*, 2013.

[113] A. Kemper and T. Neumann. Hyper: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[114] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *SIGMOD*, 2016.

[115] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2), 1981.

[116] B. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. 1979.

[117] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4), 2011.

[118] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on b-trees. *TODS*, 6(4), 1981.

[119] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The art of practical synchronization. In *DAMON*, 2016.

[120] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. Multi-version range concurrency control in deuteronomy. *PVLDB*, 8(13), 2015.

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

[122] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35. ACM, 2017.

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

[124] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *ICDE*, 2014.

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

[126] P. E. Mckenney, J. Appavoo, A. Kleen, O. Krieger, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *OLS*, 2001.

[127] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *TOCS*, 9(1), 1991.

[128] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PPoPP*, 1991.

[129] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *TPDS*, 15(6), 2004.

[130] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR-599, University of Rochester Computer Science, 1995.

[131] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.

[132] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *JPDC*, 51(1), 1998.

[133] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 17(1), 1992.

[134] C. Mohan and F. Levine. Aries/im: An efficient and high concurrency index management method using write-ahead logging. In *SIGMOD*, 1992.

[135] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proc. of SOSP*, 2013.

[136] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *OSDI*, 2014.

[137] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *OSDI*, 2014.

[138] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD*, 2015.

[139] P. E. O'Neil. The escrow transactional method. *TODS*, 11(4), 1986.

[140] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1-2), 2010.

[141] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. Plp: page latch-free shared-everything oltp. *PVLDB*, 4(10), 2011.

[142] P. K. Reddy and M. Kitsuregawa. Speculative locking protocols to improve performance for distributed database systems. *TKDE*, 16(2), 2004.

[143] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1):3–23.

[144] K. Ren, J. M. Faleiro, and D. J. Abadi. Design principles for scaling multi-core oltp under high contention. In *SIGMOD*, 2016.

[145] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *PVLDB*, 6(2), 2012.

[146] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *PVLDB*, 7(10), 2014.

[147] S. Revilak, P. O'Neil, and E. O'Neil. Precisely serializable snapshot isolation (pssi). In *ICDE*, 2011.

[148] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *ISCA*, 1984.

[149] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), 1990.

[150] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. *PVLDB*, 4(11), 2011.

[151] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *TODS*, 20(3), 1995.

[152] V. Srinivasan and M. J. Carey. Performance of b+ tree concurrency control algorithms. *VLDBJ*, 2(4), 1993.

[153] M. Stonebraker. Operating system support for database management. *CACM*, 24(7), 1981.

[154] M. Stonebraker, G. Held, E. Wong, and P. Kreps. The design and implementation of ingres. *ACM Transactions on Database Systems (TODS)*, 1(3):189–222, 1976.

[155] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, 2007.

[156] A. Thomson and D. J. Abadi. The case for determinism in database systems. *PVLDB*, 3(1-2), 2010.

[157] A. Thomson, T. Diamond, S. chun Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[158] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Fast distributed transactions and strongly consistent replication for oltp database systems. *ACM Trans. Database Syst.*, 39(2):11:1–11:39, May 2014.

[159] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *SOSP*, 2013.

[160] J. D. Valois. Lock-free linked lists using compare-and-swap. In *PODC*, 1995.

[161] VoltDB. Website. voltdb.com.

[162] C. Wadsworth. *Semantics and Pragmatics of the Lambda-calculus*. University of Oxford, 1971.

[163] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2), 2016.

[164] Z. Wang, S. Mu, H. Y. Yang Cui, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *SIGMOD*, 2016.

[165] T. Warszawski and P. Bailis. Acidrain: Concurrency-related attacks on database-backed web applications. In *SIGMOD*. ACM, 2017.

[166] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, et al. vcorfu: A cloud-scale object store on a shared log. In *NSDI*, pages 35–49, 2017.

[167] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.

[168] A. Whitney, D. Shasha, and S. Apter. High volume transaction processing without concurrency control, two phase commit, SQL or C++. In *HPTS*, 1997.

[169] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining acid and base in a distributed database. In *OSDI*, 2014.

[170] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance acid via modular concurrency control. In *SOSP*, 2015.

[171] C. Yao, D. Agrawal, G. Chen, Q. Lin, B. C. Ooi, W.-F. Wong, and M. Zhang. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Transactions on Knowledge and Data Engineering*, 28(10):2635–2650, 2016.

[172] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3), 2014.

[173] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, 2013.