Logical Leases: Scalable Hardware and Software Systems through Time Traveling

by

Xiangyao Yu

B.S. in Electronic Engineering, Tsinghua University (2012)
S.M. in Electrical Engineering and Computer Science, Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2018

Signature redacted

Author . . . .
Department of Electrical Engineering and Computer Science
September 29, 2017

Signature redacted

Certified by . . . .
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Signature redacted

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Logical Leases: Scalable Hardware and Software Systems through Time Traveling

by

Xiangyao Yu

Submitted to the Department of Electrical Engineering and Computer Science
on September 29, 2017, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Parallelism is the key to performance in modern computer systems. However, parallel systems are hard to design and to use. Challenges of parallelism exist across the whole system stack in both hardware and software.

In this thesis, we improve the scalability and performance of shared-memory parallel systems in both hardware and software. We propose a simple technique, called *logical leases*, to achieve this goal. Logical leases allow a parallel system to dynamically reorder operations to minimize conflicts, leading to high performance and scalability. The power of logical leases comes from its novel way of combining physical and logical time. The new notion of time, called *physiological time*, is more flexible and efficient than pure physical or logical time alone.

We implemented three systems based on logical leases.

Tardis leverages logical leases to implement a scalable cache coherence protocol for multi-core shared-memory processors. Compared to the widely-adopted directory-based cache coherence protocol, and its variants, Tardis avoids multicasting and broadcasting and only requires $\Theta(\log N)$ storage per cache block for an N-core system rather than $\Theta(N)$ sharer information. Compared to directory-based counterparts, Tardis has lower storage overheads, lower network traffic, higher performance, and yet is simpler to reason about. We present the basic protocol, its extensions and optimizations, and a formal proof of correctness of Tardis.

TicToc leverages logical leases to implement a high-performance concurrency control algorithm for on-line transaction processing (OLTP) database management systems (DBMSs). Based on logical leases, TicToc provides a simple way of serializing transactions logically at runtime. TicToc has no centralized bottleneck, and commits transactions that would be aborted by conventional concurrency control algorithms. TicToc outperforms its state-of-the-art counterparts by up to 92% while reducing aborts by up to 3.3×.

Sundial is a distributed concurrency control protocol, also based on logical leases. Sundial has two salient features. First, it uses a logical-lease-based distributed concurrency control algorithm to achieve high performance and low abort rate for distributed transactions. Second, it allows a server to cache data from remote servers at runtime while maintaining strong correctness guarantees (i.e., serializability). Sundial *seamlessly* integrates both features into a *unified* protocol which has low complexity. Evaluation shows that the distributed concurrency control algorithm in Sundial improves system performance by up to 57%, compared to the best baseline protocol we evaluate. Caching further improves performance by up to 3× when hotspot read-intensive tables exist.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I thank my advisor Srinivas Devadas for his guidance and support throughout my graduate studies. I am grateful for the freedom I had in Srini's group. Srini has always been supportive and inspiring. Srini's great taste in research (i.e., there must be some coolness factor in a project) influenced my own research standard.

I am also thankful to the rest of my committee members. I thank Andrew Pavlo for being a great collaborator and a true friend. Andy led me to the field of databases and offered unceasing help throughout the projects we collaborated on. I thank Daniel Sanchez for all his time with me brainstorming and discussing ideas. I thank Daniel also for his patience and conscientiousness in helping me prepare presentations and write papers and my thesis.

I give special thanks to Onur Mutlu, for leading to me the field of computer architecture when I was still an undergraduate student. Onur's hardworking and diligence have been inspiring me all these years. I also enjoyed the collaboration with Chris Hughes and Satish Nadathur from the parallel computing lab at Intel. They showed me a different view of research from the industry side.

I would like to thank other faculty members of CSAIL whom I had opportunities to interact with. I want to thank Michael Stonebraker, Samuel Madden, Charles Leiserson, Arvind, Frans Kaashoek, and Nickolai Zeldovich for all your help and feedback on my research. I worked as a teaching assistant with Chris Terman, Erik Demaine, and Nancy Lynch, from whom I learned how to become more effective at teaching.

I thank all my fellow MIT students. Thanks to Ling Ren, Chris Fletcher, George Bezerra, Quan Min Nguyen, Yu Xia, George Kurian, Rachael Harding, Albert Kwon for all the discussion and research collaboration. I want to thank other friends I met at MIT for making the past few years enjoyable. Thanks to Yuxuan Lin, Yu Zhang, Tianfan Xue, Changyang Linghu, Shuotao Xu, Yunming Zhang, Sizuo Zhang, Guowei Zhang, and many others. Thanks to my loyal friend Yubei Chen, for all that we experienced together over the years.

I am especially thankful to my wife, Lei Zhou, for her support and love during all my ups and downs. The PhD journey would be grinding without her at my side. I also thank my parents, Wanli Yu and Cuirong Wang, for their constant support and encouragement. They gave me the strength to pursue my dreams.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Clock frequency of single-core processors has plateaued in the mid 2000s due to the limitations of technology scaling. Since then, parallelism has become the main way to improve performance. This has led to proliferation of cores within a processor. Both hardware and software systems have to leverage parallelism for high performance.

Parallel systems, however, are notoriously hard to design and to use. Three major challenges exist: 1) Synchronization and coordination of parallel tasks limit performance. 2) Storage and communication overheads limit scalability. 3) Parallel systems are complex to design and verify. These challenges exist in multiple levels across the hardware and software stack of a parallel system.

In this thesis, we focus on *shared-memory* parallel systems, a type of parallel system widely used in both hardware (e.g., multicore processors) and software (e.g., databases). Specifically, we design and prototype three protocols to improve performance and scalability while maintaining low complexity, in their corresponding problem domains.

1. **Tardis** is a new cache coherence protocol for multicore processors. Tardis only requires $\Theta(\log N)$ storage for an N-core processor, while achieving better performance and being simpler to reason about than traditional directory-based cache coherence protocols.

2. **TicToc** is a new concurrency control protocol for multicore databases. TicToc provides high scalability, up to 92% better performance and $3.3\times$ lower abort rate compared to the best state-of-the-art counterpart we evaluated.

3. **Sundial** is a distributed concurrency control protocol for distributed databases. Sundial seamlessly integrates distributed concurrency control with caching in a unified protocol. Compared

to the best state-of-the-art protocols we evaluated, Sundial's distributed concurrency control improves performance by 57%; Sundial's caching improves performance by 300% when hotspot read-intensive tables exist.

The key idea behind the three protocols is a new concept, called *logical leases*. They are logical counters associated with data elements, allowing parallel read and write operations to be ordered logically. Logical leases are simple to implement, yet give an parallel system flexibility to reorder operations for avoiding conflicts. Logical leases are based on a new notion of time, *physiological time*, which combines the power of physical and logical time, while removing their corresponding drawbacks. Logical leases simplify the protocols in shared-memory systems, while improving performance and scalability.

1.1 Scalable of Multi-Core Cache Coherence

A multicore processor implements a shared-memory model, where the memory system appears as monolithic to the software's point of view. In practice, however, the memory system contains multiple levels of caches to hide the long latency of main memory accesses. Caches private to cores complicate the system design, since different cores may observe different values of the same address depending on the contents in their private caches. To solve this problem, a cache coherence protocol is implemented to provide the illusion of a monolithic memory, even in the presence of private caches. A cache coherence protocol guarantees that no read returns a stale value; namely, each read returns the value of the last write to the same address.

As the core count increases, however, cache coherence becomes a major scalability bottleneck. Traditional cache coherence protocols require an invalidation mechanism to enforce memory consistency. In particular, a write to a memory location must invalidate or update *all* the shared copies of that memory location in other cores' private caches. There are two major classes of cache coherence protocols: *snoopy* and *directory-based* protocols. In a snoopy protocol, a write request has to broadcast invalidations to all the cores, incurring significant network traffic overhead. In a directory-based protocol, the list of sharers of each memory location has to be maintained, incurring storage overhead and protocol complexity.

Tardis is a new cache coherence protocol to resolve the drawbacks of traditional protocols (Chapter 3). The key idea behind Tardis is to replace the invalidation mechanism with logical leases. Each shared copy in a private cache contains a logical lease. A shared copy self-invalidates after the lease (logically) expires. A write request jumps ahead in logical time (i.e., time-travels) to the end of the lease. Logical time is maintained in each processor and advances asynchronously, therefore requiring no clock synchronization.

Tardis has three salient features. First, Tardis has small storage overheads that scale well to large core counts. Second, Tardis has low network traffic as it does not broadcast or multicast messages. Third, Tardis is easy to reason about as it explicitly exposes the memory order using logical time. Our experimental evaluation shows that Tardis has similar or better performance compared to a full-map sparse directory-based coherence protocol over a wide range of benchmarks on processors with varying core counts.

On top of the basic Tardis protocol, we propose three new extensions and optimizations making Tardis more practical (Chapter 4). First, we extend Tardis to support relaxed memory models like *total store order* (TSO) and *release consistency* (RC) besides *sequential consistency* (SC). These relaxed models are important since they are widely used in commercial processors due to their better performance. Second, we propose a *livelock detection* mechanism to improve performance when a core spins on memory locations for inter-core communication. Third, we propose a *lease prediction* mechanism to dynamically determine the value of a logical lease to reduce the number of lease expirations, further improving performance.

Finally, we formally prove the correctness of the basic Tardis protocol, namely, that it implements the target memory consistency model (Chapter 5). The proof also shows that the protocol is deadlock- and livelock-free. Our proof technique is based on simple and intuitive system invariants. In contrast to model checking [42, 79, 109] verification techniques, our proof applies to any number of cores in the processor. More important, the invariants developed in the proof are more intuitive to system designers and thus can better guide system implementation.

1.2 Scalable Concurrency Control for Databases

Concurrency control algorithms have a long history dating back to the 1970s [47, 16]. Traditional concurrency control algorithms, however, were designed for disk-based databases with a small number of processor. As OLTP databases start to fit in main memory and core counts increase to hundreds, traditional algorithms are not necessarily the best fit.

With commercial processors now supporting tens to hundreds of hardware threads, it is challenging for software applications to fully exploit this level of parallelism. Depending on the computation structure, it is more difficult to exploit parallelism in some applications than others. Concurrency control in database transaction processing is among the applications where parallelism is the most difficult to exploit. Concurrency control guarantees that transactions (i.e., a sequence of reads and writes to the shared database) are executed atomically with respect to each other. Concurrency control requires complex coordination among transactions, and the coordination overhead increases with the level of concurrency. This leads to both artificial and fundamental scalability bottlenecks for traditional concurrency control algorithms.

We perform a study to understand how traditional concurrency control algorithms work on modern and future multicore processors (Chapter 6). We implement seven classic algorithms and run them on a simulated 1000-core processor. We found none of them is able to scale to this level of parallelism. For each algorithm, we identify its fundamental scalability bottlenecks.

To resolve these bottlenecks, we design TicToc, a new concurrency control protocol that achieves better scalability and performance than state-of-the-art counterparts (Chapter 7). The key idea in TicToc is to assign a logical lease to each data element (i.e., a tuple) in the database, and to dynamically determine the commit order of transactions based on the leases of tuples accessed by each transaction. This *data-driven timestamp assignment* brings two major benefits. First, it removes the timestamp allocation bottleneck, since leases are stored and managed in a distributed manner. Second, it reduces the abort rate, since the commit order is dynamically calculated based on the data access pattern of transactions. Implemented on an in-memory database on a 40-core processor, TicToc improves performance by up to 92% and reduces aborts by up to 3.3× compared to state-of-the-art counterparts.

Finally, we apply logical leases to distributed systems and design the Sundial distributed concurrency control protocol (Chapter 8). Sundial is the first protocol that seamlessly integrates concurrency control with caching in a unified way. While distributed concurrency control and caching are both complex, the power of logical leases makes them easy to combine. The distributed concurrency control algorithm in Sundial improves performance of distributed transactions by up to 57%, compared to the best state-of-the-art protocol we evaluated. Caching can further improve performance by up to 3× when hotspot read-intensive tables exist.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides relevant background and motivation. The thesis contains two major parts.

Part I of the thesis focuses on the Tardis cache coherence protocol. We describe the basic Tardis protocol in Chapter 3, optimizations and extensions to Tardis in Chapter 4, and a formal proof of correctness of Tardis in Chapter 5.

Part II of the thesis focuses on concurrency control in database transaction processing. We present an evaluation of classical concurrency control algorithms on a simulated 1000-core processor in Chapter 6. We describe TicToc, our new concurrency control protocol for multicore processors, in Chapter 7. We describe Sundial, our new distributed concurrency control protocol, in Chapter 8. Finally, Chapter 9 concludes the thesis and discusses future work.

Chapter 2

Background

In this chapter, we present the background and some prior work related to this thesis. In Section 2.1, we discuss cache coherence protocols in multicore processors. In Section 2.2, we discuss concurrency control algorithms in database transaction processing.

2.1 Multicore Processors

A multicore processor or chip multiprocessor (CMP) is a single processor with multiple processing units (i.e., cores). Figure 2-1 shows a simplified architecture of a multicore processor. A core executes a stream of instructions and interacts with the memory system through *reads* and *writes*. The order in which the instructions are executed by a core is the program order of the core. The memory system contains multiple levels of caches and a main memory. Caches closer to the cores have lower latency and higher bandwidth but smaller capacity due to the higher cost. Although

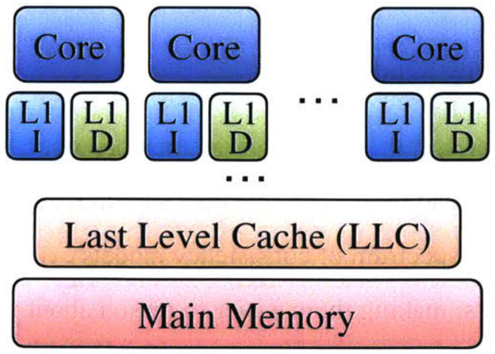


Figure 2-1: **Architecture of a Multi-Core Processor** – Each core has a private L1 instruction cache (L1I) and a private L1 data cache. The last-level cache (LLC) and main memory are shared across all the cores.

the majority of a program's instructions and data cannot fit in caches due to their small capacity, practical applications exhibit *spatial* and/or *temporal* locality such that considerable fraction of memory accesses are served by a cache. Therefore, the design of the cache system is critical to the overall system performance.

A cache can be either *private* or *shared.* A private cache can be accessed only by its corresponding core. In a level 1 cache (L1), the instruction (L1I) and data (L1D) caches are typically maintained as separate structures. The higher levels (e.g., L2, L3, etc.) in the cache hierarchy may be private or shared by some cores, depending on the design. For our example in Figure 2-1, the last level cache (LLC) is shared across all the cores. Finally, the main memory is located off-chip and is shared by all cores.

For ease of programming, a multicore processor provides a *shared-memory* abstraction to software to hide the complexity of the memory system. All cores share the same address space. Cores communicate by writing and reading the same memory location.

The shared-memory programming model is intuitive to use compared to alternative approaches where the software has to explicitly perform inter-core communication (e.g., MPI [64, 65]) or data partitioning (e.g., PGAS [156, 134]). As a result, virtually all multicore processors implemented the shared-memory model.

However, the simplicity in programmability comes at the cost of complexity in design. It is a challenge to efficiently provide a shared-memory interface in hardware due to the complex structure of a memory system. The memory model a multicore processor provides represents a trade-off between programmability and hardware efficiency. Specifically, the behavior of a multicore processor is specified via a *memory consistency* model (cf. Section 2.1.1), which is implemented based on a *cache coherence* protocol (cf. Section 2.1.2).

2.1.1 Memory Consistency Models

A memory consistency model specifies the rules of memory operations that the hardware must follow. It defines what value a load request from a core should return. It is a contract between programmers and the processor. A spectrum of consistency models exist. *Stronger* consistency models have more stringent constraints, making them easier to reason about for programmers, but perform worse due to their inflexibility in reordering memory operations. In contrast, *weaker* or *relaxed* consistency models are able to execute more memory operations concurrently, thereby obtaining better performance. However, relaxed models have more complex rules, making it hard for programmers to write correct code.

Sequential consistency (SC) is one of the strongest consistency models in multicore processors. It defines the correctness of a parallel program with respect to a sequential program. The sequential program is derived by interleaving the operations performed by different cores.

According to its definition, a parallel program is sequentially consistent if *"the result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program"* [93]. Sequential consistency is relatively easy to reason about since it relates a parallel program to an equivalent sequential program; and a sequential program is well studied and easy to understand.

Most of our discussions in Part I of the thesis assume sequential consistency as the underlying memory model. We will also discuss relaxed memory models in Chapter 4 and Chapter 5.

2.1.2 Cache Coherence Protocols

A cache coherence protocol is a crucial component in a multicore processor for implementing a memory consistency model. Intuitively, coherence means that for any data block, if it is stored in multiple private caches, then all the copies of the block must have the same value. Specifically, a cache coherence protocol enforces the *single-writer multiple-reader* (SWMR) invariant, which means that for any given memory location, at any given moment in *time*, there is either a single core that may write it (and that may also read it) or some number of cores that may read it. Here, the notion of *time* means the global memory order defined in the consistency model. It can be either physical, logical, or some combination of both.

The definition above is not the only definition of cache coherence, and others exist in the literature [8, 54, 56, 71] that are equivalent to each other. From the perspective of a software application, only the consistency model is important. Coherence is just a means to implement consistency. However, coherence and consistency are usually considered separately in hardware to reduce design complexity and to improve modularity. As we will show later in the thesis (Chapter 3 and 4), Tardis takes a holistic approach to handle consistency and coherence. It directly implements the consistency model in hardware without requiring a traditional cache coherence protocol.

Many cache coherence protocols have been proposed in the literature. We discuss a few popular classes of protocols in this section. For each class of cache coherence protocols, we also discuss their scalability bottlenecks when implemented in a multicore system with hundreds to thousands of cores.

Snoopy cache coherence protocols [58] were widely used in processors with small core counts. The protocol enforces the SWMR invariant via an *invalidation mechanism*. Before a core performs a write, it broadcasts the invalidation requests on the address to all the other cores through a shared bus. Each core snoops on the bus, and upon seeing the request, invalidates or updates the requested block in its private cache (if such a block exists). The write is performed after all the cached blocks are invalidated or updated. At this point, the block with the requested address exists only in the requesting core's private cache (i.e., single-writer); when no core is writing to a block, the block can be shared in multiple private caches (i.e., multiple-reader).

The advantage of a snoopy cache coherence protocol is its design simplicity and low latency in cache-to-cache data transfer. Snoopy protocols, however, have the following two major disadvantages. First, the shared bus is a major scalability bottleneck. Previous works have proposed to implement a *virtual bus* on top of scalable interconnection techniques [104, 13, 105, 136]. These designs provide ordering guarantees among memory accesses without requiring a central ordering point. Second, a more fundamental scalability bottleneck in snoopy protocols is broadcasting memory requests through out the whole system. Previous works have proposed *snoop filtering* to reduce broadcasting messages by throttling broadcasting at the source [27, 114] or in the network [12]. As the core count continued increasing, however, the cost of broadcasting increased and limited system scalability.

Directory-based cache coherence protocols [29, 137] also use the invalidation mechanism, but are more scalable than snoopy protocols. Instead of broadcasting invalidations globally, invalidations are only delivered to private caches that have a copy of the requested block. Compared to a snoopy cache coherence protocol, the amount of network traffic for invalidations is significantly reduced, leading to better scalability. However, directory-based protocols maintain a directory structure to track which caches (i.e., the sharer list) have shared copies of each data block. The sharer list incurs significant storage overhead as the system scales. For an N-core system, a full-map directory requires N-bit storage for each data block. This translates to a prohibitive 200% storage overhead at 1000 cores (i.e., 1000 bits for the sharer list vs. 512 bits for the data in each block).

Previous works have proposed optimizations to reduce the storage overhead of the directory structure. Hierarchical directories [146, 101] store the sharer list hierarchically, where a higher level directory encodes whether a block is shared in a group of private caches. Coarse sharing vectors [66, 161] reduce the storage overhead by tracking multiple sharers using fewer bits; these protocols incur unnecessary invalidations. SCD [127] tracks exact sharers using variable sharing vector representations. All the designs described above require complex directory organization. Tardis, in contrast, completely eliminates the directory structure, thereby avoiding the complexity associated with it.

Lease-based cache coherence protocols [59, 99, 131] get rid of the invalidation mechanism through using leases. Upon a private cache miss, both the data and a lease are returned and stored in the private cache. The data in a private cache is read only if it has not expired, i.e., the current clock time is earlier than the end of the block's lease. A read to an expired block is considered a cache miss. In order to preserve the SWMR invariant, a block is written to only after the leases of all the shared copies of that block have expired. Compared to invalidation-based protocols (i.e., snoopy and directory), lease-base protocols have lower design complexity. Previous studies, however, show that lease-based protocols have worse performance due to 1) excessive cache misses caused by lease expiration, and 2) longer write latency due to waiting for leases to expire [99]. Furthermore, these protocols require globally synchronized clocks in all private and shared caches, which is complex to implement as the system scales.

Consistency model specific coherence protocols exploit the fact that processors or compilers typically implement relaxed memory consistency models instead of the strong sequential consistency. Traditional cache coherence protocols, however, are designed to work efficiently with *all* consistency models. Therefore, traditional cache coherence protocols may be an overkill if the system only needs relaxed consistency. When optimized for a relaxed consistency model, the coherence protocol has less stringent constraints, leading to protocols with lower complexity and better scalability. Prior art has explored coherence protocols designed for consistency models including Total Store Order (TSO) [45], Release Consistency (RC) [15], and Data-Race-Free programs (DRF) [33, 125]. The downside of these protocols is that they can only support a certain relaxed consistency model, which limits their applicability.

2.2 OLTP Databases

Modern database management systems (DBMSs) have two major application domains: *on-line analytical processing* (OLAP) and *on-line transaction processing* (OLTP). OLAP DBMSs are used for extracting important information from a database through running complex queries over the data. These databases are typically read-only or read-intensive. Due to the large quantity of data being processed, OLAP queries are relatively straightforward to parallelize over multiple cores or servers.

OLTP DBMSs, in contrast, support the part of an application that interacts with end-users. An end-user interacts with the database by sending it requests to perform some function in the form of a transaction. A transaction performs a sequence of one or more operations over the shared database (e.g., reserve a seat on a flight, or transfer money between bank accounts). At a high level, it is challenging to scale OLTP DBMS for two reasons: 1) a large number of transactions can run in the database at the same time, and 2) different transactions may conflict with each other leading to high coordination overhead.

An OLTP DBMS is expected to maintain four properties for each transaction that it executes: (1) *atomicity*, (2) *consistency*, (3) *isolation*, and (4) *durability*. These unifying concepts are collectively referred to with the *ACID* acronym [68]. Specifically, *atomicity* guarantees that a transaction either succeeds or fails, leaving the database unchanged if it does fail. *Consistency* implies that every transaction will bring the database from one valid state to another. *Isolation* specifies when a transaction is allowed to see modifications made by other transactions. *Durability* requires that a committed transaction must be recoverable if the DBMS crashes.

Concurrency control is a critical component in an OLTP DBMS. It essentially provides the atomicity and isolation guarantees to the system. Similar to memory consistency models, many different isolation levels exist. Strong isolation levels are more intuitive to use, but provide lower performance. In contrast, relaxed isolation levels allow more concurrency, thereby better performance; but they introduce anomalies into the system which may cause concurrency bugs.

The strongest isolation level, *serializability*, requires that the effect of a group of transactions on the database's state is equivalent to some sequential execution of all transactions. The definition is similar to the definition of sequential consistency. Both definitions use an equivalent *sequential* execution to define the correctness of a parallel execution. When transactions access overlapping data elements, concurrency control requires sophisticated inter-transaction coordination in order to enforce serializability. As the level of parallelism increases, concurrency control can become a serious scalability bottleneck of a DBMS.

2.2.1 Concurrency Control Algorithms

In this section, we briefly describe some classic concurrency control algorithms. For each type of algorithm, we describe how it works, and show the potential scalability bottlenecks in it. Three classes of concurrency control algorithms will be discussed here [18, 86]: *two-phase locking* (2PL), *timestamp ordering* (T/O), and *optimistic concurrency control* (OCC).

Two-Phase Locking (2PL) [21, 47] protocols are pessimistic and allow a transaction to access an object in the database only after acquiring a lock with the proper permission (read or write). Each transaction contains two phases: locks are acquired during the *growing phase* and released during the *shrinking phase*. Since deadlocks may occur due to cycles in the lock dependency graph, different techniques can be used to prevent or detect-and-resolve them. A major scalability bottleneck in 2PL algorithms is the *lock thrashing* problem. Thrashing happens when transactions spend significant amount of time waiting for locks, while the locks held by a transaction block other concurrent transactions. The problem aggravates in highly contentious workload.

Timestamp ordering (T/O) concurrency control schemes [18] generate a serialization order of transactions *a priori* and then the DBMS enforces this order. A transaction is assigned a unique, monotonically increasing timestamp before it is executed; this timestamp is used by the DBMS to resolve conflicts between transactions [18]. T/O schemes can use multi-versioning to further improve concurrency. The resulting algorithm is called multi-version concurrency control (MVCC). In MVCC, a write creates a new version of a tuple in the database instead of overwriting the previous version. This allows a read operation that arrives late (the timestamp of the reading transaction is smaller than the version of the latest data) to proceed instead of aborting.

A major scalability bottleneck in T/O schemes is the timestamp allocation process. The timestamps need to be unique and monotonically increasing. Two solutions are commonly used: 1) a centralized and atomically incremented counter, and 2) globally synchronized clocks. The first solution creates a central bottleneck (i.e., the counter) that limits the throughput of the system to a few million transactions per second; the second solution is complex to implement and not supported in all processors today.

Optimistic Concurrency Control (OCC) [86] schemes assume conflicts are infrequent. Transactions are speculatively executed without holding locks. At the end of its execution, a transaction verifies that it is serializable with respect to other transactions. If serializability is violated, the transaction aborts and may restart later. Since no locks are maintained during execution, a transaction does not block or wait for other transactions, eliminating the lock thrashing problem during execution. At low contention, OCC algorithms perform well. When contention is high, however, OCC algorithms lead to more aborts; and each abort is relatively expensive because it is detected only at the end of the execution, at which point the transaction has already consumed valuable resources (e.g., CPU time, cache capacity, and memory bandwidth).

The concurrency control algorithms described above can be applied to both multicore processors and distributed systems. The scalability bottlenecks described here also apply to both settings. In Chapter 6, we discuss the scalability bottlenecks of these algorithms in more detail. Then, we describe our solutions to improve the scalability of concurrency control in Chapter 7 and Chapter 8, for the settings of multicore processors and distributed systems, respectively.

Part I

Multi-Core Cache Coherence

Chapter 3

Tardis Cache Coherence

3.1 Introduction

As discussed in Section 2.1, a major scalability bottleneck of a multicore processor is the cache coherence protocol. The directory cache coherence protocol is the *de facto* standard in processors having a large number of cores. But the storage requirement of a full-map directory structure does not scale with the number of cores; and optimized directory structures are complex and usually hurt performance.

In this chapter we present Tardis, a logical-lease-based cache coherence protocol that is simpler and more scalable than directory-based protocols, but has equivalent performance. Tardis eliminates the invalidation mechanism entirely through the use of leases. Therefore, the storage for sharer information is not required. To resolve the downsides of traditional lease-based protocols, Tardis *uses logical instead of physical leases* and determine the order among operations using a combination of physical and logical time. Unlike previously proposed physical-lease-based cache coherence protocols (cf. Section 2.1.2), Tardis uses *logical* leases, which eliminates the requirement of clock synchronization, and allows operations to "jump forward in time" to avoid conflicts. This chapter presents the following contributions.

1. We present the basic Tardis protocol, the first cache coherence protocol that uses logical time as the fundamental building block. In Tardis, only timestamps (requiring $\Theta(1)$ storage) and the ID of the owner core (requiring $\Theta(\log N)$ storage) need to be stored for each memory location for an $\Theta(\log N)$ cost in total, where N is the number of cores. A full-map directory-based cache coherence protocol, in contrast, requires $O(N)$ bits to store the sharer information.

2. We propose optimizations and extensions to improve the performance of Tardis. These optimizations include speculative reads to hide latency of renewals, timestamp compression, supporting Out-of-Order (OoO) processors, and supporting remote word accesses.

3. We evaluated Tardis in the Graphite multicore simulator. Our experiments show that Tardis achieves similar performance to its directory-based counterparts over a wide range of benchmarks. Due to its simplicity and excellent performance, we believe Tardis is a competitive alternative to directory-based cache coherence protocols for massive-core and distributed shared memory (DSM) systems.

The rest of the chapter is organized as follows. We provide a formal definition of sequential consistency in Section 3.2. We then introduce the notion of logical leases in Section 3.3. We describe the basic Tardis protocol in Section 3.4, and optimizations in Section 3.5. We present a case study in Section 3.6, evaluate the performance of Tardis in Section 3.7, discuss additional related work in Section 3.8.

3.2 Sequential Consistency

In Section 2.1.1, we already saw a definition of sequential consistency. In this section, we provide a more formal definition. The rest of the chapter will occasionally refer to this definition.

Sequential consistency can be defined using the following two rules [149]. We use X and Y to represent a memory operation (i.e., either a load or a store) performed by a core, $L(a)$ and $S(a)$ to represent a load or a store to address a, respectively. We use $<_p$ and $<_m$ to denote program order and global memory order, respectively. The $Max_{<m}$ operator selects from a set the most recent operation in the global memory order.

Rule 1: $X <_p Y \implies X <_m Y$

Rule 2: $\textit{Value of } L(a) = \textit{Value of } Max_{<m}\{S(a)|S(a) <_m L(a)\}$

Rule 1 says that if operation X precedes another operation Y in the program order in any core, X must precede Y in the global memory order as well. This means that the program order is preserved when interleaving parallel operations. Rule 2 says that a load to an address should return the value of the most recent store to that address with respect to the global memory order. This guarantees that the parallel program produces the same result as the sequential one defined by the global memory order.

In virtually all multicore processors implemented in practice and proposed in the literature, the global memory order is simply the physical time order. This makes it trivial to implement Rule 1 in sequential consistency, since processors just execute instructions in the time order as they already do.[1] For Rule 2, a traditional multicore system guarantees that all reads to an address return the value of the last write to the address in *physical time order*. In other words, once a write operation is performed, it must be seen instantaneously by all the cores. This requirement causes the complexity and overhead in traditional cache coherence protocols (cf. Section 2.1.2). In the next section, we describe the solution that Tardis adopts, which avoids these problems by introducing logical time into the system.

3.3 Logical Leases

A shared memory system orders memory operations to the same memory location. According to Rule 2 of sequential consistency (cf. Section 3.2), a read is ordered after a write if the read returns the value created by the write. Meanwhile, a read is ordered before a write if the write overwrites the value that the read returns. For systems that use physical time to determine the memory order, after a write is successfully performed to an address, no core sees the previous value of the address anymore.

Physical time is not the only means to determine an order among operations. Logical time can achieve the same goal. While logical time has been widely used in distributed systems to order events [92, 51, 106], it has not been widely applied to shared memory systems. In this section, we propose logical leases as a simple mechanism to determine the order among reads and writes in a shared memory system.

A logical lease is bound to each memory location (e.g., a cacheline in a multicore processor) in the shared memory. A logical lease comprises two integer timestamps, *wts* and *rts*, to indicate the start and the end of the lease, respectively. Conceptually, the *wts*, or *write timestamp*, means that the cacheline was written at logical time *wts*. The *rts*, or *read timestamp*, means that the cacheline is valid until logical time *rts*. A read to a cacheline must happen at a timestamp (*ts*) between *wts* and *rts* ($wts \leq ts \leq rts$); the read extends *rts* when necessary. A write to a cacheline must happen at a timestamp (*ts*) greater than the current *rts* of the line ($ts \geq rts + 1$); the write also updates both *wts* and *rts* to the timestamp of the write (*ts*) to reflect the new version.

[1] In an Out-of-Order processor, the instructions commit in-order.

Definition 1 gives the definition of physiological time.[2] We show that with logical leases, all the memory operations can be properly ordered according to this new notion of time. In the definition, $<_m$ is the global memory order, $<_{ts}$ is the timestamp order and $<_{pt}$ is the physical time order.

Definition 1 (Physiological Time).

$$X <_m Y \ := \ X <_{ts} Y \ or \ (X =_{ts} Y \ and \ X <_{pt} Y)$$

In Definition 1, an operation X precedes an operation Y in the global memory order if X has a smaller timestamp, or X happens earlier in physical time if they both have the same logical timestamp.

Using this definition, all reads and writes to the same memory location are properly ordered. Namely, the effect of these memory operations is the same as if they are sequentially executed in the physiological time order. Specifically, if a read returns the value created by a write, the read must have a timestamp no less than that of the write; if they have the same logical timestamp, the read must happen after the write in physical time order. Therefore, the read is ordered after the write in physiological time order. Similarly, if a write overwrites the value returned by a read, then the write must have a greater timestamp than the that of the read, and therefore be ordered after the read in physiological time order.

Logical leases provide a general technique to partially order reads and writes to a shared address space. The technique can be applied to different shared memory systems. We discuss its application to cache coherence protocols in the rest of this chapter, Chapter 4, and Chapter 5. We will talk about how it is applied to transaction processing in Part II of the thesis.

3.4 Tardis Protocol

Traditional cache coherence protocols are treated separately from the consistency model. In contrast, Tardis directly implements the memory consistency model using logical leases, without requiring a coherence protocol in a traditional sense. Each core maintains a program timestamp (*pts*) to indicate the program order. *pts* is the timestamp of the last memory operation executed by the core; it is the logical time that the core is at. In sequential consistency, *pts* monotonically increases in the program order. Note that *pts* should not be confused with the processor (physical) clock. *pts*

[2]The name physiological time comes from an easier pronunciation of physical-logical time.

is purely *logical* and advances asynchronously, requiring *no* clock synchronization.

3.4.1 Tardis without Private Cache

For illustrative purposes, we first show the Tardis protocol assuming no private caches and all data fitting in the shared last level cache (LLC). Each cacheline has a unique copy in the LLC which serves all the memory requests. The lease is attached to each cacheline. Although no coherence protocol is required in such a system, the protocol in this section provides necessary background in understanding the more general Tardis protocol in Section 3.4.2.

Table 3.1 shows the timestamp management policy in Tardis. The timestamp of a load request (*ts*) is the maximum of the requesting core's *pts* and the accessed cacheline's *wts*. *ts* cannot be less than *pts* because the timestamp of a memory operation must monotonically increase in the program order (i.e., time should only move forward). *ts* cannot be less than *wts* because the data does not exist before it is created. The *rts* of the cacheline is extended to *ts* if necessary. Finally, *pts* is updated to the new *ts*.

Table 3.1: **Timestamp Rules without Private Caches** – The timestamp management in Tardis without Private Caches. The timestamp of the current memory operation is *ts*.

Request Type	Load Request	Store Request
Timestamp Operation	$ts = Max(pts, wts)$ $rts = Max(ts, rts)$ $pts = ts$	$ts = Max(pts, rts + 1)$ $wts = rts = ts$ $pts = ts$

For a store request, the timestamp of the store (*ts*) is the maximum of the requesting core's *pts* and the accessed cacheline's $rts + 1$. *ts* cannot be less than or equal to *rts* because the previous version of the line was still valid at that logical time. After the write, both *wts* and *rts* of the cacheline are updated to *ts*, reflecting the creation of a new version. Finally, *pts* is updated to *ts*.

Following Table 3.1, both rules in sequential consistency (cf. Section 3.2) can be enforced; and the global memory order is the physiological time order. We make the assumption that a processor performs instructions in the program order, namely, $X <_p Y \implies X <_{pt} Y$. The proof below is an informal sketch; a formal proof can be found in Chapter 5.

According to Table 3.1, *pts* monotonically increases in the program order. This means $X <_p Y \implies X \leq_{ts} Y$. Using the in-order assumption above, we have $X <_p Y \implies X \leq_{ts} Y \text{ and } X <_{pt} Y \implies X <_{ts} Y \text{ or } (X =_{ts} Y \text{ and } X <_{pt} Y) \implies X <_m Y$. Therefore, Rule 1 is proven.

For Rule 2, the value of a load $L(a)$ equals the value of the store $S(a)$ that has a timestamp equal to the *wts* of the cacheline returned by the load. Therefore $S(a) <_{ts} L(a)$ *or* $(S(a) =_{ts} L(a)$ *and* $S(a) <_{pt} L(a))$ which means $S(a) <_m L(a)$. To prove Rule 2, we just need to show that no other store $S'(a)$ exists such that $S(a) <_m S'(a) <_m L(a)$. This is true because $L(a)$ extends the *rts* of the lease on cacheline a to the timestamp of $L(a)$. This protects the cacheline from being overwritten at any timestamp within the lease.

3.4.2 Tardis with Private Cache

With private caching, data can be replicated in private caches. For simplicity, we assume a 2-level cache hierarchy with private L1 caches and a shared L2 cache. The timestamp manager is the structure that replaces the directory in a directory-based coherence protocol. It collocates with the LCC. The protocol discussed in Section 3.4.1 largely remains the same. However, two extra mechanisms are needed.

Timestamp Leasing: Each shared cacheline in a private cache takes a logical lease. This allows a core to read the cached data for a period of logical time, instead of going to the LLC for each access. The cacheline can be read until the lease expires (i.e., $pts > rts$). After the lease expires, however, a request must be sent to the timestamp manager to extend the lease.

Exclusive Ownership: Like in a directory protocol, a modified cacheline can be exclusively owned by a private cache. In the timestamp manager, the owner of the cacheline is stored, which requires $\log N$ bits. The data can be read or written by the owner core as long as it is still the owner; and the timestamps may be updated with each access. If another core later accesses the same cacheline, a write back (the owner continues to cache the line in shared state) or flush (the owner invalidates the line) request is sent to the owner. The owner replies with the latest data and timestamps after receiving such requests (i.e., write back or flush requests).

The timestamp manager only tracks the leases of a limited number of cachelines, namely, the cachelines exist in on-chip caches. To maintain the timestamps of the large quantity of data in main memory, a new pair of timestamps (*mwts* and *mrts*) are introduced. They indicate the largest *wts* and *rts*, respectively, of the cachelines in the main memory. They effectively compress timestamps of all data in main memory into a single pair of timestamps, which can be stored on chip.

The state transition and timestamp management of Tardis in private and shared caches are shown in Table 3.2 and Table 3.3, respectively. Table 3.4 shows the network message types used in Tardis.

In the protocol, each cacheline (denoted as *D*) has a write timestamp (*D.wts*) and a read timestamp (*D.rts*). Initially, all timestamps (*wts*, *rts*, *pts*, *mrts* and *mwts*) in the system are zeroes and

Table 3.2: **State Transition in a Private Cache** – State Transition of Tardis in a private cache. *TM* is the shared timestamp manager, *D* is requested cacheline, *req* and *resp* are the request and response messages. Timestamp related rules are highlighted in red.

States	Core Event			Network Event		
	Load	Store	Eviction	SH_REP or EX_REP	RENEW_REP or UPGRADE_REP	FLUSH_REQ or WB_REQ
Invalid	Miss ***req.wts = null*** ***req.pts = pts*** *req.type* = SH_REQ send *req* to TM	Miss ***req.wts = null*** *req.type* = EX_REQ send *req* to TM		Fill in data **SH_REP** ***D.wts = resp.wts*** ***D.rts = resp.rts*** state = Shared **EX_REP** ***D.wts = resp.wts*** ***D.rts = resp.rts*** state = Modified		
Shared $pts \leq rts$	Hit ***pts* = Max(*pts*, *D.wts*)**	Miss ***req.wts = D.wts*** *req.type* = EX_REQ send *req* to TM	state = Invalid No msg sent.		**RENEW_REP** ***D.rts = resp.rts*** **UPGRADE_REP** ***D.rts = resp.rts*** state = Modified	
Shared $pts > rts$	Miss ***req.wts = D.wts*** ***req.pts = pts*** *req.type* = SH_REQ send *req* to TM					
Modified	Hit ***pts* = Max(*pts*, *D.wts*)** ***D.rts* = Max(*pts*, *D.rts*)**	Hit ***pts* = Max(*pts*, *D.rts* + 1)** ***D.wts = pts*** ***D.rts = pts***	state = Invalid ***req.wts = D.wts*** ***req.rts = D.rts*** *req.type* = FLUSH_REP send *req* to TM			**FLUSH_REQ** ***resp.wts = D.wts*** ***resp.rts = D.rts*** *resp.type* = FLUSH_REP send *resp* to TM state = Invalid **WB_REQ** ***D.rts* = Max(*D.rts*, *req.rts*)** ***resp.wts = D.wts*** ***resp.rts = D.rts*** *resp.type* = WB_REP send *resp* to TM state = Shared

all caches are empty. Some network messages (denoted as *req* or *resp*) also have timestamps associated with them. The protocol shown here is a MSI protocol with three possible states for each cacheline, *Modified*, *Shared* and *Invalid*. Support for other states (e.g., Exclusive and Owned) will be discussed in Section 3.5.

State Transition in Private Cache (Table 3.2)

Load to Private Cache (column 1, 4, 5): A load to the private cache is considered as a hit if the cacheline is in exclusive state or is in shared state and has not expired ($pts \leq rts$). Otherwise, a SH_REQ is sent to the timestamp manager to load the data or to extend the existing lease. The request message has the current *wts* of the cacheline indicating the version of the cached data.

Store to Private Cache (column 2, 4, 5): A store to the private cache can only happen if the cacheline is exclusively owned by the core, which is the same as a directory coherence protocol. Otherwise, EX_REQ is sent to the timestamp manager for exclusive ownership. For a hit, both the *rts* and *wts* of the cacheline are both updated to $Max(pts, rts + 1)$.

Eviction (column 3): Evicting shared cachelines does not require sending any network message. The cacheline is simply dropped. This behavior is very different from directory-based protocols where a message is sent to the directory to remove the core from the sharer list. Tardis does not maintain sharer list since it is lease-based. To evict a cacheline in modified state, the data needs to be returned to the LCC (through a FLUSH_REP message).

Flush or Write Back (column 6): Exclusive cachelines in the private caches may receive flush or write back requests from the timestamp manager if the cacheline is evicted from the LLC or accessed by another core. A flush is handled similarly to an eviction where the data is returned and the line invalidated. For a write back request, the data is returned but the line transitions to shared state.

State Transition in Timestamp Manager (Table 3.3)

Table 3.3: **State Transition in the Timestamp Manager**.

States	SH_REQ	EX_REQ	Eviction	DRAM_REP	FLUSH_REP or WB_REP
Invalid	Send request to DRAM			Fill in data ***D.wts = mwts*** ***D.rts = mrts*** state = Shared	
Shared	***D.rts* = Max(*D.rts*, *req.pts* + *lease*, *D.wts* + *lease*)** **if *req.wts* == *D.wts*** ***resp*.rts = *D*.rts** *resp.type* = RENEW_REP send *resp* to requester **else** ***resp.wts = D.wts*** ***resp.rts = D.rts*** *resp.type* = SH_REP send *resp* to requester	***resp.rts = D.rts*** state = Modified **if *req.wts* == *D.wts*** *resp.type* = UPGRADE_REP send *resp* to requester **else** **M.wts = *D*.wts** *resp.type* = EX_REP send *resp* to requester	***mwts* = Max(*mwts*, *D.wts*)** ***mrts* = Max(*mrts*, *D*.rts)** Store data back to DRAM if dirty state = Invalid		
Modified	***req'.rts = req.pts + lease*** send *req'* to the owner	*req'.type* = FLUSH_REQ send *req'* to the owner			Fill in data ***D.wts = resp.wts*** ***D.rts = resp.rts*** state = Shared

Shared request to the timestamp manager (column 1): If the cacheline is invalid in the LLC, it must be loaded from DRAM. If it is exclusively owned by another core, then a write back request (WB_REQ) is sent to the owner. When the cacheline is in the *Shared* state, it is reserved for a period of logical time by setting the *rts* to be the end of the timestamp lease, and the line can only be read from *wts* to *rts* in the private cache.

If the *wts* of the request equals the *wts* of the cacheline in the timestamp manager, the data in the private cache must be the same as the data in the LLC. So a RENEW_REP is sent back to the requesting core. The RENEW_REP message does not contain data payload. Otherwise SH_REP is sent back with the data. RENEW_REP incurs much less network traffic than a SH_REP.

Exclusive request to the timestamp manager (column 2): An exclusive request can be caused by a store or exclusive read from the core. Similar to a directory-based protocol, if the cacheline is invalid, it should be loaded from DRAM; if the line is exclusively owned by another core, a flush request is sent to the owner.

If the requested cacheline is in shared state, however, *no invalidation messages are sent.* The timestamp manager can immediately give exclusive ownership to the requesting core which bumps up its local *pts* to the cacheline's current *rts* + 1 when performing the write, i.e., jumps ahead in time. Other cores still read their local copies of the cacheline if those local copies have not expired according to the cores' *pts*. This does not violate sequential consistency since the read operations in the sharing cores are ordered before the write operation in physiological time order, even though the read may happen after the write in physical time order. If the cacheline expires in a sharing core, the core will send a request to renew the line at which point the latest version of the data is returned.

If the *wts* of the request equals the *wts* of the cacheline in the timestamp manager, an UPGRADE_REP is sent to the requester without the data payload.

Evictions (column 3): Evicting a cacheline in modified state is the same as in directory protocols, i.e., a flush request is sent to the owner. The cacheline is removed from the cache only after the flush response comes back. To evict a shared cacheline, however, no invalidation messages are sent. Sharing cores can still read their local copies until they expire. Again, correctness is not violated since the memory operations are ordered using both logical and physical time instead of pure physical time.

Main memory (column 3, 4): Tardis only stores timestamps on chip but not in DRAM. The *memory timestamps* (*mwts* and *mrts*) are used to maintain coherence for DRAM data. They indicate the maximal write and read timestamps of all the cachelines stored in main memory but not cached. For each cacheline evicted from the LLC, *mwts* and *mrts* are updated to $Max(wts, mwts)$ and $Max(rts, mrts)$, respectively. When a cacheline is loaded from main memory, its *wts* and *rts* are assigned to be *mwts* and *mrts*, respectively. This guarantees that previous accesses to the same memory location are ordered before the accesses to the newly loaded cacheline. This takes care of the case when a cacheline is evicted from the LLC but is still cached in some core's private cache. Note that multiple *mwts* and *mrts* can be used for different partitions of data. In this thesis, we assign a pair *mwts* and *mrts* for each LLC cache bank.

Flush or write back response (column 5): Finally, the flush response and write back response

Table 3.4: **Network Messages** – The check marks indicate what components a message contains.

Message Type	*pts*	*rts*	*wts*	data
SH_REQ	√		√	
EX_REQ			√	
FLUSH_REQ				
WB_REQ		√		
SH_REP		√	√	√
EX_REP		√	√	√
UPGRADE_REP		√		
RENEW_REP		√		
FLUSH_REP		√	√	√
WB_REP		√	√	√
DRAM_ST_REQ				√
DRAM_LD_REQ				
DRAM_LD_REP				√

are handled in the same way as in directory-based protocols. Note that when a cacheline is exclusively owned by a core, only the owner has the latest *rts* and *wts*; the *rts* and *wts* in the timestamp manager are invalid and the same hardware bits can be used to store the ID of the owner core.

An Example Program

We use an example to show how Tardis works with a parallel program. Figure 3-1 shows how the simple program in Listing 3.1 runs with the Tardis protocol. In the example, we assume a lease of 10 and that both instructions from Core 0 are executed before the instructions in Core 1.

Listing 3.1: Example Program

```
          initially A = B = 0
[Core 0]                [Core 1]
 A = 1                   B = 1
 print B                 print A
```

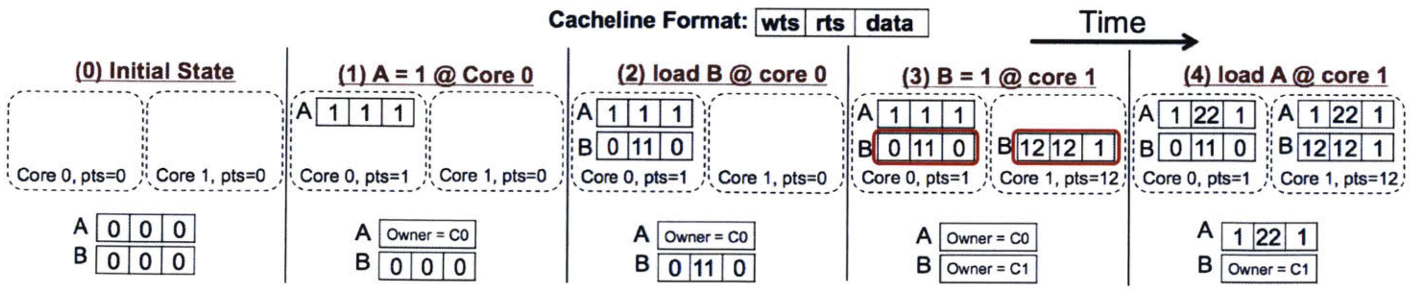


Figure 3-1: **An Example of Tardis** – An example program running with Tardis (*lease*= 10). Cachelines in private caches and LLC are shown. The cacheline format is shown at the top of the figure.

Step 1 : The store to A misses in Core 0's private cache and an EX_REQ is sent to the timestamp manager. The store operation happens at $pts = Max(pts, A.rts + 1) = 1$ and the A.rts and A.wts in the private cache both become 1. The timestamp manager marks A as exclusively owned by Core 0.

Step 2 : The load of B misses in Core 0's private cache. After Step 1, Core 0's pts becomes 1. So the end of the lease should be $Max(rts, wts + lease, pts + lease) = 11$.

Step 3 : The store to B misses in Core 1's private cache. At the timestamp manager, the exclusive ownership of B is immediately given to Core 1 at $pts = rts + 1 = 12$. Note that two different versions of B exist in the private caches of core 0 and core 1 (marked in red circles). In core 0, $B = 0$ but is valid when $0 \leq timestamp \leq 11$; in Core 1, B = 1 and is only valid when $timestamp \geq 12$. This does not violate sequential consistency since the loads of B at core 0 will be ordered logically before the loads of B at core 1, even if they happen in the opposite order with respect to the physical time.

Step 4 : Finally the load of A misses in Core 1's private cache. The timestamp manager sends a WB_REQ to the owner (Core 0) which updates its own timestamps and writes back the data. Both cores will have the same data with the same range of valid timestamps on cacheline A.

With Tardis on sequential consistency, it is impossible for the example program above to output 0 for both print commands, even if the cores are out-of-order. The reason will be discussed in Section 3.4.3.

3.4.3 Out-of-Order Execution

So far we have assumed in-order cores, i.e., a second instruction is executed only after the first instruction commits and updates the pts. For out-of-order cores, a memory instruction can start execution before previous instructions finish and thus the current pts is not known. However, with sequential consistency, all instructions must commit in the program order. Tardis therefore enforces timestamp order at the commit time.

Timestamp Checking

In the re-order buffer (ROB) of an out-of-order core, instructions commit in order. We slightly change the meaning of pts to mean the timestamp of the last *committed* instruction. For sequential consistency, pts still increases monotonically. Before committing an instruction, the timestamps are checked. Specifically, the following cases may happen for shared and exclusive cachelines, respectively.

An exclusive cacheline can be accessed by both load and store requests. And the accessing instruction can always commit with $pts \Leftarrow Max(pts, wts)$ for a load operation and $pts \Leftarrow Max(pts, rts + 1)$ for a store operation.

A shared cacheline can be accessed by load requests. And there are two possible cases.

1. $pts \leq rts$. The instruction commits. $pts \Leftarrow Max(wts, pts)$.

2. $pts > rts$. The instruction aborts and is restarted with the latest pts. Re-execution will trigger a renew request.

There are two possible outcomes of a restarted load. If the cacheline is successfully renewed, the contents of the cacheline do not change. Otherwise, the load returns a different version of data and all the depending instructions in the ROB need to abort and be restarted. This hurts performance and wastes energy. However, the same flushing operation is also required for an OoO core on a baseline directory protocol under the same scenario [55]. A cacheline can be loaded by a core but invalidated due to another core's store before the load instruction commits. In this case, the load needs to be restarted and the ROB flushed. Therefore, the renewal failure in Tardis serves as similar functionality to an invalidation in directory protocols.

Out-of-Order Example

If the example program in Section 3.4.2 runs on an out-of-order core, both loads may be scheduled before the corresponding stores. In this section, we show how this scenario can be detected by the timestamp checking at commit time and therefore the two prints never both output 0. For the program to output $A = B = 0$, both loads are executed before the corresponding stores in the timestamp order.

$$L(A) <_{ts} S(A), \quad L(B) <_{ts} S(B)$$

For the instruction sequence to pass the timestamp checking, we have

$$S(A) \leq_{ts} L(B), \quad S(B) \leq_{ts} L(A)$$

Putting them together leads to the following contradiction.

$$L(A) <_{ts} S(A) \leq_{ts} L(B) <_{ts} S(B) \leq_{ts} L(A)$$

This means that at least at one core, the timestamp checking will fail. The load at that core is restarted with the updated *pts*. The restarted load will not be able to renew the lease but will return the latest value (i.e., 1). So at least at one core, the output value is 1.

3.4.4 Avoiding Livelock

Although Tardis strictly follows sequential consistency, the protocol discussed so far may generate livelocks due to deferred update propagation. In directory coherence, a write is quickly observed by all the cores through the invalidation mechanism. In Tardis, however, a core may still read the old cached data even if another core has updated it, as long as the cacheline has not expired in logical time. It is the responsibility of the core caching the data to detect staleness. Therefore, there is a delay in physical time between the data update at the writing and reading core's caches. In the worst case when deferment becomes indefinite, livelock occurs. For example, if a core spins on a variable which is set by another core, the *pts* of the spinning core does not increase and thus the old data never expires. As a result, the core may spin forever without observing the updated data.

Tardis uses a simple solution to handle this livelock problem. To guarantee forward progress, we only need to make sure that an update is *eventually* observed by following loads, that is, the update becomes globally visible within some finite physical time. This is achieved by occasionally incrementing the *pts* in each core so that the old data in the private cache eventually expires and the latest update becomes visible. The self increment can be periodic or based on more intelligent heuristics. We restrict ourselves to periodic increments in this chapter, and explore optimizations in Chapter 4.

3.4.5 Tardis vs. Directory Coherence

In this section, we qualitatively compare Tardis to the directory coherence protocol in terms of network messages, scalability and complexity.

Protocol Messages

In Table 3.2 and Table 3.3, the advantages and disadvantages of Tardis compared to directory protocols are shaded in light green and light red, respectively. Both schemes have similar behavior and performance in the other state transitions (the white cells).

Invalidation: In a directory protocol, when the directory receives an exclusive request to a *Shared* cacheline, the directory sends invalidations to all the cores sharing the cacheline and waits for acknowledgements. This usually incurs significant latency which may hurt performance. In Tardis, however, no invalidation happens (cf. Section 3.4.2) and the exclusive ownership can be immediately returned without waiting. The timestamps guarantee that sequential consistency is maintained. The elimination of invalidations makes Tardis much simpler to implement and reason about.

Eviction: In a directory protocol, when a shared cacheline is evicted from the private cache, a message is sent to the directory where the sharer information is maintained. Similarly, when a shared cacheline is evicted from the LLC, all the copies in the private caches should be invalidated. In Tardis, correctness does not require maintaining sharer information and thus no such invalidations are required. When a cacheline is evicted from the LLC, the copies in the private caches can still exist and be accessed.

Data Renewal: In directory coherence, a load hit only requires the cacheline to exist in the private cache. In Tardis, however, a cacheline in the private cache may have expired and cannot be accessed. In this case, a renew request is sent to the timestamp manager which incurs extra latency and network traffic. In Section 3.5.1, we present techniques to reduce the overhead of data renewal.

Scalability

A key advantage of Tardis over directory coherence protocols is scalability. Tardis only requires the storage of timestamps for each cacheline and the owner ID for each LLC cacheline ($O(\log N)$, where N is the number of cores). In practice, the same hardware bits can be used for both timestamps and owner ID in the LLC; because when the owner ID needs to be stored, the cacheline is exclusively owned and the timestamp manager does not maintain the timestamps.

On the contrary, a full-map directory-based coherence protocol maintains the list of cores sharing a cacheline which requires $O(N)$ storage overhead. Previous works proposed techniques to improve the scalability of directory protocols by introducing broadcast or other complexity (cf. Section 3.8.2).

Simplicity

Another advantage of Tardis is its conceptual simplicity. Tardis is directly derived from the definition of sequential consistency and the timestamps explicitly express the global memory order. This makes it easier to reason the correctness of the protocol. Concretely, given that Tardis does not have to multicast/broadcast invalidations and collect acknowledgements, the number of transient states in Tardis is smaller than that of a directory-based protocol.

3.5 Optimizations and Extensions

We introduce several optimizations in the Tardis protocol in this section. All of them are enabled during our evaluation. A few extensions are also discussed.

3.5.1 Speculative Execution

As discussed in Section 3.4.5, the main disadvantage of Tardis compared to a directory-based coherence protocol is the renew request. In a pathological case, the *pts* of a core may rapidly increase since some cachelines are frequently read-write shared by different cores. Meanwhile, the read-only cachelines will frequently expire and a large number of renew requests are generated incurring both latency and network traffic. Observe, however, that most renew requests will successfully extend the lease and the renew response does not transfer the data. This indicates that the renewal traffic is not significant. More important, this means that the data in the expired cacheline is actually valid and we could have used the value without stalling the pipeline of the core. Based on this observation, we propose to use speculation to hide renew latency. When a core reads a cacheline which has expired in the private cache, instead of stalling and waiting for the renew response, the core reads the current value and continues executing speculatively. If the renewal fails and the latest cacheline is returned, the core rolls back by discarding the speculative computation that depends on the load. The rollback process is similar to a branch misprediction which has already been implemented in most processors.

For processors that can hide the renew latency by allowing multiple outstanding misses, successful renewals (which is the common case) do not hurt performance. Speculation failure does incur performance overhead since we have to rollback and rerun the instructions. However, if the same instruction sequence is executed in a directory protocol, the expired cacheline is not in the private cache in the first place; the update from another core has already invalidated this cacheline and a cache miss happens. As a result, in both Tardis and directory coherence, the value of the load is returned at the same time incurring the same latency and network traffic. Tardis still has some extra overhead as it needs to discard the speculated computation, but this overhead is relatively small.

Speculation successfully hides renew latency in most cases. The renew messages, however, may still increase the on-chip network traffic. This is especially problematic if the private caches have a large number of *shared* cachelines that all expire when the *pts* jumps ahead due to a write or self increment. This is a fundamental disadvantage of Tardis compared to directory coherence protocols. According to our evaluations in Section 3.7, however, Tardis has good performance and acceptable network traffic on real benchmarks even with this disadvantage. In Chapter 4, we will discuss relaxed memory consistency models and dynamic leases which are solutions to mitigate the renewal overhead.

3.5.2 Timestamp Compression

In Tardis, all the timestamps increase monotonically and may roll over. One simple solution is to use 64-bit timestamps which never roll over in practice. This requires 128 extra bits to be stored per cacheline, which is a significant overhead. Observe, however, that the high-order bits in a timestamp change infrequently and are usually the same across most of the timestamps. We exploit this observation and propose to compress this redundant information using a *base-delta* compression scheme.

In each cache, a *base timestamp* (*bts*) stores the common high-order bits of *wts* and *rts*. In each cacheline, only the *delta timestamps* (*delta_ts*) are stored (*delta_wts* = *wts* − *bts* and *delta_rts* = *rts* − *bts*). The actual timestamp is the sum of the *bts* and the corresponding *delta_ts*. The *bts* is 64 bits to prevent rollover. The storage requirement for *bts* is small since only one *bts* is required per cache. The per cacheline *delta_ts* is much shorter with lower storage overhead.

When any *delta_ts* in the cache rolls over, we will rebase where the local *bts* is increased and all the *delta_ts* in the cache are decreased by the same amount, e.g., half of the maximum *delta_ts*. For simplicity, we assume that the cache does not serve any request during the rebase operation.

Note that increasing the *bts* in a cache may end up with some *delta_ts* being negative. In this case, we just set the *delta_ts* to 0. This effectively increases the *wts* and *rts* in the cacheline. However, it does not violate the consistency model. Consider a shared LLC cacheline or an exclusive private cacheline – the *wts* and *rts* can be safely increased. Increasing the *wts* corresponds to writing the same data to the line at a later logical time, and increasing the *rts* corresponds to a hypothetical read at a later logical time. Neither operation violates the rules of sequential consistency. Similarly, for a shared cacheline in the private cache, *wts* can be safely increased as long as it is smaller than *rts*. However, *rts* cannot be increased without coordinating with the timestamp manager. So if *delta_rts* goes negative in a shared line in a private cache, we simply invalidate the line from the cache. The last possible case is an exclusive cacheline in the LLC. No special operation is required since the timestamp manager has neither the timestamps nor the data in this case.

The key advantage of this base-delta compression scheme is that all computation is local to each cache without coordination between different caches. This makes the scheme very scalable.

The scheme discussed here does not compress the timestamps over the network and we assume that the network messages still use 64-bit timestamps. It is possible to reduce this overhead by extending the base-delta scheme over the network but this requires global coordination amongst multiple caches. We did not implement this extension in order to keep the basic protocol simple and straightforward.

3.5.3 Private Write

According to Table 3.2, writing to a cacheline in exclusive state updates both *wts* and *pts* to $Max(pts, rts+1)$. In this case, if the core keeps writing to the same address, *pts* keeps increasing causing other cachelines to expire. If the updated cacheline is completely private to the updating thread, however, there is actually no need to increase the timestamps in order to achieve sequential consistency. Since the global memory order in Tardis is the physiological time order (cf. Section 3.3), we can use physical time to order these operations implicitly without increasing the logical time.

Specifically, when a core writes to a cacheline, the *modified bit* will be set. For future writes to the same cacheline, if the bit is set, then the *pts*, *wts* and *rts* are just set to $Max(pts, rts)$. This means the write may have the same timestamp as previous writes or reads to the same cacheline. But since they are performed by the same core, the write must happen after the read in physical time order, and therefore in physiological time order as well. This means that *pts* will not increase if the line is repeatedly written to. The optimization will significantly reduce the rate at which timestamps increase if most of the accesses from a core are to thread private data.

3.5.4 Extension: Exclusive and Owned States

The Tardis protocol presented so far assumes three states for each cacheline: **M**odified, **S**hared and **I**nvalid (**MSI**). In practice, invalidation-based protocols are usually optimized by adding **E**xclusive and/or **O**wned states. The resulting protocols are **MESI**, **MOSI** and **MOESI**. In this section, we show how *E* and *O* states are implemented in Tardis.

Similar to the *M* state, the *E* state allows the cacheline to be exclusively cached and be modified without contacting the timestamp manager. Different from the *M* state, however, the *E* state can be acquired upon a shared request if the requesting core seems to be the only sharer of the line. This can improve performance for the case where a write follows a read to the same private data. Without the *E* state, the write needs to upgrade the cacheline's state from *S* to *M* by sending an extra message to the timestamp manager. The message is not required if the *E* state is supported.

In the timestamp manager, cachelines in *M* and *E* states are handled in the same way. One difference that makes it a little tricky to implement the *E* state in Tardis is to tell if the requesting core is the only sharer. This is tricky since Tardis does not maintain the sharer list as in a directory-based protocol. To solve this problem, Tardis adds an extra bit for each cacheline indicating whether any core has accessed it since it was put into the LLC (i.e., loaded from main memory, or written back by a previous owner). And if the bit is unset, the requesting core gets the line in exclusive state, else in shared state. Note that with this scheme, an exclusive response means the line is likely to have only one sharer, but it does not guarantee it. However, even if other cores are sharing the line, it can still be returned to the requester in exclusive state without affecting correctness. This is because the same cacheline can be shared and exclusively owned by different cores at the same physical time, as long as the lines have non-overlapping logical leases.

The *O* state allows a cacheline to be dirty but shared in the private caches. Upon receiving the WB_REQ request, instead of writing the data back to the LLC or DRAM, the core can change the cacheline to *O* state and directly forward the data to the requesting core. In Tardis, the *O* state can also be supported by keeping track of the owner at the timestamp manager. SH_REQs to the timestamp manager are forwarded to the owner which does cache-to-cache data transfers. Similar to the basic Tardis protocol, when the owner is evicted from the private cache, the cacheline is written back to the LLC or the DRAM and its state in the timestamp manager is changed to Shared or Invalid.

3.5.5 Extension: Remote Word Access

Traditionally, a core loads a cacheline into the private cache before accessing the data. But it can access data remotely without caching it. Remote word access has been studied in the context of locality-aware directory coherence [88]. Remote atomic operation has been implemented on Tilera processors [38, 73]. Allowing data accesses or computations to happen remotely can reduce the coherence messages and thus improve performance [157].

However, it is not easy to maintain the performance gain of these remote operations with directory coherence under TSO or sequential consistency. For a remote load operation (which might be part of a remote atomic operation), it is not very easy to determine its global memory order since it is hard to know the physical time when the load operation actually happens. As a result, integration of remote access with directory coherence is possible but fairly involved [87].

Consider the example program in Listing 3.1 assuming all memory requests are remote accesses. If all requests are issued simultaneously, then both loads may be executed before both stores and the program outputs 0 for both prints. It is not easy to detect this violation in a directory protocol since we do not know when each memory operation happens. As a result, either the remote accesses are sequentially issued or additional mechanisms need to be added [87].

In Tardis, however, memory operations are ordered through timestamps. It is very easy to determine the memory order for remote accesses since it is determined by the timestamp of the operation. In Tardis, multiple remote accesses can be issued in parallel and the order can be checked after they return. If any load violates the memory order, it can be reissued with the updated timestamp information (similar to timestamp checking in an out-of-order core).

3.5.6 Other Extensions

Atomic operations in Tardis can be implemented the same way as in directory protocols. Tardis can be extended to relaxed consistency models such as Total Store Order (TSO) which is implemented in Intel x86 processors [129]. A more thorough discussion of relaxed memory models is presented in Chapter 4.

3.6 Case Study

In this section, we use an example parallel program as a case study to compare Tardis with an full-map MSI directory protocol.

3.6.1 Example

Listing 3.2 shows the parallel program we use for the case study. In this program, the two cores issue loads and stores to addresses `A` and `B`. The `nop` in Core 1 means that the core spends that cycle without accessing the memory subsystem. The program we use here is a contrived example to highlight the difference between Tardis and a directory coherence protocol.

Listing 3.2: The case study parallel program

```
[Core 0]        [Core 1]
 L(B)            nop
 A = 1           B = 2
 L(A)            L(A)
 L(B)            B = 4
 A = 3
```

Figure 3-2 and Fig. 3-3 show the execution of the program on a directory-based and the Tardis coherence protocols, respectively. A cacheline is either in shared (*S*) or modified (*M*) state. For Tardis, a static lease of 10 is used. Initially, all private caches are empty and all timestamps are 0. We will explain step by step how Tardis executes the program and highlight the differences between the two protocols.

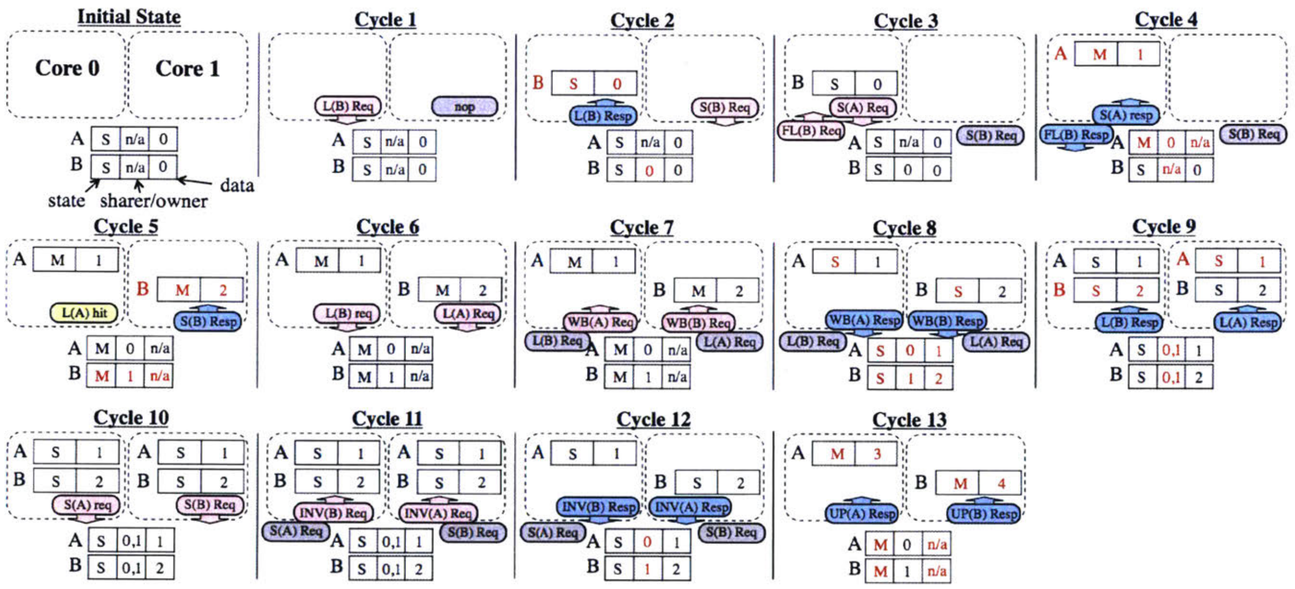


Figure 3-2: **An Example of a Directory-Based Coherence Protocol** – Execution of the case study program with a directory-based coherence protocol. *FL*, *WB*, *INV* and *UP* stand for Flush, Writeback, Invalidate, and Update, respectively.

Cycle 1 and 2: Core 0 sends a shared request to address *B* in cycle 1, and receives the response in cycle 2. The cacheline is reserved till timestamp 10. Core 1 sends an exclusive request to address *B* at cycle 2. In these two cycles, both the directory protocol and Tardis have the same network messages sent and received.

Cycle 3: In Tardis, the exclusive request from core 1 sees that address *B* is shared till timestamp 10. The exclusive ownership is instantly returned and the store is performed at timestamp 11. In

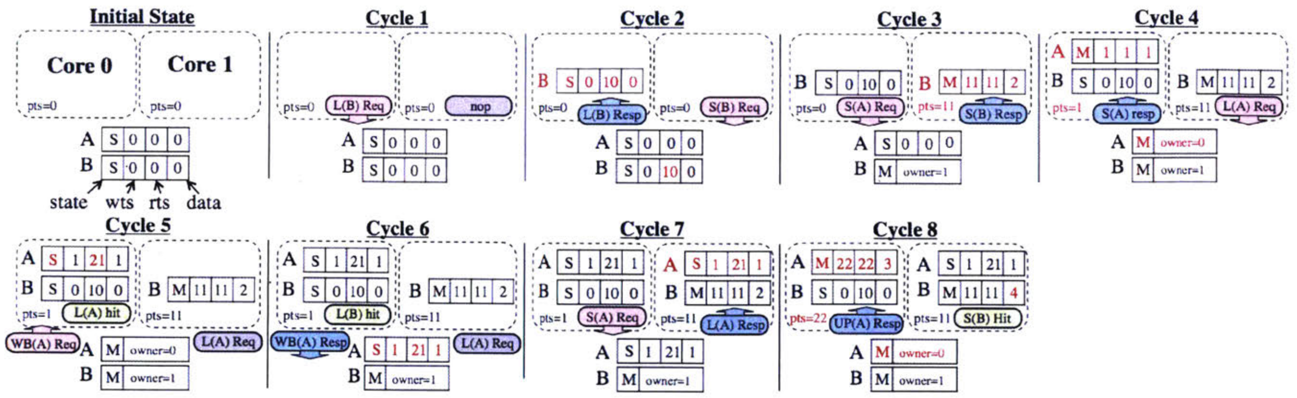


Figure 3-3: **An Example of Tardis Protocol** – Execution of the case study program with Tardis.

the directory protocol, however, an invalidation must be sent to Core 0 to invalidate address *B*. As a result, the exclusive response is delayed to cycle 5. At this cycle, core 0 sends an exclusive request to address *A*.

Cycle 4: In both Tardis and the directory protocol, address *A*'s exclusive ownership can be instantly returned to core 0 since no core is sharing it. The *pts* of core 0 becomes 1 after performing the store. Core 1 performs a shared request to address *A* which needs to get the latest data from core 0 through write back. So the shared response returns in cycle 7. The same *L(A)* instruction in the directory protocol incurs the same latency and network traffic from cycle 6 to 9.

Cycle 5 and 6: In cycle 5, the *L(A)* instruction in core 0 hits in the private cache and thus no request is sent. Also in core 0, the write back request increases address *A*'s *rts* to 21 since the requester's (core 1) *pts* is 11 and the lease is 10. In cycle 6, the *L(B)* instruction in core 0 hits in the private cache since the *pts* is 1 and the cached address *B* is valid till timestamp 10. In the directory protocol, the same *L(B)* instruction is also issued at cycle 6. However, it misses in the private cache since the cacheline was already invalidated by core 1 at cycle 4. So a write back request to core 1 needs to be sent and the shared response returns at cycle 9.

Cycle 7 and 8: At cycle 7, core 0 sends an exclusive request to address *A* and core 1 gets the shared response to address *A*. At cycle 8, the exclusive ownership of address *A* is instantly returned to core 0 and the store happens at timestamp 22 (because address *A* has been reserved for reading until timestamp 21). In the directory protocol, the same *S(A)* instruction happens at cycle 10 and the shared copy in core 1 must be invalidated before exclusive ownership is given. Therefore, the exclusive response is returned at cycle 13. Also in cycle 8 in Tardis, core 1 stores to address *B*. The store hits in the private cache. In the directory protocol, the same store instruction happens at cycle

10. Since core 0 has a shared copy of address *B*, an invalidation must be sent and the exclusive response is returned at cycle 13.

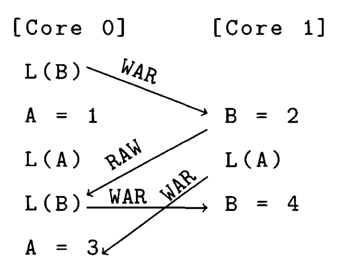
3.6.2 Discussion

In this case study, the cycle saving of Tardis mainly comes from the removal of invalidations. While a directory protocol requires that only one version of an address exist at any point in time across all caches, Tardis allows multiple versions to coexist as long as they are accessed at different timestamps.

The *pts* in each core shows how Tardis orders the memory operations. At cycle 3, core 1's *pts* jumps to 11. Later at cycle 4, core 0's *pts* jumps to 1. Although the operation from core 0 happens later than the operation from core 1 in physical time, it is the opposite in global memory and physiological time order. Later at cycle 8, core 0's *pts* jumps to 22 and becomes bigger than core 1's *pts*.

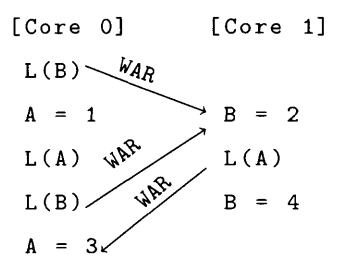
In Tardis, a load may still return a old version of a memory location after the memory location is updated by a different core, as long as sequential consistency is not violated. As a result, Tardis may produce a different instruction interleaving than a directory-based protocol. Listings 3.3 and 3.4 show the instruction interleaving of the directory-based protocol and Tardis, respectively, on our example program.

Listing 3.3: Instruction interleaving in the directory-based protocol



```
[Core 0]      [Core 1]
 L(B)
 A = 1         B = 2
 L(A)          L(A)
 L(B)          B = 4
 A = 3
```

Listing 3.4: Instruction interleaving in Tardis



```
[Core 0]      [Core 1]
 L(B)
 A = 1         B = 2
 L(A)          L(A)
 L(B)          B = 4
 A = 3
```

In the directory protocol, the second *L(B)* instruction from core 0 is ordered between the two stores to address *B* from core 1 in the global memory order. In Tardis, however, the same *L(B)* instruction is ordered before both stores. Such reordering is possible because Tardis enforces sequential consistency in physiological time order, which can be different from the physical time order.

3.7 Evaluation

We now evaluate the performance of Tardis in the context of multicore processors.

3.7.1 Methodology

We use the Graphite [110] multicore simulator for our experiments. The default hardware parameters are listed in Table 3.5. The full-map directory-based protocol with *MSI* is used as the baseline in this section.[3] This baseline keeps the full sharer information for each cacheline and thus incurs non-scalable storage overhead. To model a more scalable protocol, we use the Ackwise [89] protocol which keeps a limited number of sharers and broadcasts invalidations to all cores when the number of sharers exceeds the limit.

Table 3.5: **System Configuration**.

System Configuration	
Number of Cores	N = 64 @ 1 GHz
Core Model	In-order, Single-issue
Memory Subsystem	
Cacheline Size	64 bytes
L1 I Cache	16 KB, 4-way
L1 D Cache	32 KB, 4-way
Shared L2 Cache per Core	256 KB, 8-way
DRAM Bandwidth	8 MCs, 10 GB/s per MC
DRAM Latency	100 ns
2-D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link)
Flit Width	128 bits
Tardis Parameters	
Lease	10
Self Increment Period	100 cache accesses
Delta Timestamp Size	20 bits
L1 Rebase Overhead	128 ns
L2 Rebase Overhead	1024 ns

In our simulation mode, Graphite includes functional correctness checks, where the simulation fails, e.g., if wrong values are read. All the benchmarks we evaluated in this section completed, which corresponds to a level of validation of Tardis and its Graphite implementation. A formal verification of Tardis will be presented in Chapter 5.

Splash-2 [153] benchmarks are used for performance evaluation. For most experiments, we report both the throughput (in bars) and network traffic (in red dots).

[3] Other states, e.g., **O** (Owner) and **E** (Exclusive) can be added to an MSI protocol to improve performance; such states can be added to Tardis as well to improve performance as described in Section 3.5.4.

Tardis Configurations

Table 3.5 also shows the default Tardis configuration. For a load request, a static lease of 10 is taken. The *pts* at each core self increments by one if it has not changed for 100 cache accesses (self increment period). The Base-delta compression scheme is applied with 20-bit delta timestamps and 64-bit base timestamps. When the timestamp rolls over, the rebase overhead is 128 *ns* in L1 and 1024 *ns* in an LLC slice.

The lease and self increment period are both static in the evaluation here. Both parameters can be changed dynamically for better performance based on the data access pattern. Exploration of such techniques will be discussed in Chapter 4.

3.7.2 Main Results

Throughput

Figure 3-4 shows the throughput of Ackwise and Tardis on 64 in-order cores, normalized to baseline MSI. For Tardis, we also show the performance with speculation turned off. For most benchmarks, Tardis achieves similar performance compared to the directory baselines. On average, the performance of Tardis is within 0.5% of the baseline MSI and Ackwise.

Figure 3-4: **Performance of Tardis at 64 Cores** – Both the throughput (bars) and the network traffic (dots) are normalized to the baseline full-map directory-based protocol.

If speculation is turned off, the average performance of Tardis becomes 7% worse than the directory-based baseline. In this case, the core stalls while waiting for the renewal, in contrast to the default Tardis where the core reads the value speculatively and continues execution. Since most renewals are successful, speculation hides a significant amount of latency and makes a big difference in performance.

Network Traffic

The red dots in Figure 3-4 show the network traffic of Ackwise and Tardis normalized to the baseline-directory protocol. On average, Tardis with and without speculation incurs 19.4% and 21.2% more network traffic. Most of this traffic comes from renewals. Figure 3-5 shows the percentage of renew requests and misspeculations out of all LLC accesses. Note that the y-axis is in log scale.

In benchmarks with lots of synchronizations (e.g., `CHOLESKY`, `VOLREND`), cachelines in the private cache frequently expire generating a large number of renewals. In `VOLREND`, for example, 65.8% of LLC requests are renew requests which is 2× of normal LLC requests. As discussed in Section 3.4.5, a successful renewal only requires a single flit message, which is cheaper than a normal LLC access. So the relative network traffic overhead is small (36.9% in `VOLREND` compared to the directory-based baseline).

An outlier is `WATER-SP`, where Tardis increases the network traffic by 3×. This benchmark has very low L1 miss rate and thus very low network utilization to begin with. But Tardis increments the *pts* of each core periodically, which incurs a significant amount of renew requests due to cacheline expiration. But the absolute amount of traffic is still very small even though Tardis incurs 3× more traffic.

In many other benchmarks (e.g., `BARNES`, `WATER-NSQ`, etc.), Tardis has less network traffic than baseline MSI. The traffic reduction comes from the elimination of invalidation and cache eviction traffic.

From Figure 3-5, we see that the misspeculation rate for Tardis is very low; less than 1% renewals failed on average. A speculative load is considered a miss if the renew fails and a new version of data is returned. Having a low misspeculation rate indicates that the vast majority of renewals are successful.

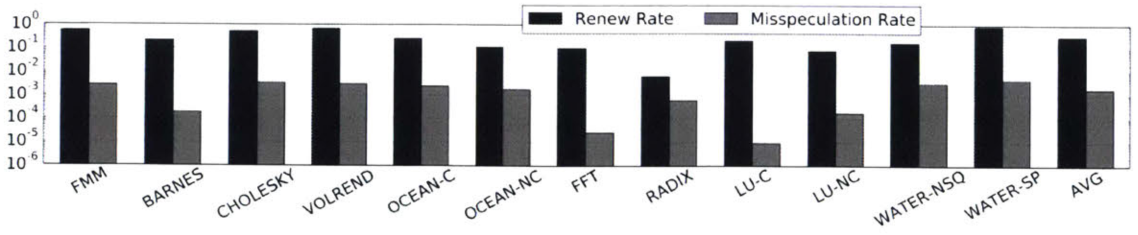


Figure 3-5: **Renew and Misspeculation Rate in Tardis** – Fraction of renewals and misspeculations out of all L1 data cache misses. The processor has 64 cores. The Y-axis is in log scale.

Table 3.6: **Timestamp Statistics**

Benchmarks	Ts. Incr. Rate (cycle / timestamp)	Self Incr. Perc.
FMM	322	22.5%
BARNES	155	33.7%
CHOLESKY	146	35.6%
VOLREND	121	23.6%
OCEAN-C	81	7.0%
OCEAN-NC	85	5.6%
FFT	699	88.5%
RADIX	639	59.3%
LU-C	422	1.4%
LU-NC	61	0.1%
WATER-NSQ	73	12.8%
WATER-SP	363	29.1%
AVG	263	26.6%

Timestamp Discussion

Table 3.6 shows how fast the *pts* in a core increases, in terms of the average number of cycles to increase the *pts* by 1. Table 3.6 also shows the percentage of *pts* increasing caused by self increment (cf. Section 3.4.4).

Over all the benchmarks, *pts* is incremented by 1 every 263 cycles, at each core. For a delta timestamp size of 20 bits, the timestamp rolls over every 0.28 seconds. In comparison, the rebase overhead (128 ns in L1 and 1 μs in L2) becomes negligible. This result also indicates that timestamps in Tardis increase very slowly. This is because they can only be increased by accessing shared read/write cachelines or self increment.

On average, 26.6% of timestamp increasing is caused by self increment and the percentage can be as high as 88.5% (FFT). This has negative impact on performance and network traffic since unnecessarily increasing timestamps causes increased expiration and renewals. Better livelock avoidance algorithms can resolve this issue, as will be discussed in Chapter 4.

3.7.3 Sensitivity Study

L1 Data Cache Size

We first study how sensitive Tardis is with respect to the private cache size.

Figure 3-6 shows the performance of different cache coherence protocols as we sweep the L1 data cache size from 32 KB to 512 KB, for a subset of Splash-2 benchmarks. We observe that both Tardis and the full-map directory-based protocol achieve better performance as the L1 data cache size increases. Without the speculative read optimization (cf. Section 3.5.1), the performance of Tardis is worse than the directory-based protocol; the performance gap widens as the L1 cache size increases. This is because Tardis incurs more cache misses than the directory-based protocol due to lease expirations; and the number of expiration-induced misses increases with the private cache size. This is because a larger private cache contains more cold cachelines, which are more likely to contain stale leases. Without the optimization of speculative reads, these extra misses degrade Tardis' performance. With the optimization, however, the performance of Tardis matches that of the directory-based protocol.

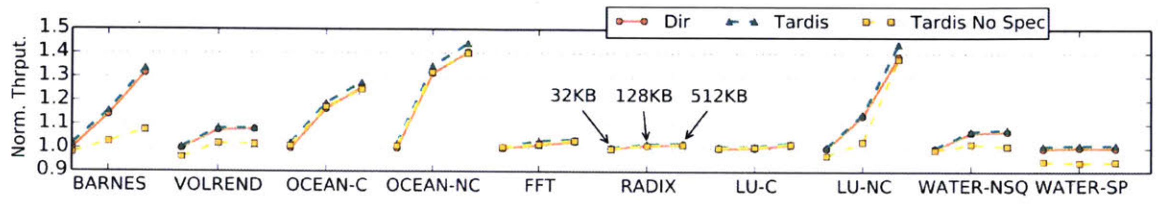


Figure 3-6: **Sensitivity of throughput to L1 Data Cache Size** – Performance of a full-map directory-based protocol and Tardis (with and without the optimization of speculative reads) on 64 cores, with different L1 data cache sizes. All results are normalized to the full-map directory-based protocol with 32 KB L1 data caches.

Figure 3-7 shows the network traffic breakdown for the same sets of experiments. For each benchmark, we report traffic breakdown for both directory-based and Tardis (with and without speculation) protocols for two difference L1 data cache sizes (i.e., 32 KB and 512 KB). For all the benchmarks we evaluated, having a larger L1 data cache reduces the overall network traffic. For Tardis, we observe that the amount of renewal traffic increases as the L1 data cache gets larger, especially for BARNES, VOLREND and WATER-NSQ. As explained above, this is because that a larger private cache leads to more cacheline expirations and thereby more renewals.

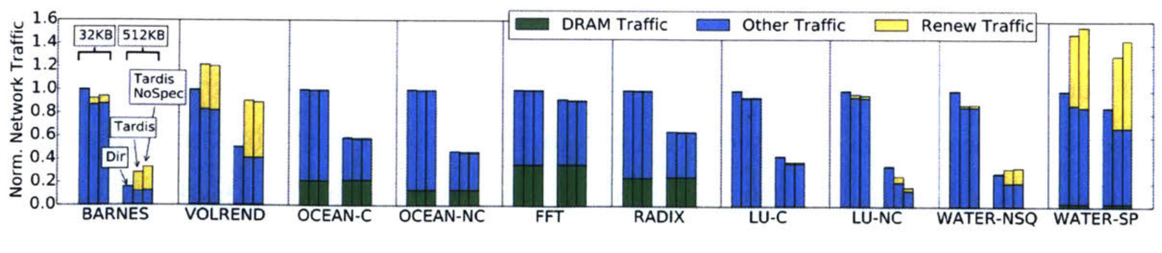


Figure 3-7: **Sensitivity of network traffic to L1 Data Cache Size** – Network traffic breakdown of a full-map directory-based protocol and Tardis (with and without the optimization of speculative reads) on 64 cores, with two different L1 data cache sizes (32 KB and 512 KB). All results are normalized to the full-map directory-based protocol with 32 KB L1 data caches.

In-order vs. Out-of-Order Core

Figure 3-8 shows the performance of Tardis on out-of-order cores. Compared to in-order cores (cf. Figure 3-4), the performance impact of speculation is much smaller. When a renew request is outstanding, an out-of-order core is able to execute independent instructions even if it does not speculate. As a result, the renewal's latency can be hidden. On average, Tardis with and without speculation is 0.2% and 1.2% within the performance of the directory-based protocol, respectively.

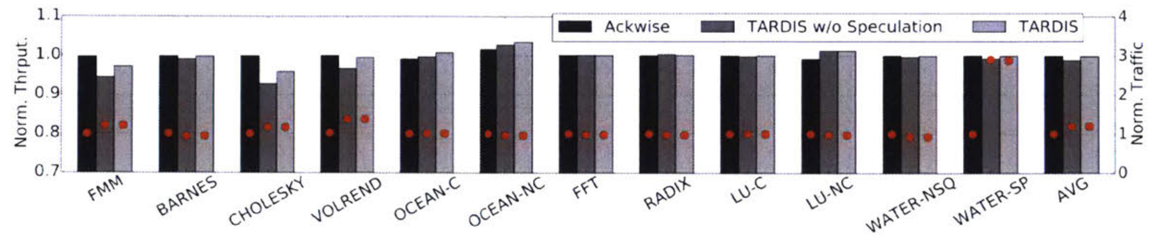


Figure 3-8: **Out-of-Order Core Experiments** – Performance of Tardis on 64 out-of-order cores.

The normalized traffic of Tardis on out-of-order cores is not much different from in-order cores. This is because both core models follow sequential consistency and the timestamps assigned to the memory operations are virtually identical. As a result, the same amount of renewals is generated.

Self Increment Period

As discussed in Section 3.4.4, we periodically increment the *pts* at each core to avoid livelock. The *self increment period* specifies the number of data cache accesses before self incrementing the *pts* by one. If the period is too small, the *pts* increases too fast causing more expirations; more renewals will be generated which increases network traffic and hurts performance. A fast growing *pts* leads to more frequent lease expirations, which also hurts performance. If the period is too large, however, an update at a core may not be observed by another core quickly enough, which degrades performance.

Figure 3-9 shows the performance of Tardis with different self increment period. The performance of most benchmarks is not sensitive to this parameter. In `FMM` and `CHOLESKY`, performance goes down when the period is 1000. This is because these two benchmarks heavily use spinning (busy waiting) to synchronize between threads. If the period is too large, each core spends a long time spinning on the stale value in the private cache and cannot make forward progress.

Having a larger self increment period always reduces the total network traffic because of fewer renewals. Given the same performance, a larger period should be preferred due to network traffic

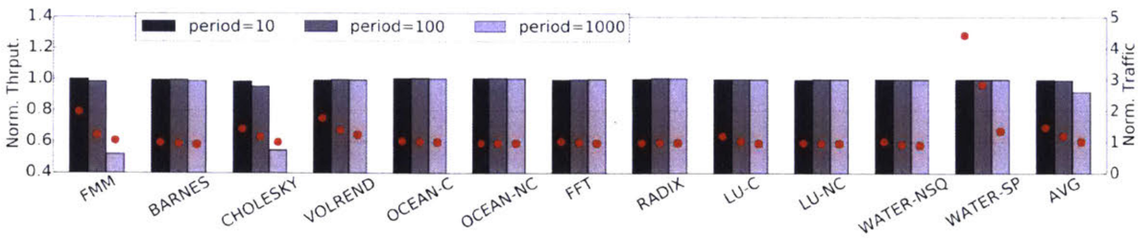


Figure 3-9: **Tardis' Performance Sensitivity to Self Increment Period.**

reduction. Our default self increment period is 100 which has reasonable performance and network traffic.

Scalability

Figure 3-10 shows the performance of Tardis on 16 and 256 cores respectively.

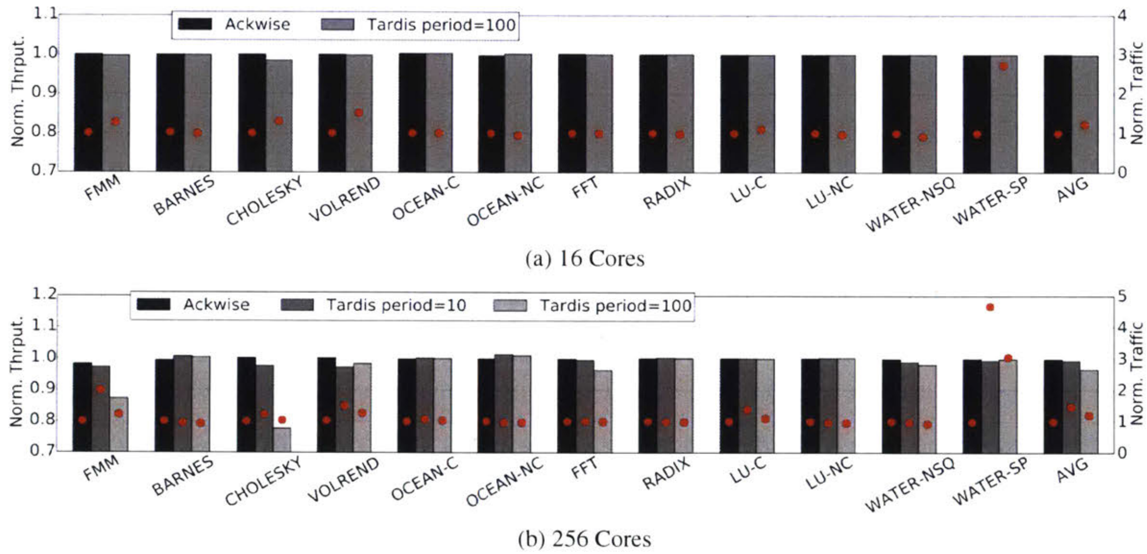


Figure 3-10: **Scalability Study** – Performance of Tardis on 16 and 256 cores.

At 16 cores, the same configurations are used as at 64 cores. On average, the throughput is within 0.2% of baseline MSI and the network traffic is 22.4% more than the baseline MSI.

At 256 cores, two Tardis configurations are shown with self increment period 10 and 100. For most benchmarks, both Tardis configurations achieve similar performance. For `FMM` and `CHOLESKY` however, performance is worse than the directory-based baselines when the period is set to 100. As discussed in Section 3.7.3, both benchmarks heavily rely on spinning for synchronization. At 256 cores, spinning becomes the system bottleneck and period = 100 significantly delays the spinning core from observing the updated variable. It is generally considered bad practice to heavily use spinning at high core count.

Table 3.7: **Storage Requirement** – Storage requirement of different cache coherence protocols (bits per LLC cacheline). Ackwise assumes 4 sharers at 16 or 64 cores and 8 sharers at 256 cores.

# cores (N)	full-map directory	Ackwise	Tardis
16	16 bits	16 bits	40 bits
64	64 bits	24 bits	40 bits
256	256 bits	64 bits	40 bits

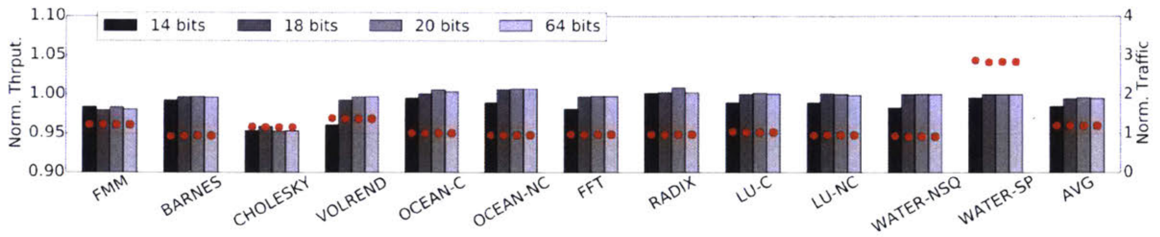


Figure 3-11: **Tardis' Performance Sensivity to the Timestamp Size.**

On average, Tardis with self increment period of 100 performs 3.4% worse than MSI with 19.9% more network traffic. Tardis with self increment period of 10 makes the performance 0.6% within baseline MSI with 46.7% traffic overhead.

Scalable storage is one advantage of Tardis over directory protocols. Table 3.7 shows the per cacheline storage overhead in the LLC for two directory baselines and Tardis. Full-map directory requires one bit for each core in the system, which is $O(N)$ bits per cacheline. Both Ackwise and Tardis can achieve $O(\log N)$ storage but Ackwise requires broadcasting support and is thus more complicated to implement.

Different from directory protocols, Tardis also requires timestamp storage for each L1 cacheline. But the per cacheline storage overhead does not increase with the number of cores.

Timestamp Size

Figure 3-11 shows Tardis's performance with different timestamp sizes. All numbers are normalized to the full-map directory-based baseline. As discussed in Section 3.5.2, short timestamps roll over more frequently, which degrades performance due to the rebase overhead. According to the results, at 64 cores, 20-bit timestamps can achieve almost the same performance as 64-bit timestamps (which never roll over in practice).

Lease

Finally, we sweep the value of the lease in Figure 8-1. Similar to the self increment period, the lease controls when a cacheline expires in the L1 cache. Roughly speaking, a large lease is equivalent to

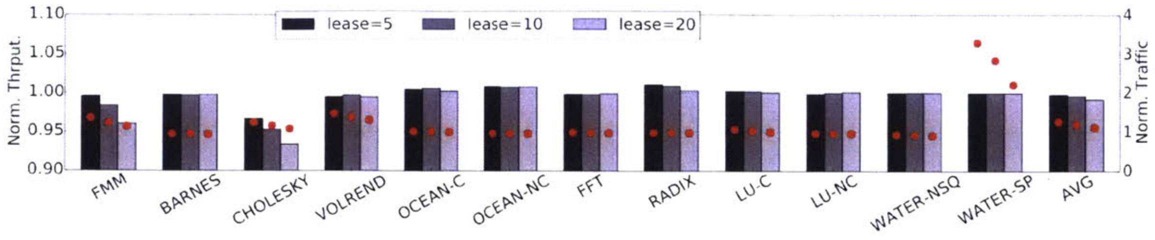


Figure 3-12: **Tardis' Performance Sensitivity to the Length of Leases.**

a long self increment period. For benchmarks using a lot of spinning, performance degrades since an update is deferred longer. The network traffic also goes down as the lease increases. For most benchmarks, however, performance is not sensitive to the choice of lease. However, we believe that intelligently choosing leases can appreciably improve performance; for example, data that is read-only can be given an infinite lease and will never require renewal. We defer the exploration of intelligent leasing to Chapter 4.

3.8 Additional Related Work

We discuss related work on timestamp based coherence protocols (cf. Section 3.8.1) and scalable directory coherence protocols (cf. Section 3.8.2).

3.8.1 Timestamp based coherence

To the best of our knowledge, none of the existing timestamp based coherence protocols is as simple as Tardis while achieving the same level of performance. In all of these protocols, the timestamp notion is either tightly coupled with physical time, or these protocols rely on broadcast or snooping for invalidation.

Using timestamps for coherence has been explored in both software [112] and hardware [116]. TSO-CC [45] proposed a hardware coherence protocol based on timestamps. However, it only works for the TSO consistency model, requires broadcasting support and frequently self-invalidates data in private caches. The protocol is also more complex than Tardis.

In the literature we studied, Library Cache Coherence (LCC) [99] is the closest algorithm to Tardis. Different from Tardis, LCC uses physical time based timestamps and requires a globally synchronized clock. LCC has bad performance because a write to a shared variable in LCC needs to wait for all the shared copies to expire and this takes a long time. This is much more expensive than Tardis which only updates a counter without any waiting. Singh et al. used a variant of LCC on GPUs with performance optimizations [131]. However, the algorithm only works efficiently for release consistency and not sequential consistency.

Timestamps have also been used for verifying directory coherence protocols [121], for ordering network messages in a snoopy coherence protocol [105], and to build write-through coherence protocols [25, 152]. None of these works built coherence protocols purely based on timestamps. Similar to our work, Martin et. al [105] give a scheme where processor and memory nodes process coherence transactions in the same logical order, but not necessarily in the same physical time order. The network assigns each transaction a logical timestamp and then broadcasts it to all processor and memory nodes without regard for order, and the network is required to meet logical time deadlines. Tardis requires neither broadcast nor network guarantees. The protocol of [25] requires maintaining absolute time across the different processors, and the protocol of [152] assumes isotach networks [80], where all messages travel the same logical distance in the same logical time.

3.8.2 Scalable directory coherence

Some previous works have proposed techniques to make directory coherence more scalable. Limited directory schemes (e.g., [11]) only track a small number of sharers and rely on broadcasting [89] or invalidations when the number of sharers exceeds a threshold. Although only $O(\log N)$ storage is required per cacheline, these schemes incur performance overhead and/or require broadcasting which is not a scalable mechanism.

Other schemes have proposed to store the sharer information in a chain [30] or hierarchical structures [101]. Hierarchical directories reduce the storage overhead by storing the sharer information as a k-level structure with $\log_k N$ bits at each level. The protocol needs to access multiple places for each directory access and thus is more complex and harder to verify.

Previous works have also proposed the use of coarse vectors [66], sparse directory [66], software support [31] or disciplined programs [33] for scalable coherence. Recently, some cache coherence protocols have been proposed for 1000-core processors [84, 127]. These schemes are directory based and require complex hardware/software support. In contrast, Tardis can achieve similar performance with a very simple protocol.

Chapter 4

Tardis with Relaxed Consistency Models

4.1 Introduction

The Tardis protocol discussed in the previous section has a few drawbacks, making it suboptimal in a real multicore system. First and foremost, Tardis only supports the *sequential consistency* (SC) memory model. While SC is simple and well studied, commercial processors usually implement more relaxed models since they can deliver better performance. Intel x86 [129] and SPARC [149] processors can support *Total Store Order* (TSO); ARM [128] and IBM Power [102] processors implement weaker consistency models. It is difficult for commercial processors to adopt Tardis if the target memory models are not supported.

Another drawback of Tardis is the renew message that is used to extend the logical lease of a shared cacheline in a private cache. These messages incur extra latency and bandwidth overhead. Finally, Tardis uses a timestamp self-increment strategy to avoid livelock. This strategy has suboptimal performance when threads communicate via spinning.

In this chapter, we will address these drawbacks of the original Tardis protocol and make it more practical. Specifically, we will discuss the changes to the cores and the memory subsystem in order to implement TSO and other relaxed consistency models on Tardis. We also propose new optimization techniques (MESI, livelock detection and dynamic leasing) to reduce the number of renew messages in Tardis, which improves performance.

Our simulations over a wide range of benchmarks indicate that our optimizations improve the performance and reduce the network traffic of Tardis. Compared to a full-map directory-based coherence protocol at 64 cores, optimized Tardis is able to achieve better performance (1.1% average improvement, up to 9.8%) and lower network traffic (2.9% average reduction, up to 12.4%) at the same time. The improvement becomes more prominent as the system scales to 256 cores where Tardis further improves performance (average 4.7%, upto 36.8%) and reduces the network traffic (average 2.6%, upto 14.6%). While the optimized and baseline Tardis protocols require timestamps in the L1 cache, they are, overall, more space-efficient than full-map directory protocols and simpler to implement.

The rest of this chapter is organized as follows. In Section 4.2, we discuss how Total Store Order (TSO) is implemented in Tardis. Then we discuss optimziations to Tardis to reduce the number of renewals in Section 4.3. Our evaluation is presented in Section 4.4.

4.2 Tardis with TSO

The original Tardis protocol only supports the sequential consistency (SC) memory model. Although SC is intuitive, it may overly constrain the ordering of memory operations. In practice, this may lead to suboptimal performance. To resolve this disadvantage of SC, relaxed consistency models have been proposed and widely implemented in real systems. Most of these models focus on the relaxation of the program order in SC (Rule 1 in the SC definition). Specifically, the program order of a core may appear out-of-order in the global memory order. The more relaxed a model is, the more flexibility it has to reorder memory operations, which usually leads to better overall performance.

In this section, we show how Tardis can be generalized to relaxed consistency models. We first use Total Store Order (TSO) as a case study since it has a precise definition and is the most widely adopted. We will present the formal definition of TSO (Section 4.2.1), the Tardis-TSO protocol (Section 4.2.2), an example program (Section 4.2.3) and optimizations (Section 4.2.4). Finally, we generalize the discussion to other memory models (Section 4.2.5).

4.2.1 Formal Definition of TSO

The TSO consistency model relaxed the Store $\rightarrow$ Load constraint in the program order. This allows a load after a store in the program order to be flipped in the global memory order (assuming that the load and the store have no data or control dependency). This means that an outstanding store does not block the following loads. In an Out-of-Order (OoO) processor, when a store reaches the head of the *Re-Order Buffer* (ROB), it can retire to the *store buffer* and finish the rest of the store operation there. The loads following the store can therefore commit early before the store finishes. Since store misses are common in real applications, this relaxation can lead to significant performance improvement.

Similar to SC, the definition of TSO also requires a global order (specified using $<_m$) of all memory instructions. However, the global memory order does not need to follow the program order for Store $\rightarrow$ Load dependency. Specifically, TSO can be defined using the following three rules [132]. The differences between TSO and SC are highlighted in boldface.

Rule 1: L(a) $<_p$ L(b) $\Rightarrow$ L(a) $<_m$ L(b)

L(a) $<_p$ S(b) $\Rightarrow$ L(a) $<_m$ S(b)

S(a) $<_p$ S(b) $\Rightarrow$ S(a) $<_m$ S(b)

~~**S(a) $<_p$ L(b) $\Rightarrow$ S(a) $<_m$ L(b)**~~ *# required in SC but not in TSO*

Rule 2: Value of $L(a)$ = Value of Max$_{<m}$ $\{S(a) \mid S(a) <_m L(a)$ **or S(a) $<_p$ L(a)**$\}$

Rule 3: **X $<_p$ FENCE $\Rightarrow$ X $<_m$ FENCE**

FENCE $<_p$ X $\Rightarrow$ FENCE $<_m$ X

In TSO, the program order implies the global memory order only for Load $\rightarrow$ Load, Load $\rightarrow$ Store and Store $\rightarrow$ Store constraints. Since there is no Store $\rightarrow$ Load constraint, a load can bypass the pending store requests and commit earlier (Rule 1). Although the load is after the store in the program order, it is before the store in the global memory order.

In an out-of-order processor, TSO can be implemented using a store buffer, which is a FIFO for pending store requests that have retired from ROB. If the address of a load is found in the store buffer, then the pending data in the store buffer is directly returned; otherwise, the load accesses the memory hierarchy (Rule 2).

TSO uses a *fence* instruction when a Store $\rightarrow$ Load order needs to be enforced (Rule 3). In a processor, a fence flushes the store buffer enforcing that all previous stores have finished so that a later committed load is ordered after stores before the fence in physical time. If all memory operations are also fences, then TSO becomes SC.

4.2.2 Tardis-TSO Protocol

In this section, we describe how TSO can be implemented on Tardis. Specifically, we discuss the changes to the timestamp management policy as compared to the Tardis SC protocol.

Program Timestamp Management

The original Tardis SC protocol uses a single program timestamp (*pts*) to represent the commit timestamp of an instruction. Since the program order always agrees with the global memory order in SC, *pts* monotonically increases in the program order.

In TSO, however, the program order does not always agree with the global memory order. Following Rule 1 in TSO's definition, a store's timestamp is no less than the timestamps of all preceding loads, stores and fences in the program order. A load's timestamp is no less than the timestamps of all preceding loads and fences, but not necessarily preceding stores. As a result, a single monotonically increasing *pts* is insufficient to represent the ordering constraint.

To express the different constraints for loads and stores respectively, we split the original *pts* into two timestamps. The *store timestamp* (*sts*) represents the commit timestamp of the last store, and the *load timestamp* (*lts*) represents the commit timestamp of the last load. Like *pts*, both *sts* and *lts* are maintained in each core in hardware. According to Rule 1, both should monotonically increase in the program order because of the Load $\rightarrow$ Load and Store $\rightarrow$ Store constraints. Furthermore, the timestamp of a store (*sts*) should be no less than the timestamp of the preceding load (*lts*) because of the Load $\rightarrow$ Store constraint. For a load, however, its *lts* can be smaller than *sts* because there is no Store $\rightarrow$ Load constraint.

A fence can be simply implemented as a synchronization point between *sts* and *lts*. Specifically, a fence sets $lts = Max(lts, sts)$. This enforces Rule 3 in TSO because operations after the fence are ordered after operations before the fence in physiological time order (and therefore the global memory order). If each memory operation is also a fence, then the commit timestamp for each operation monotonically increases and the protocol becomes Tardis SC.

In a traditional coherence protocol, the main advantage of TSO over SC is the performance gain due to loads bypassing stores in the store buffer. In Tardis, besides bypassing, TSO can also reduce the number of renewals compared to SC. This is because the *lts*/*sts* in TSO may increase more slowly as compared to the *pts* in SC. As a result, fewer shared cachelines expire.

Data Timestamp Management

The timestamp management logic largely remains the same when the consistency model switches from SC to TSO. However, the timestamp rules for data in the store buffer need some slight changes. For single-threaded cores, timestamp management in the private L1 can also be changed for load requests for potentially better performance.

Specifically, for a dirty cacheline in modified state in the store buffer or L1 cache, the *lts* does not have to be greater than the *wts* of the cacheline. With SC, in contrast, *lts* cannot be smaller than *wts*. With respect to the global memory order, the behavior in TSO means that the load can commit at an *lts* smaller than the commit timestamp of the store creating the data (*wts*). This behavior certainly violates SC but it is completely legal in TSO.

According to Rule 2 of TSO, a load should return either the last store in global memory order or the last store in program order, depending on which one has a larger physiological time. Since a dirty cacheline was written by a store from the current core prior to the load, even if the load has a smaller commit timestamp than the store, Rule 2 still holds. A more formal proof of correctness is presented in Chapter 5.

Unlike in traditional processors, TSO can be implemented with Tardis even on in-order cores that do not have a store buffer. This is because Tardis can directly implement the correct memory ordering using logical timestamps. This will become clear in the example presented in the next section. We note that our implementation of Tardis TSO still has a store buffer to improve performance.

Note that if multiple threads can access the same private cache, then the above optimization for dirty L1 cachelines may not be directly applied in the L1 cache (but it is still applied in the store buffer). This is because a dirty line might be written by any thread sharing the L1. For these systems, this optimization can be turned off in the L1.

4.2.3 TSO Example

Listing 4.1: Example Program

```
[core0]                 [core1]
 B = 1                  A = 2
 L(B) → r1              FENCE
 L(A) → r2              L(B) → r3
```

We use the example program in Listing 4.1 to demonstrate how timestamps are managed in Tardis TSO. The execution of the program is shown in Figure 4-1. For simplicity, we do not model the store buffer and execute one instruction per step for each core.

Initially, both addresses A and B are cached in Shared (S) state in both cores' private caches as well as the shared LLC. *wts* of all lines are 0; *rts* of all lines of address A are 5 and *rts* of all lines of address B are 10.

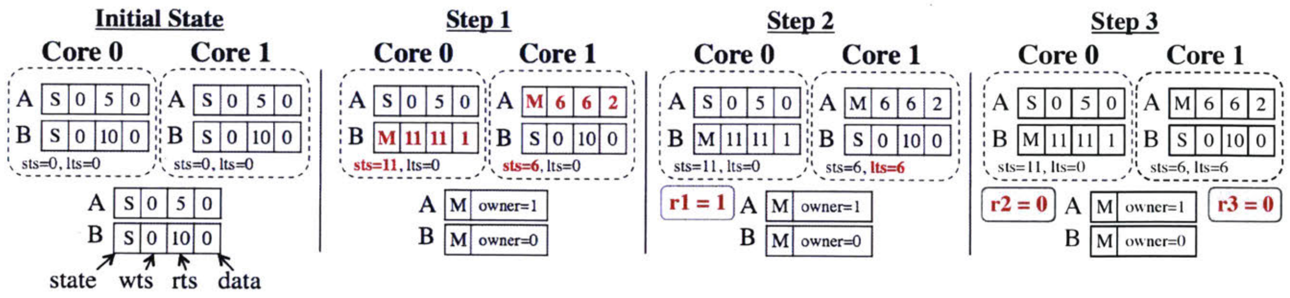


Figure 4-1: **Tardis-TSO Example** – Execution of the program in Listing 4.1 in Tardis with *lease* = 10. A private cacheline in *M* state means it is modified and dirty. Changed states at each step are highlighted in red.

Step 1: core 0 writes to address B and core 1 writes to address A. Exclusive ownership of A and B are given to core 1 and core 0, respectively, and both stores are performed by jumping ahead in logical time to the end of the lease. After the stores, core 0's *sts* jumps to timestamp 11 and core 1's *sts* jumps to 6, but the *lts* of both cores remain 0.

Step 2: core 0 loads address B. The value of the previous store from core 0 is returned ($r1 = 1$). Since B is in the dirty state, the load does not increase the *lts* (cf. Section 4.2.2). In core 1, a fence instruction is executed which synchronizes the *lts* and *sts* to timestamp 6.

Step 3: core 0 loads address A. Since its *lts* is 0 which falls between the *wts* and *rts* of cacheline A, this is a cache hit and value 0 is returned ($r2 = 0$). In core 1, the load to address B also hits the L1 since its *lts* = 6 falls within B's lease. As a result, the loaded value is also 0 ($r3 = 0$).

Listing 4.2 shows the physiological commit time for each instruction in Listing 4.1. It also shows the global memory order using arrows. Physiological time is represented using a logical timestamp and physical time pair (*ts*, *pt*) where *ts* is the commit timestamp and *pt* is the physical commit time of the instruction. According to the definition, $(ts_1, pt_1) < (ts_2, pt_2)$ if $ts_1 < ts_2$ or $(ts_1 = ts_2$ and $pt_1 < pt_2)$.

Listing 4.2: **Global memory order of the Tardis-TSO example** – The arrows show the global memory order of the all the six memory operations performed by the two cores.

$$\begin{tikzcd}
\text{[core0]} & & \text{[core1]} \\
(11,\ 1) & & (6,\ 1) \arrow[d] \\
(0,\ 2) \arrow[d] & & (6,\ 2) \arrow[d] \\
(0,\ 3) \arrow[uurr] & & (6,\ 3) \arrow[uull]
\end{tikzcd}$$

The execution is definitely not sequentially consistent since the program order in core 0 is violated between the first (`B = 1`) and the second (`L(B)`) instructions. But it obeys all the invariants of TSO. Note that the store buffer is not included in the example since we are modeling in-order cores, but TSO can still be implemented. This feature is not available in traditional physical time based coherence protocols. For this example, adding the store buffer will not change the hardware behavior.

4.2.4 TSO Optimizations

Many optimization techniques have been proposed in the literature to improve performance of the basic SC/TSO consistency model. Examples include load speculation to hide instruction stall latency due to Load $\rightarrow$ Load and Store $\rightarrow$ Load dependency, and store prefetch to enhance Store $\rightarrow$ Store and Load $\rightarrow$ Store performance [55]. For TSO, the speculation can also go over fences.

Tardis TSO is compatible with these optimizations. In fact, it may be even simpler to support them on Tardis than on traditional coherence protocols since the timestamps can help preserve/check memory order. For example, for load $\rightarrow$ load relaxation, multiple loads can be speculatively executed in parallel, and the *wts* and *rts* of the loaded cachelines are stored inside the core (e.g., in the ROB). To enforce load $\rightarrow$ load dependency, the processor only needs to commit instructions with ascending timestamp order (and reissue a load with a new timestamp if necessary). In contrast, a traditional processor needs to snoop on invalidation messages in order to detect a speculation failure. Fence speculation can also be implemented in a similar way using timestamps. In general, Tardis allows all memory operations to be speculatively executed arbitrarily, as long as their commit timestamps obey the consistency model. This flexibility makes it easier to reason about and implement these optimizations.

Table 4.1: **Memory Order Constraints for Different Consistency Models** – *L* represents a load, *S* represents a store, and *F* represents a fence. For release consistency, *rel* represents a release and *acq* represents an acquire.

Consistency Model	Ordinary Orderings	Synchronization Orderings
SC	$L \to L$, $L \to S$, $S \to L$, $S \to S$	
TSO	$L \to L$, $L \to S$, $S \to S$	$S \to F$, $F \to L$, $F \to F$
PSO	$L \to L$, $L \to S$	$S \to F$, $F \to S$, $F \to L$, $F \to F$
RC		L/S $\to$ rel, acq $\to$ L/S, rel/acq $\to$ rel/acq

4.2.5 Other Relaxed Consistency Models

Similar to TSO, other memory consistency models (Table 4.1) can also be supported in Tardis with proper changes to the timestamp rules. Given the relationship between the program order and the global memory order, it is usually straightforward to adapt Tardis for different models. In this section, we briefly discuss Partial Store Order (PSO) and Release Consistency (RC) to illustrate how Tardis can support them with minimal algorithmic change.

Partial Store Order (PSO)

The PSO consistency model [149] relaxes both the Store $\to$ Load and the Store $\to$ Store orderings. Similar to TSO, we use the *lts* and *sts* to model the program order constraints. In PSO, since Load $\to$ Load is enforced but Store $\to$ Load is not, which is the same as TSO, the rule for *lts* is also the same. Namely, *pts* should monotonically increase independently of store timestamps.

The timestamp order for stores, however, does not need to monotonically increase, since Store $\to$ Store is relaxed. Therefore, the timestamp of a store (*ts*) only needs to be no less than the current *lts* ($ts \geq lts$, due to the Load $\to$ Store constraint). And *sts* represents the largest store timestamp so far (instead of the last store timestamp), namely $sts = Max(sts, ts)$.

For a fence instruction, *lts* synchronizes with *sts*, namely $lts = Max(lts, sts)$. The resulting *lts* is the timestamp of the fence.

Release Consistency (RC)

Release consistency [56] relaxes all the program order constraints; furthermore, it also relaxes the ordering constraints for synchronizations. Specifically, an *acquire* guarantees that all of the following (but not the previous) operations are ordered after the acquire and a *release* guarantees that all the previous (but not the following) operations are ordered before the release.

In Tardis, we need to maintain timestamps for acquire (*acquire_ts*) and release (*release_ts*) operations, as well as the maximal commit timestamp (*max_ts*) so far. A normal load or store operation (commit timestamp *ts*) can be performed as long as its timestamp is greater than *acquire_ts* ($ts \geq acquire\_ts$); *max_ts* represents the largest commit timestamp as seen by the core so far ($max\_ts = Max(max\_ts, ts)$). At a release instruction, *release_ts* and *max_ts* are synchronized ($release\_ts = Max(release\_ts, max\_ts)$). At an acquire instruction, *acquire_ts* and *release_ts* are synchronized ($acquire\_ts = Max(acquire\_ts, release\_ts)$).

4.3 Renewal Reduction Techniques

As previously discussed, a major drawback of the original Tardis protocol is the requirement of cacheline renewal. With load speculation, latency of renew messages can be largely hidden, but network traffic overhead remains. As shown in Section 3.7, most of the renewed cachelines have their values unchanged. Therefore, these renewals are not necessary in theory. In this section, we discuss techniques to reduce unnecessary renew messages.

4.3.1 MESI on Tardis

The original Tardis protocol implements MSI where a read to a private cacheline loads it in *S* (shared) state in the L1 cache. As the *lts* (or *pts* in SC) increases in a core, these data will expire and be renewed. When the cacheline is private to a core, these renewals are unnecessary since there is no need to maintain coherence for private data to a core.

The MESI protocol can mitigate this issue. MESI adds an *E* (Exclusive) state to MSI. The *E* state is granted for a request if the cacheline is not shared by other cores. Like a cacheline in *M* state, an exclusive cacheline is owned by a private cache and therefore never expires. If the *lts* is greater than the *rts*, the *rts* of that line is extended silently without contacting the timestamp manager. This can be done since the line is owned by the core and not shared by other cores. Since all thread local data are in *E* state in a MESI protocol, these data do not incur any renewal request.

Different from traditional coherence protocols, Tardis can grant *E* state to a core even if other cores are still sharing the line. This is similar to granting M state without the need for invalidation. However, for performance reasons, it is still desirable to only grant *E* state to private data. Because if the *E* state is granted to a shared cacheline, then extra write back messages are required when other cores read the cacheline that is exclusively owned. In Tardis, a cacheline is likely to be not shared if it has just been loaded from DRAM, or if it has just been downgraded from the *E* or *M* to *S* state. Therefore, we add an E-bit to each cacheline in the LLC to indicate whether the cacheline is likely to be shared or not. The E-bit is set when the cacheline is loaded from DRAM and also when the cacheline is written back from a private cache. The E-bit is reset when the cacheline becomes cached upon a load request. Note that the E-bit may be unset even if no core is sharing the line (e.g., all sharers silently evict the line); or it may be set when some cores are sharing the data (e.g., the cacheline was evicted from the LLC but still cached in the L1s). Neither case affects the functional correctness of the implementation.

4.3.2 Livelock Detection

A disadvantage of Tardis is that propagating a store to other cores may take an arbitrarily long time. Because there is no invalidation mechanism, a writer does not notify the cores that currently cache a stale version of the cacheline. In the worst case, if a core spins on a stale cacheline with a small *lts* (or *pts* in SC), it never sees the latest update and livelock occurs (Listing 4.3). In practice, this spinning behavior is commonly used for communication in parallel programs. Although such livelock is not precluded by the consistency model, it should be disallowed by the coherence protocol. Therefore, every write should eventually propagate to other cores.

Listing 4.3: Threads communicate through spinning

```
      [Core 0]                [Core 1]
//spin (lts = 10)        //store (sts = 20)
while (!done) {          ...
      pause              done = true
}                        ...
```

Baseline: Periodic Self Increment

The original Tardis protocol solves the livelock problem by self incrementing the *pts* (or *lts* in TSO) periodically to force the logical time in each core to move forward. For a spinning core (e.g., core 0 in Listing 4.3), the *lts* will increase and eventually become greater than the *rts* of the cacheline being spun on at which point the line expires and the latest value will be loaded.

However, this simple livelock avoidance mechanism has some performance issues. It is difficult to set the self increment rate for this mechanism. Setting the rate too high or too low both cause performance degradation. For example, a cacheline being spun on may have an *rts* much larger than the current *lts* of the core. Therefore, if the increasing rate is set low, it takes significant time before the *lts* increases to *rts* for the stale line to expire.

On the other hand, if the rate is set too high, performance also suffers since the shared cachelines would expire frequently. In this case, the *lts* quickly exceeds the *rts* of cached data and thus renewals are required. Most of these renewals are unnecessary and can be avoided if the self increment rate is lower.

Livelock Detector

Relying on cacheline expiration is not the only way to avoid livelocks. We make a key observation that a separate message can be sent to check the freshness of a cacheline before its lease expires in a private cache. Like a renew request, if the latest data is newer than the cached data, the latest cacheline is returned. If the cached data is already the latest version, however, a check response is returned without extending the *rts* of the cacheline in the LLC.

The check request can resolve the dilemma of picking an appropriate increasing rate for *lts*. First, with the introduction of the check message, a core does not need increase its *lts* to be greater than the *rts* of a cached line to detect staleness. Therefore, the self increment rate can be reduced, which decreases the number of renewal requests. Second, since a check request can be sent when *lts* is much smaller than *rts*, a core no longer need to wait for a long physical time to detect staleness.

Generally, a check request should be sent when the program seems to livelock as it keeps loading stale cachelines. In practical programs, such a livelock is usually associated with variable spinning, which typically involves a small number of cachelines and is therefore easy to detect in hardware.

We designed a small piece of hardware next to each core to detect livelock. It contains an *address history buffer* (AHB) and a *threshold counter* (thresh_count). The AHB is a circular buffer keeping track of the most recently loaded addresses. Each entry in AHB contains the address of a memory access, and an *access_count*, which is the number of accesses to the address since it was loaded to AHB. When *access_count* becomes greater than the *thresh_count*, a check request is sent for this address (Algorithm 1). The value of *thresh_count* can be static or dynamic. We chose to use an adaptive threshold counter scheme (Algorithm 2) in order to minimize the number of unnecessary check messages.

The livelock detection algorithm (Algorithm 1) is executed when a core reads a shared cacheline. It is not executed when a core accesses cachelines in E or M state since no livelock can occur for those accesses. If the accessed address does not exist in the AHB, a new entry is allocated. Since AHB is a circular buffer, this may evict an old entry from it. We use the LRU replacement policy here but other replacement policies should work equally well. For an AHB hit, the *access_count* is incremented by 1. If the counter saturates (i.e., reaches *thresh_count*), a check request is sent and the *access_count* is reset. All *access_counts* are reset to 0 when the *lts* increases due to a memory access, since this indicates that the core is not livelocking, and thus there is no need to send checks.

Algorithm 1: Livelock Detection Algorithm (called for each read request to a shared L1 cacheline).

```
1: Input: addr    // memory access address
2: Return Value: whether to issue check request
3: Internal State: AHB, thresh_count
4: if AHB.contains(addr) then
5:    AHB[addr].access_count ++
6:    if AHB[addr].access_count == thresh_count then
7:       AHB[addr].access_count = 0
8:       return true
9:    end if
10: else
11:    AHB.enqueue(addr)
12:    AHB[addr].access_count = 0
13:    return false
14: end if
```

The counter *thresh_count* may be updated for each check response (Algorithm 2). If the checked address was updated, then *thresh_count* should be reset to the minimal value, indicating that check requests should be sent more frequently since data seems to be updated frequently. Otherwise, if *check_thresh* number of consecutive check requests returned and the cacheline has not be modified, then *thresh_count* is doubled since it appears unnecessary to send check requests that often. Adaptively determining the value of *thresh_count* can reduce the number of unnecessary check requests if a thread needs to spin for a long time before the data is updated.

Note that the livelock detector can only detect spinning involving loads to less than M (the number of entries in AHB) distinct addresses. Therefore, the livelock detector cannot capture all possible livelocks and thus self incrementing *lts* is still required to guarantee forward progress. For practical programs, however, spinning typically involves a small number of distinct addresses.

Algorithm 2: Adaptive Threshold Counter Algorithm (called for each check response).

```
1: Input: check_update // whether the checked address has been updated.
2: Internal State: thresh_count
3: Constant: min_count, max_count, check_count, check_thresh
4: if check_update then
5:     thresh_count = min_count
6:     check_count = 0
7: else
8:     check_count ++
9:     if check_count == check_thresh and
       thresh_count < max_count then
10:        thresh_count = thresh_count × 2
11:    end if
12: end if
```

So the livelock detector is able to capture livelock in most programs. We still self increment *lts* periodically but the frequency can be much lower, which can significantly reduce the number of renewals due to lease expiration.

4.3.3 Lease Prediction

Besides self incrementing *lts*, memory stores are the main reason that the timestamps increase in Tardis. The amount that an *sts* increases is determined by the lease of the previous data version, because the *sts* of the store must be no less than the cacheline's previous *rts*. Therefore, the lease of each cacheline is important to the timestamp incrementing rate as well as the renew rate. The original Tardis protocol uses a static lease for every shared cacheline. We first show that a static leasing policy may incur unnecessary renewals. We then propose a dynamic leasing policy to mitigate the problem.

Static Lease vs. Dynamic Lease

We use the example in Listing 4.4 as a case study to show when static leases are ineffective. In this example, both cores run the same program. They both load addresses A and B and then store to B. When either address A or B is loaded to an L1 cache, a lease L is taken on the cacheline. With static leases, L is a constant. In the first iteration, after loading, each cacheline has a lease of $(0, L)$. When the store to address B is performed, the core's *sts* jumps ahead to the end of the lease ($sts = L+1$). At the end of the first iteration, the FENCE instruction increases *lts* to the value of *sts*. Therefore, both *lts* and *sts* equal $L+1$.

Listing 4.4: The case study parallel program

```
[Core 0]                  [Core 1]

 while(B < 10) {          while(B < 10) {
   print A                   print A
   B++                       B++
   FENCE                     FENCE
 }                         }
```

In the next iteration, when both cores load A again, they both see that A has expired in their L1 caches. The expiration of cacheline A is because the *lts* (i.e., $L+1$) is greater than the end of the lease (i.e., L). This expiration is due to the store to B in the previous iteration as well as the FENCE at the end of the loop. As a result, both cores need to renew cacheline A at the timestamp manager. And these renewals need to happen for each following iteration of the loop. In principle, these renewals to A are unnecessary, since A has never been changed. Note that using a larger static L does not solve the problem. This is because a store to B will simply jump further ahead in logical time and cacheline A will still expire.

Our solution to this problem is to use different leases for different addresses. Intuitively, we want to use large leases for read only or read intensive data, and use small leases for write intensive data. In the example in Listing 4.4, if A has a lease 10 and B has a lease 0, then each store to B increases the *sts* and *lts* only by 1. So it takes about 10 iterations before A has to be renewed again. The renew rate is mainly a function of the ratio between these two leases; the absolute lease values are not critical.

In a real system, it is non-trivial to decide what data is more read intensive and therefore should have a larger lease. Here, we explore hardware only solutions and design a predictor to decide the lease for each cacheline. It is possible to do this more accurately with software support; such explorations are left for future work.

Lease Predictor

Our lease predictor is based on the observation that cachelines that are frequently renewed are more likely to be read intensive. Therefore, a cacheline should have a larger lease if it is renewed frequently. A *lease predictor* is built into the timestamp manager to process this logic.

Initially, all the cachelines have the minimal leases (*min_lease*). The lease predictor increases the lease for a certain cacheline when it appears more read intensive. The algorithm for dynamic leasing is shown in Algorithm 3. For a renew request from the L1 to the timestamp manager, the last lease (*req_lease*) of the cacheline is also sent.

Algorithm 3: Lease Prediction Algorithm (called for each LLC request).

```
1: Input
2:     req_lease  # For a Renew request, req_lease is the previous lease of the cacheline
3:     req_type   # Write, Read or Renew
4: Return Value: For a Read or Renew request, the lease taken on the cacheline.
5: Internal State: cur_lease, min_lease, max_lease

6: if req_type == WRITE then
7:     cur_lease = min_lease
8: else if req_type == RENEW and req_lease == cur_lease
     and cur_lease < max_lease then
9:     cur_lease = cur_lease × 2
10: end if
11: return cur_lease
```

For a write request, the *cur_lease* is updated to the minimal value that a lease can be (*min_lease*). Our reasoning is that the write indicates that the cacheline might be write intensive. Therefore, assigning a large lease to it makes the *lts* jump ahead further, which causes unnecessary renewals of other cachelines. For read request (i.e., a L1 cache miss), *cur_lease* is used for the requested cacheline. For a renew request, *cur_lease* is compared with the request lease (*req_lease*). If they are different, then *cur_lease* is used for the cacheline. Otherwise, *cur_lease* is doubled since the cacheline seems to be renewed multiple times by the same core and is therefore likely to be read intensive. If *cur_lease* already reached the maximal value (*max_lease*), then it should no longer increase.

The initial value of *cur_lease* is the minimal value a lease can have (*min_lease*). This means that for a cacheline first loaded to the LLC, we always assume it is write intensive. We made this design decision because incorrectly giving a large lease to a write intensive cacheline is much more harmful than giving a small lease to a read intensive cacheline. If a cacheline with a large lease is written, a large number of cachelines in the core's L1 might expire due to the program timestamp jumping ahead. In contrast, if a read only cacheline is given a small lease, only this cacheline needs to be renewed and other cachelines are not affected.

4.4 Evaluations

In this section, we evaluate the performance of Tardis with relaxed consistency models and the optimizations proposed in Section 4.3.

Table 4.2: **System Configuration**.

System Configuration	
Number of Cores	N = 64
Core Model	Out-of-order, 128-entry ROB
Memory Subsystem	
Cacheline Size	64 bytes
L1 I Cache	16 KB, 4-way
L1 D Cache	32 KB, 4-way
Shared L2 Cache per Core	256 KB, 8-way
DRAM Bandwidth	8 MCs, 10 GB/s per MC
DRAM Latency	100 ns
2-D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link)
Flit Width	128 bits

Table 4.3: **Tardis Configuration**.

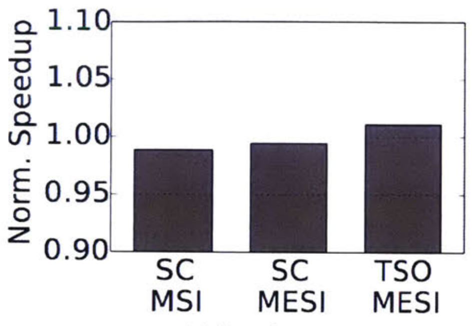
Baseline Tardis	
Static Lease	8
Self Increment Period	100 memory accesses
Livelock Detector	
AHB size	8 entries
Threshold Counter	min 100, max 800
Check Threshold	10
Lease Prediction	
Minimal Lease Value	8
Maximal Lease Value	64

4.4.1 Methodology

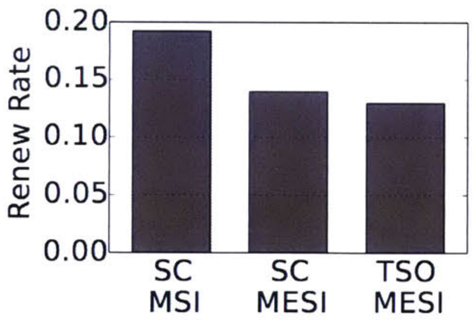
System Configuration

We use the Graphite [110] multicore simulator to model the Tardis coherence protocol. Configurations of the hardware, Tardis, and its enhancements are shown in Table 4.2 and Table 4.3.

The baseline Tardis has the same configuration as discussed in Section 3.7. The static lease is chosen to be 8. And the *lts* self increments by 1 for every 100 memory accesses. In the optimized Tardis, the address history buffer (AHB) contains 8 entries by default. The threshold counter can take values ranging from 100 to 800. The threshold counter is doubled if 10 consecutive checks respond that the data has not been changed. The minimal lease is chosen to be 8 and the maximum lease is 64. Therefore, four possible lease values (i.e., 8, 16, 32, 64) exist. We chose four lease values because having more values does not significantly affect performance.



(a) Speedup



(b) Renew Rate

Figure 4-2: **MESI and TSO** – Average speedup (normalized to directory + MESI + TSO) and renew rate over all benchmarks.

Baselines

The following coherence protocols are implemented and evaluated for comparison. With the exception of Section 4.4.2, all the configurations use the TSO consistency model and MESI.

Directory: Full-map MESI directory coherence protocol.

Base Tardis: Baseline Tardis as discussed in Chapter 3.

Tardis + live: Baseline Tardis with livelock detection. *lts* self increments by 1 for every 1000 memory accesses.

Tardis + live + lease: Tardis with both a livelock detector and lease predictor.

Our experiments are executed over 22 benchmarks selected from Splash2 [153], PARSEC [24], sparse linear algebra [43] and OLTP database applications [157]. For sparse linear algebra, we evaluated sparse matrix multiplication (`SPMV`) and symmetric Gauss-Seidel smoother (`SYMGS`) from the HPCG benchmark suite (Top 500 supercomputer ranking). For OLTP database, we evaluated two benchmarks `YCSB` and `TPCC`. All benchmarks are executed to completion.

4.4.2 Consistency Models and MESI

Figure 4-2 shows the speedup of MESI and TSO on Tardis normalized to the baseline directory coherence protocol with MESI and TSO. All experiments have both livelock detection and lease prediction enabled. Both MESI and TSO can improve the overall performance of Tardis. On average, using MESI instead of MSI improves the performance of Tardis SC by 0.6%; using TSO instead of SC further improves performance by 1.7%.

MESI and TSO can also reduce the renew rate of Tardis. We define renew rate as the ratio of the number of renew requests over the total number of LLC accesses. Figure 4-2b shows the renew rate

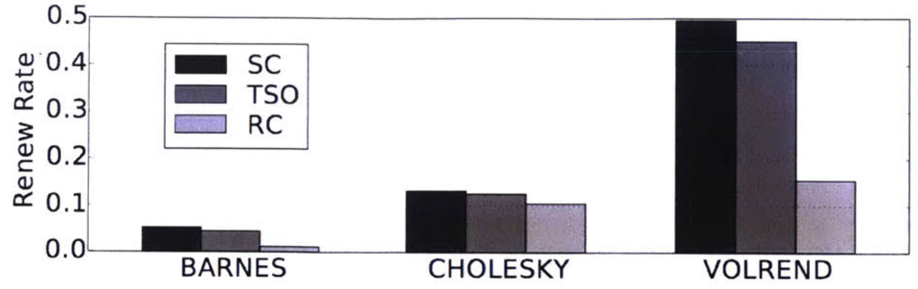


Figure 4-3: **Renew Rate of SC, TSO and RC.**

reduction of MESI and TSO on Tardis. MESI reduces the number of renew messages since private read-only data is always cached in E state and therefore never renewed. TSO further reduces the renew rate since the *lts* may increase slower than the *pts* in SC (cf. Section 4.2.2) leading to fewer expirations and renewals. MESI and TSO together can reduce the average renew rate from 19.2% to 13.0%.

Although not shown in these figures, TSO can also significantly decrease the rate at which timestamps increase. This is because *lts* can stay behind *sts*. Therefore, *lts* and *sts* may increase slower than how *pts* increases in Tardis SC. On average, the timestamp increment rate in Tardis TSO is 78% of the rate in Tardis SC.

Figure 4-3 shows the renew rate of SC, TSO and RC on Tardis with MESI. Due to some issues with running pthread synchronization primitives on RC, we implemented hardware-based synchronization for this experiment and therefore use a separate graph to present the results. For these benchmarks, relaxed consistency models lead to significantly fewer renewals. This is because more relaxed models allow stale cachelines to be read by a core, making renewals less necessary.

4.4.3 Livelock Detector and Lease Predictor

We now evaluate the performance and hardware cost of the livelock detector and lease predictor presented in Section 4.3. All coherence protocols in this section use MESI and TSO.

Performance and Network Traffic

Figure 4-4 shows the performance of Tardis after adding livelock detection and lease prediction compared to the baseline Tardis protocol. All numbers are normalized to the baseline full-map directory protocol.

First, we see that `CHOLESKY` and `SYMGS` on baseline Tardis have much worse performance than the directory protocol. Both benchmarks *heavily* use spinning to communicate between threads. As

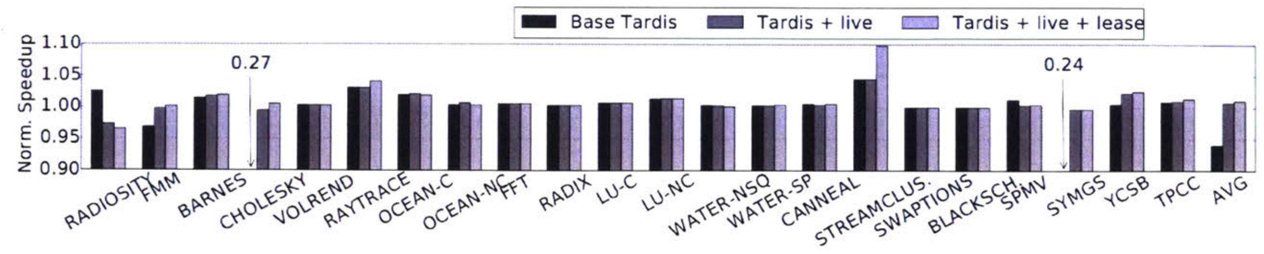


Figure 4-4: **Speedup from Renew Reduction Optimizations** – Performance improvement of livelock detection and lease prediction optimizations. All results are normalized to the full-map directory baseline.

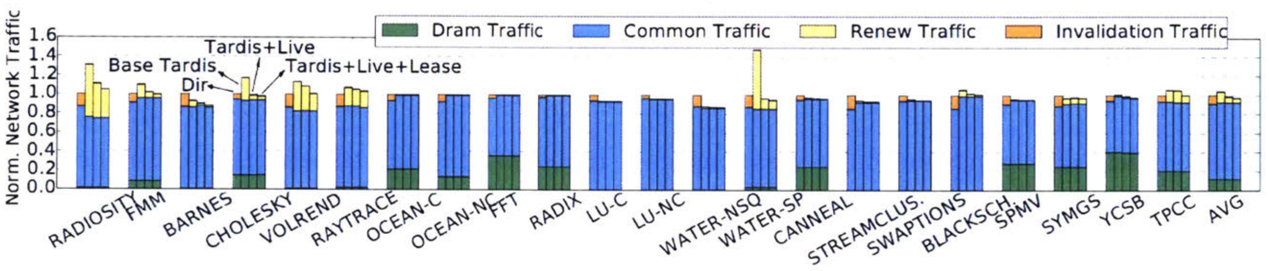


Figure 4-5: **Network Traffic Reduction from Renew Reduction Optimizations** – Network traffic breakdown of different cache coherence configurations. All results normalized to the directory baseline.

a result, it may take a long time for the cacheline spun on to expire. The livelock detector can close the performance gap between Tardis and directory because a spinning core is able to observe the latest data much earlier. RADIOSITY also uses spinning. However, the core does other computation between checking the value of the variable being spun on. Therefore, our livelock detector cannot capture such spinning behavior and forward progress is enforced through self incrementing *lts*. This leads to suboptimal performance. Completely eliminating such performance degradation requires rewriting the application using synchronization primitives that are better than spinning. Over the benchmark set, the optimizations improve the performance of Tardis by 7.5% with respect to the baseline Tardis and 1.1% (up to 9.8%) with respect to baseline directory protocol.

Figure 4-5 shows the network traffic breakdown for the same four configurations as in Figure 4-4. For each experiment, we show *dram traffic*, *common traffic*, *renew traffic* and *invalidation traffic*. *Common traffic* is the traffic in common for both directory coherence and Tardis, including shared, exclusive and write back memory requests and responses. *Renew traffic* is specific to Tardis including renew and check requests and responses. *Invalidation Traffic* is specific to the directory-based protocol, including the invalidation requests to shared copies from the directory, as well as the messages sent between the L1 caches and the LLC when a shared cacheline is evicted.

Compared to a directory-based protocol, Tardis is able to remove all the invalidation traffic. However, the renew traffic adds extra overhead. The baseline Tardis configuration incurs a large amount of renew traffic on some benchmarks (e.g., `RADIOSITY`, `CHOLESKY`, `VOLREND` and `WATER-SP`). Some of the renew traffic is due to fast self incrementing *lts* (e.g., `RADIOSITY`, `CHOLESKY` and `WATER-SP`). For these benchmarks, the livelock detection scheme can significantly reduce the self increment rate and therefore reduce the amount of renew traffic. On average, the livelock detection algorithm reduces the total network traffic by 5.4% compared to the baseline Tardis.

For some benchmarks, shared cachelines expire because the *lts* jumps ahead due to a write (e.g., `VOLREND`, `RADIOSITY`) and renew messages are generated. Our lease prediction algorithm is able to reduce these unnecessary renewals by using a larger lease for read intensive cachelines. On top of the livelock detection optimization, lease prediction further reduces the total network traffic by 1.5% on average. With both livelock detection and lease prediction, Tardis can reduce the total network traffic by 2.9% (up to 12.4%) compared to the baseline directory protocol.

Although not shown here, we also evaluated an idealized leasing scheme which is modeled by giving each successful renew message zero overhead. This idealized configuration models a scheme where the optimal leases are chosen through an oracle. The idealized scheme has similar performance as the optimized Tardis but eliminates almost all the renewal messages; some renewals are still needed if the data has actually been changed.

Hardware Complexity

The hardware overhead for the livelock detector and lease predictor is generally small. Each livelock detector contains 8 AHB entries and each entry requires an address and a counter. Assuming 48-bit address space and a counter size of 2 bytes, the detector only requires 64 bytes of storage per core.

To implement the lease predictor, we need to store the current lease for each LLC and L1 cacheline. The lease is also transferred for each renew request and response. However, there is no need to store or transfer the actual value of the lease. Since a lease can only take a small number of possible values (e.g., 4 in our evaluation), we can use a smaller number of bits (e.g., 2 bits) to encode a lease. Therefore, the storage overhead is less than 0.4%.

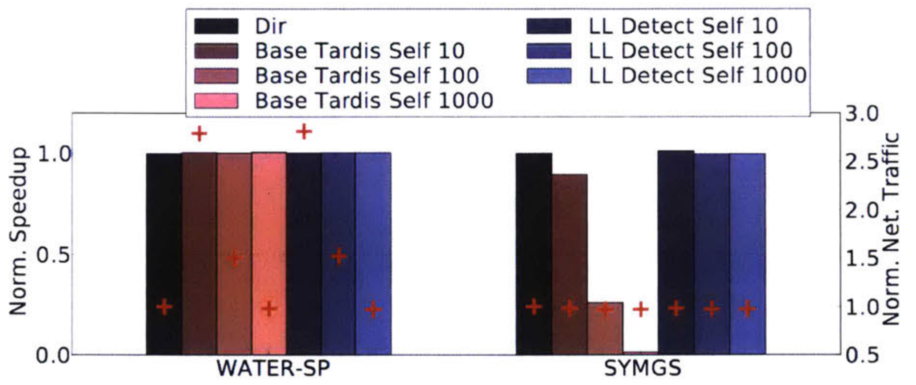


Figure 4-6: **Sensititvity of Tardis to the *lts* Self Increment Rate** – Normalized speedup (bars) and network traffic (+ signs) of Tardis as the *lts* self increment period changes.

4.4.4 Sensitivity Study

Self Increment Rate

Figure 4-6 shows the performance and network traffic of Tardis sweeping the self increment period with and without livelock detection. The self increment period is the number of read requests before *lts* increments by one. All results are normalized to a baseline directory-based protocol. The `Base Tardis Self 100` is the default baseline Tardis configuration and `LL Detect Self 1000` is the default optimized Tardis configuration (`LL Detect` stands for livelock detection).

In `WATER-SP`, changing the self incrementing rate does not affect performance regardless of whether livelock detection is turned on or not. This is because `WATER-SP` does not have spinning so livelock detection does not have any effect. However, the network traffic sees a significant reduction when the self increment period is large. This is because a large period slows down the increasing rate of *lts*, which reduces the number of cacheline expirations and renewals.

In `SYMGS`, for Tardis without livelock detection, performance is very sensitive to the self increment period because `SYMGS` intensively uses spinning to communicate between threads. If self increment is less frequent, a thread waits longer for the stale data to expire and thus performance degrades. With livelock detection, however, check requests are sent when spinning (potential livelock) is detected. Therefore, the latest value of a cacheline spun on can be returned much earlier. In contrast, Tardis with the livelock detector can always match the performance of the baseline directory protocol, regardless of the self increment period. With livelock detection and a large self increment period (e.g., 1000), Tardis can achieve good performance and low network traffic at the same time.

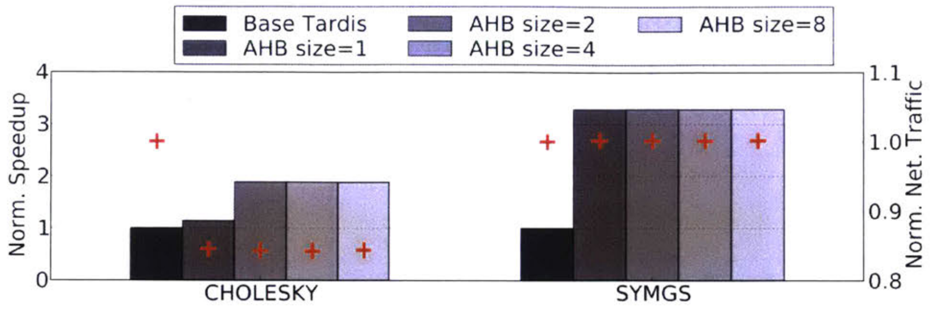


Figure 4-7: **Sensitivity of Tardis to the Size of the Address History Buffer** – Throughput and network traffic of CHOLESKY and SYMGS in Tardis as the AHB size changes. Results normalized to the baseline Tardis without livelock detection.

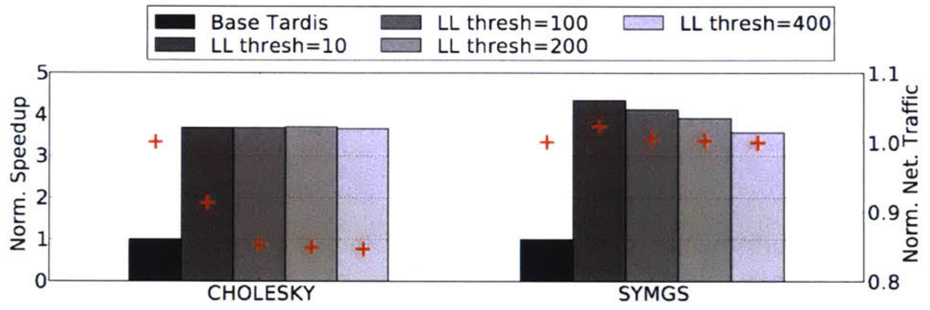


Figure 4-8: **Sensitivity of Tardis to the Threshold of Livelock Detection** – Throughput and network traffic of CHOLESKY and SYMGS in Tardis as the check threshold changes. Results normalized to the baseline Tardis without livelock detection.

Address History Buffer Size

In Figure 4-7, we swept the number of entries in the address history buffer in a livelock detector for CHOLESKY and SYMGS. According to the results, after the AHB buffer size is no less than 2, performance does not change. This is because in both (and most other) programs, spinning only involves a very small number of distinct memory addresses. CHOLESKY only spins on two addresses and SYMGS only spins on one address. Tardis uses a buffer size eight by default.

There do exist benchmarks where some other work is done during spinning and thus more than eight distinct addresses are involved (e.g., RADIOSITY). Here, livelock detection is ineffective and forward progress is guaranteed by self incrementing the program timestamp.

Livelock Threshold Counter

Figure 4-8 shows the performance and network traffic normalized to the baseline Tardis when sweeping the minimal threshold counter (*min_counter* in Algorithm 2) in the livelock detector. The

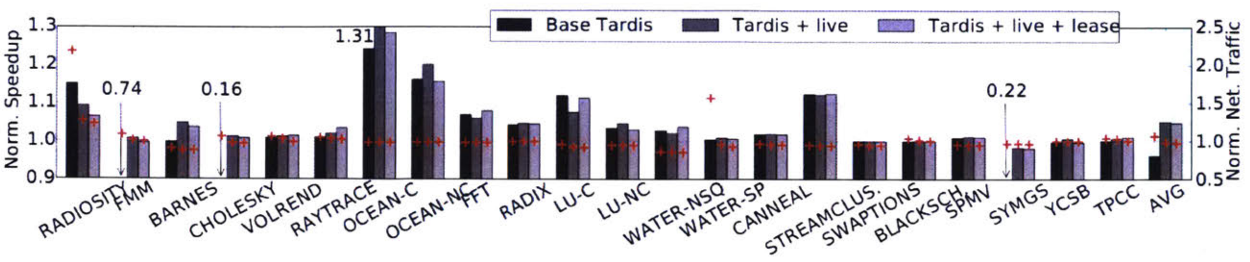


Figure 4-9: **Performance Comparison at 256 Cores** – Speedup and network traffic of different cache coherence protocols at 256 cores.

maximal threshold counter is always eight times of the minimal value. Whenever an address in the AHB has been accessed for *thresh_count* times, a check request is sent to the timestamp manager. With a larger *thresh_count*, checks are sent after spinning for a longer period of time which can hurt performance. On the other hand, larger *thresh_count* can reduce the total number of check messages and network traffic. In practice, the *thresh_count* should be chosen to balance this trade-off. We chose 100 as the default threshold counter.

Scalability

Finally, Figure 4-9 shows the performance and network traffic (normalized to the directory baseline) of all benchmarks running on a 256-core system. Compared to the directory baseline, the optimized Tardis improves performance by 4.7% (upto 36.8%) and reduces the network traffic by 2.6% (upto 14.6%). Although not shown in the graph, we also evaluated Tardis and the directory baseline where both schemes consume the same area overhead. Since Tardis requires less area than directory for coherence meta data, it can have a 37% larger LLC. In area normalized evaluation, Tardis can outperform the baseline directory by 6% on average.

Note that at 256 cores, the performance improvement of Tardis is greater than the 64 core case. This indicates that Tardis not only has better scalability in terms of storage as core count increases, it also scales better in terms of performance.

Chapter 5

Formal Proof of Correctness for Tardis

5.1 Introduction

In this chapter, we formally prove the correctness of the Tardis cache coherence protocol by showing that a program using Tardis strictly follows *Sequential Consistency* (SC) and Total Store Order (TSO). We also prove that the Tardis protocol can never deadlock or livelock. The original Tardis protocol has a number of optimization techniques applied for performance improvement. In this chapter, however, we only prove the correctness for the core Tardis protocol with MSI. We then briefly discuss the correctness of generalizations of the base protocol.

We prove the correctness of Tardis by developing simple and intuitive invariants of the system. Compared to the popular model checking [42, 79, 109] verification techniques, our proof technique is able to scale to high processor counts. More important, the invariants we developed are more intuitive to system designers and thus provide more guidance for system implementation.

The rest of the chapter is organized as follows. First, the Tardis protocol is formally defined in Section 5.2. It is proven to obey sequential consistency in Section 5.3 and to be deadlock-free and livelock-free in Section 5.4. We then prove that Tardis can correctly implement TSO in Section 5.5. We extend the proof to systems with main memory in Section 5.6. Finally, Section 5.7 describes some related work.

5.2 Tardis Coherence Protocol

We first present the model of the shared memory system we use, along with our assumptions, in Section 5.2.1. Then, we introduce system components of the Tardis protocol in Section 5.2.2 and

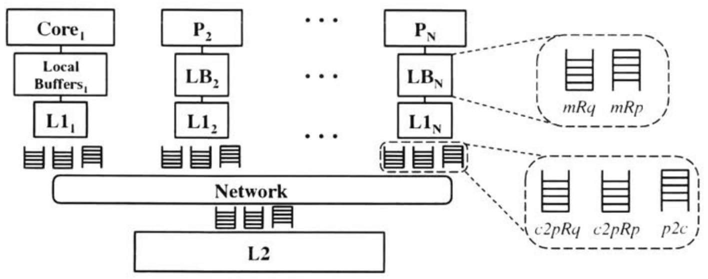


Figure 5-1: **The Model of a Multi-Core Processor** – Buffers are used for communication between a core and its local L1 cache (*mRq* and *mRp*), and between an L1 and the shared L2 (*c2pRq*, *c2pRp* and *p2c*).

formally specify the protocol in Section 5.2.3.

5.2.1 System Description

Figure 5-1 shows the architecture of a shared memory multicore system based on which Tardis will be defined. The cores can execute instructions in-order or out-of-order but always commit instructions in order. A processor talks to the memory subsystem through a pair of *local buffers* (LBs). Load and store requests are inserted into the *memory request buffer* (*mRq*) and responses from the memory subsystem are inserted into the *memory response buffer* (*mRp*).

We model a two-level memory subsystem assuming all data fits in the L2 cache. Discussion of the main memory will be discussed later in Section 5.6. The network between L1 and L2 caches is modeled as buffers. *c2pRq* (standing for "child to parent requests") contains requests from L1 to L2, *c2pRp* (standing for "child to parent responses") contains responses from L1 to L2, and *p2c* (standing for "parent to child") contains both requests and responses from L2 to L1. For simplicity, all the buffers are modeled as FIFOs and *get_msg*() returns the head message in the buffer. However, the protocol also works if the buffers only have the FIFO property for messages with the same address, and messages with different addresses can be received out-of-order. Each L1 cache has a unique *id* from 1 to *N* and each buffer associated with the L1 cache has the same *id*. The *id* is also associated with the cachelines or messages in the corresponding caches or buffers.

5.2.2 System Components in Tardis

In our model, the state of the system can be expressed using the states of the caches and buffers. The states of these components have been defined in Table 5.1.

Table 5.1: **System components of Tardis** – States of caches and buffers.

Component	States	Message Types
L1	*L1* [*addr*] = (*state*, *data*, *busy*, *dirty*, *wts*, *rts*)	-
L2	*L2* [*addr*] = (*state*, *data*, *busy*, *owner*, *wts*, *rts*)	-
mRq	*mRq.entry* = (*type*, *addr*, *data*, *pts*)	*LdRq*, *StRq*
mRp	*mRp.entry* = (*type*, *addr*, *data*, *pts*)	*LdRp*, *StRp*
c2pRq	*c2pRq.entry* = (*id*, *type*, *addr*, *pts*)	*GetS*, *GetM*
c2pRp	*c2pRp.entry* = (*id*, *addr*, *data*, *wts*, *rts*)	*WBRp*
p2c	*p2c.entry* = (*id*, *msg*, *addr*, *state*, *data*, *wts*, *rts*)	*ToS*, *ToM*, *WBRq*

Each L1 cacheline contains five fields: *state*, *data*, *busy*, *wts* and *rts*. The *state* can be M, S or I. For ease of discussion, we define a total ordering among the three states $I < S < M$. *data* contains the data value of the cacheline. A cacheline has $busy = True$ if a request to L2 is outstanding with the same address. A request can be sent only if *busy* is set to *False*, which prevents duplicate requests sent from the same core. If *dirty* is set to true, the cacheline has been modified by the core after loading into the L1. Finally, *wts* and *rts* indicate the lease on the cacheline.

An L2 cacheline has similar states as an L1 cacheline. But different from an L1 cacheline, it contains one more field *owner*, which is the *id* of the L1 that exclusively owns the cacheline in the M state. As in L1, *busy* in L2 is set when a write back request (*WBRq*) to an L1 is outstanding.

Each entry in *mRq* contains four fields: *type*, *addr*, *data* and *pts*. The *type* can be S or M corresponding to a load or store request, respectively. The *pts* is a timestamp specified by the processor indicating the minimum timestamp that the memory request must be performed at. *mRp* has the same format as *mRq*, but *pts* here is the actual timestamp of the memory operation. *pts* in *mRp* must be no less than the *pts* in the corresponding *mRq*.

Three buffers (*c2pRq*, *c2pRp* and *p2c*) are used to model the network. For a message in each buffer, *id* identifies the L1 cache that the message comes from or goes to. A message in *c2pRq* can be either a shared (GetS) or modified (GetM) request, which is indicated by the *type* field. The *addr*field is the address of the request, and *pts* is inherited from the request from the core (in *mRq*). A message in *c2pRp* must be a write-back response (*WBRp*), which contains both the address and data of the cacheline, as well as its *wts* and *rts*. Finally, a message in *p2c* can be a shared response (*ToS*), modified response (*ToM*), or write-back request (*WBRq*). Whether it is a request or response is determined by the *msg* field, which can be either *Req* or *Resp*. Whether it is a shared or modified response is determined by the *type* field.

Table 5.2: **State Transition Rules for L1.**

Rules and Condition	**Action**
LoadHit	mRq.deq()
let $(type, addr, \_, pts) = mRq.get\_msg()$	mRp.enq$(type, addr, data, \max(pts, wts))$
let $(state, data, busy, dirty, wts, rts) = L1\ [addr]$	**If** $(state = M)$
condition: $\neg\, busy \land type = S \land (state = M \lor (state = S \land pts \leq rts))$	$\quad rts := \max(pts, rts)$
StoreHit	**If** $(dirty = True)$
let $(type, addr, data, pts) = mRq.get\_msg()$	$\quad$ **let** $pts' = \max(pts, rts)$
let $(state, data, busy, dirty, wts, rts) = L1\ [addr]$	**Else**
condition: $\neg\, busy \land type = M \land state = M$	$\quad$ **let** $pts' = \max(pts, rts + 1)$
	mRq.deq()
	mRp.enq$(type, addr, _, pts')$
	$wts := pts'$
	$rts := pts'$
	$dirty := True$
L1Miss	$c2pRq$.enq$(id, type, addr, pts)$
let $(type, addr, data, pts) = mRq.get\_msg()$	$busy := True$
let $(state, data, busy, dirty, wts, rts) = L1\ [addr]$	
condition: $\neg\, busy \land (state < type \lor (state = S \land pts > rts))$	
L2Resp	$p2c$.deq()
let $(id, msg, addr, state, data, wts, rts) = p2c.get\_msg()$	$l1state := state$
let $(l1state, l1data, busy, dirty\ l1wts, l1rts) = L1\ [addr]$	$l1data := data$
condition: $msg = Resp$	$busy := False$
	$dirty := False$
	$l1wts := wts$
	$l1rts := rts$
Downgrade	**If** $(state = M)$
let $(state, data, busy, dirty, wts, rts) = L1\ [addr]$	$\quad c2pRp$.enq$(id, addr, data, wts, rts)$
condition: $\neg\, busy \land \exists\, state'.\ state' < state$	$state := state'$
$\land$ LoadHit and StoreHit cannot fire	
WriteBackReq	$p2c$.deq()
let $(state, data, busy, dirty, wts, rts) = L1\ [addr]$	**If** $(state = M)$
condition: $p2c.get\_msg().msg = Req$	$\quad c2pRp$.enq$(id, addr, data, wts, rts)$
$\land$ LoadHit and StoreHit cannot fire	$\quad state := S$

Table 5.3: **State Transition Rules for L2.**

Rules and Condition	Action
ShReq_S	$c2pRq$.deq()
let $(id, type, addr, pts) = c2pRq.get\_msg()$	$rts := pts'$
let $(state, data, busy, owner, wts, rts) = L2\ [addr]$	$p2c$.enq$(id, Resp, addr, S, data, wts, pts')$
condition: $type = S \land state = S$	
$\land \exists\, pts'.\ pts' \geq rts \land pts' \geq pts$	
ExReq_S	$c2pRq$.deq()
let $(id, type, addr, pts) = c2pRq.get\_msg()$	$state := M$
let $(state, data, busy, owner, wts, rts) = L2\ [addr]$	$owner := id$
condition: $type = M \land state = S$	$p2c$.enq$(id, Resp, addr, M, data, wts, rts)$
Req_M	$p2c$.enq$(owner, Req, addr, _, _, _, _)$
let $(id, type, addr, pts) = c2pRq.get\_msg()$	$busy := True$
let $(state, data, busy, owner, wts, rts) = L2\ [addr]$	
condition: $state = M \land \neg\, busy$	
WriteBackResp	$c2pRp$.deq()
let $(id, addr, data, wts, rts) = c2pRp.get\_msg()$	$state := S$
let $(state, l2data, busy, owner, l2wts, l2rts) = L2\ [addr]$	$l2data := data$
	$busy := False$
	$l2wts := wts$
	$l2rts := rts$

5.2.3 Protocol Specification

We now formally define the core algorithm of the Tardis protocol. The state transition rules for L1 and L2 caches are summarized in Table 5.2 and Table 5.3 respectively, with the timestamps related rules in Tardis highlighted in red. For all rules where a message is enqueued to a buffer, the rule can only fire if the buffer is not full.

Specifically, the following six transition rules may fire in an L1 cache.

1. LoadHit. LoadHit can fire if the requested cacheline is in *M* or *S* state and the lease has not expired. If it is in the *M* state, then *rts* is updated to reflect the latest load timestamp. The *pts* returned to the processor is the maximum of the request's *pts* and the cacheline's *wts*.

2. StoreHit. StoreHit can only fire if the requested cacheline is in the *M* state in the L1 cache. Both *wts* and *rts* are updated to the timestamp of the store operation (pts'), which is no less than the request's *pts*. If the cacheline is clean, then pts' is also no less than the cacheline's $rts + 1$. If the cacheline is dirty, however, pts' only needs to be no less than the cacheline's *rts*.

3. L1Miss. If neither *LoadHit* nor *StoreHit* can fire for a request and the cacheline is not busy, it is an L1 miss and the request (*GetS* or *GetM*) is forwarded to the L2 cache. The cacheline is then marked as busy to prevent sending duplicate requests.

4. L2Resp. A response from L2 sets all the fields in the L1 cacheline. The *busy* flag is reset to *False*. Note that after L2Resp fires, a LoadHit or StoreHit may be able to fire immediately afterwards.

5. Downgrade. A cacheline in the *M* or *S* states may downgrade if the cacheline is not busy and *LoadHit* and *StoreHit* cannot fire. For *M* to *S* or *M* to *I* downgrade, the cacheline should be written back to the L2 in a *WBRp* message. *S* to *I* downgrade, however, is silent and no message is sent.

6. WriteBackReq. When a cacheline in *M* state receives a write back request, the cacheline is returned to L2 in a *WBRp* message and the L1 state becomes *S*. If the requested cacheline is in *S* or *I* state, the request is simply ignored. This corresponds to the case where the line self downgrades (e.g., due to cache eviction) after the write back request (*WBRq*) is sent from the L2.

The following four rules may fire in the L2 cache.

1. ShReq_S. When a cacheline in the *S* state receives a shared request (i.e., *GetS*), both the *rts* and the returned *pts* are set to pts', which can be any timestamp greater than or equal to the current *rts* and *pts*. The pts' indicates the end of the lease for the cacheline. The cacheline may be loaded at any logical time between *wts* and pts'. A *ToS* message is returned to the requesting L1.

2. ExReq_S. When a cacheline in the *S* state receives an exclusive request (i.e., *GetM*), the cacheline is instantly returned in a *ToM* message. Unlike in a directory protocol, *no* invalidations are sent to the sharers. The sharing cores may still load their local copies of the cacheline but such loads have smaller timestamps than the store from the new owner core.

3. Req_M. When a cacheline in *M* state receives a request and is not busy, a write back request (i.e., *WBRq*) is sent to the current owner. *busy* is set to *True* to prevent sending duplicate *WBRq* requests.

4. WriteBackResp. Upon receiving a write back response (i.e., *WBRp*), data and timestamps are written to the L2 cacheline. The *state* becomes *S* and *busy* is reset to *False*.

5.3 Proof of Correctness

We now prove the correctness of the Tardis protocol specified in Section 5.2.3 by proving that it strictly follows sequential consistency. We first give the definition of sequential consistency in Section 5.3.1 and then introduce the basic lemma (cf. Section 5.3.2) and timestamp lemmas (cf. Section 5.3.3) that are used for the correctness proof.

Most of the lemmas and theorems in the rest of the paper are proven through induction. For each lemma or theorem, we first prove that it is true for the initial system state (base case) and then prove that it is still true after any possible state transition assuming that it was true before the transition.

In the initial system state, all the L1 cachelines are in the *I* state, all the L2 cachelines are in the *S* state and all the network buffers are empty. For all cachelines in L1 or L2, $wts = rts = 0$ and $busy = False$. Requests from the cores may exist in the *mRq* buffers. For ease of discussion, we assume that each initial value in L2 was set before the system starts at timestamp 0 through a store operation.

5.3.1 Sequential Consistency

Sequential consistency has already been defined in Section 3.2. We repeat the definition here.

Definition 2 (Sequential Consistency). *An execution of a program is sequentially consistent iff*

Rule 1: $\forall X, Y \in \{Ld, St\}$ *from the same processor,* $X <_p Y \Rightarrow X <_m Y$.

Rule 2: *Value of* $L(a)$ = *Value of* $Max_{<m}\{S(a) | S(a) <_m L(a)\}$, *where* $L(a)$ *and* $S(a)$ *are a load and a store to address a, respectively, and* $Max_{<m}$ *selects the most recent operation in the global memory order.*

In Tardis, the global memory order of sequential consistency is expressed using timestamps. Specifically, Theorem 1 states the invariants in Tardis that correspond to the two rules of sequential consistency. Here, we use $<_{ts}$ and $<_{pt}$ to represent (logical) timestamp order that is assigned by Tardis and physical time order that represents the order of events, respectively.

Theorem 1 (SC on Tardis). *An execution on Tardis has the following invariants.*

Invariant 1: *Value of $L(a)$ = Value of $Max_{<ts}\{S(a)|S(a) \leq_{ts} L(a)\}$.*

Invariant 2: $\forall S_1(a), S_2(a), S_1(a) \neq_{ts} S_2(a)$.

Invariant 3: $\forall S(a), L(a), S(a) =_{ts} L(a) \Rightarrow S(a) <_{pt} L(a)$.

Theorem 1 will be proved later in this section. However, Theorem 1 itself is not enough to guarantee sequential consistency; we also need the processor model described in Definition 3. The processor should commit instructions in the program order, which implies physical time order and monotonically increasing timestamp order. Both in-order and out-of-order processors fit this model.

Definition 3 (In-order Commit Processor). $\forall X, Y \in \{Ld, St\}$ *from the same processor,* $X <_p Y \Rightarrow X \leq_{ts} Y \wedge X <_{pt} Y$.

Now we prove that given Theorem 1 and our processor model, an execution obeys sequential consistency per Definition 1. We first introduce the following definition of the global memory order in Tardis.

Definition 4 (Global Memory Order in Tardis).

$$X <_m Y \triangleq X <_{ts} Y \vee (X =_{ts} Y \wedge X <_{pt} Y)$$

Theorem 2. *Tardis with in-order commit processors implements Sequential Consistency.*

Proof. According to Definitions 2 and 3, $X <_p Y \Rightarrow X \leq_{ts} Y \wedge X <_{pt} Y \Rightarrow X <_m Y$. So Rule 1 in Definition 1 is obeyed. We now prove that Rule 2 is always true.

$S(a) \leq_{ts} L(a) \Rightarrow S(a) <_{ts} L(a) \vee S(a) =_{ts} L(a)$. By Invariant 3 in Theorem 1, this implies $S(a) \leq_{ts} L(a) \Rightarrow S(a) <_{ts} L(a) \vee (S(a) =_{ts} L(a) \wedge S(a) <_{pt} L(a))$. Thus, from Definition 4, $S(a) \leq_{ts} L(a) \Rightarrow S(a) <_m L(a)$. We also have $S(a) <_m L(a) \Rightarrow S(a) \leq_{ts} L(a)$ from Definition 4. So $\{S(a)|S(a) \leq_{ts} L(a)\} = \{S(a)|S(a) <_m L(a)\}$. According to Invariant 2 in Theorem 1, all the elements in $\{S(a)|S(a) <_m L(a)\}$ have different timestamps, which means $<_m$ and $<_{ts}$ indicate the same ordering. Therefore, $Max_{<ts}\{S(a)|S(a) \leq_{ts} L(a)\} = Max_{<m}\{S(a)|S(a) \leq_{ts} L(a)\}$. So Rule 2 is true following Invariant 1 in Theorem 1. □

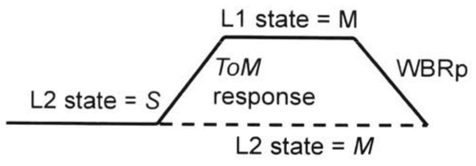


Figure 5-2: **A Visualization of Lemma 1** – Exactly one master block exists for an address.

In the following two sections, we focus on the proof of Theorem 1.

5.3.2 Basic Lemma

We first give the definition of a master block for ease of discussion.

Definition 5 (Clean Block). *A master block can be an L2 cacheline in S state, or an L1 cacheline in M state, or a ToM or WBRp message in a network buffer.*

Lemma 1. *∀ address a, exactly one master block exists for the address.*

The basic lemma is an invariant about the cacheline states and the messages in network buffers. No timestamps are involved.

A visualization of Lemma 1 is shown in Figure 5-2 where a solid line represents a clean block. When the L2 state for an address is *S*, no L1 can have that address in *M* state, and no *ToM* and *WBRp* may exist. Otherwise if the L2 state is *M*, either a *ToM* response exists, or an L1 has the address in *M* state, or a *WBRp* exists. Intuitively, Lemma 1 says only one block in the system can represent the latest data value.

Lemma 1 Proof. For the base case, the lemma is trivially true since the network and all L1s are empty; each block in the L2 is in *S* state and therefore is a master block. We now consider all the possible transition rules that may create a new master block.

Only the *ExReq_S* rule can create a *ToM* response. However, the rule changes the state of the L2 cacheline from *S* to *M* and thus removes a master block. So the number of clean blocks for that address remains one. Only the *L2Resp* rule can change an L1 cacheline state to *M*. However, it removes a *ToM* response from the *p2c* buffer. Both *Downgrade* and *WriteBackReq* can enqueue *WBRp* messages and both will change the L1 cacheline state from *M* to *S* or *I*. Only *WriteBackResp* changes the L2 cacheline to *S* state but it also dequeues a *WBRp* from the buffer.

In all of these transitions, a master block is created and another one is removed. By the induction hypothesis, exactly one master block per address exists before the current transition, and there still remains one master block for the address after the transition. For other transitions not listed above, no master block can be created or consumed so at most one master block per address exists after any transition, proving the lemma. □

5.3.3 Timestamp Lemmas

Lemma 2. *At the current physical time, a master block has the following invariants.*

Invariant 1 *Its rts is no less than the rts of all the other blocks (i.e., a cacheline in L1 or L2, or a network message in a buffer) with respect to the same address.*

Invariant 2 *Till the current physical time, no store has happened to the address at timestamp ts such that $ts > rts$.*

Proof. We prove the lemma by induction on the transition sequence. For the base case, only one block exists per address so Invariant 1 is true. All the stores thus far happened at timestamp 0 which equals the *rts* of all the master blocks, so Invariant 2 is also true.

According to Lemma 1, for an address, exactly one master block exists. By the induction hypothesis, if no timestamp changes and no master block is generated, Invariant 1 is still true after the transition. Therefore, we only consider rules where a timestamp changes or a new master block is generated. By the transition rules, a timestamp can only be changed if the block is an L2 cacheline in the *S* state or an L1 cacheline in the *M* state. In both cases the block is a master block. After the transition, the *rts* of the master block can only increase and is still no less than the *rts* of other cachelines with the same address, and thus Invariant 1 holds. In all rules where a new master block is created, it copies the *wts* and *rts* of the previous clean block to the new block. Therefore, Invariant 1 also holds in this case.

Similarly, by the induction hypothesis, Invariant 2 is true after the transition if no store happens, no *rts* changes and no master block is generated. We consider the three cases separately. A store can happen only in the *StoreHit* rule, which changes both *wts* and *rts* of a master block to $\max(pts, rts+1)$. For the stored cacheline, since no store has happened with timestamp *ts* such that $ts > old\_rts$ (induction hypothesis), after the transition, no store, including the current one, has happened with timestamp *ts* such that $ts > \max(pts, old\_rts+1) > old\_rts$. For an *rts* change, all rules touching it (i.e., LoadHit, StoreHit, ShReq_S) can only increase its value. Therefore, if no store exists after the previous *rts* in logical timestamp order, it cannot exist after the new *rts* either. Finally, in cases where a new master block is generated, the new master block has the same *rts* as the previous one, and no store exists after this *rts*. □

Lemma 3. *For any block B in a cache or a message (WBRp, ToS and ToM), the data value associated with the block is created by a store St such that $St.ts \leq B.wts$, and St has happened before the current physical time. Furthermore, no other store St' has happened such that $St.ts < St'.ts \leq B.rts$. $St.ts$ is the timestamp of the store St and $B.wts$ ($B.rts$) is the wts (rts) of block B.*

Proof. We prove the lemma by induction on the transition sequence. We have assumed that each value in the L2 is set by an initial store before the execution starts. Therefore, for the base case, each block is created by the corresponding initial store which happened before the current physical time. The initial store is also the only store happened to each block. Therefore, the hypothesis is true.

We first prove part one of the lemma, that after a transition, for each block, there exists a store St that creates the data of the block such that $St.ts \leq wts$ and St happened before the current physical time. Consider the case where the data of a block does not change or is copied from another block. By the induction hypothesis, if St exists before the transition, it must exist after the transition as well, since *wts* never decreases during data copying (actually it never changes), and physical time always advances. The only transition that changes the data of a block is *StoreHit*. In this case, a store will create a new block with *wts* equal to the timestamp of the store, and the St has happened in physical time starting from the next transition.

We now prove part two of the lemma, that for any block B, no store $St' \neq St$ has happened such that $St.ts < St'.ts \leq B.rts$. By the induction hypothesis, for the current transition, if no *data* or *rts* is changed in any block or if a block copies *data* and *rts* from an existing block, then the hypothesis is still true after the transition. The only cases in which the hypothesis may be violated are when the current transition changes *rts* or *data* for some block, which is only possible for *LoadHit*, *StoreHit* and *ShReq_S*.

For *LoadHit*, if the cacheline is in the *S* state, then *rts* remains unchanged. Otherwise, the cacheline must be a master block (i.e., a cacheline in M state in an L1), in which case *rts* is increased. Similarly, *ShReq_S* increases the *rts* and the cacheline must be a master block as well. By Invariant 2 in Lemma 2, no store has happened to a master block with timestamp greater than *rts*. And thus after the *rts* is increased, no store can have happened with timestamp between the old *rts* and the new *rts*. Namely, no store St' could have happened such that $old\_rts < St' < new\_rts$. By the induction hypothesis, we also have that no store St' could have happened such that $St.ts < St'.ts \leq old\_rts$. These two inequalities together prove the hypothesis.

For *StoreHit*, both *rts* and *data* are modified. For the stored cacheline, after the transition, $St.ts$

= *wts* = *rts* = *max(pts, old_rts* + 1). Thus, no St' can exist between *wts* and *rts* since they are equal. For all the other cachelines with the same address, by Invariant 1 in Lemma 2, their *rts* is no greater than the old *rts* of the stored cacheline and is thus smaller than the timestamp of the current store. By the induction hypothesis, no store St' exists for those blocks after the transition. Thus, in the overall system, no such store St' can exist for any block, proving the lemma. □

Finally, we prove Theorem 1.

Theorem 1 Proof. We will prove the three invariants in Theorem 1 individually.

According to Lemma 3, for each $L(a)$, the loaded data is provided by an $S(a)$ and no other store $S'(a)$ has happened between the timestamp of $S(a)$ and the current *rts* of the cacheline. Therefore, no $S'(a)$ has happened between the timestamp of $S(a)$ and the timestamp of the load, which is no greater than *rts* by the transition rules. Therefore, Invariant 1 in Theorem 1 is true.

By the transition rules, a new store can only happen to a master block and the timestamp of the store is $\max(pts, rts+1)$. By Invariant 2 in Lemma 2, for a clean block at the current physical time, no store to the same address has happened with timestamp greater than the old *rts* of the cacheline. Therefore, for each new store, no store to the same address so far has the same timestamp as the new store, because the new store's timestamp is strictly greater than the old *rts*. And thus no two stores to the same address may have the same timestamp, proving Invariant 2.

Finally, we prove Invariant 3. If $S(a) =_{ts} L(a)$, by Invariant 1 in Theorem 1, $L(a)$ returns the data stored by $S(a)$. Then, by Lemma 3, the store $S(a)$ must have happened before $L(a)$ in the physical time. □

5.4 Deadlock and Livelock Freedom

In this section, we prove that the Tardis protocol specified in Section 5.2 is both deadlock-free (Section 5.4.1) and livelock-free (Section 5.4.2).

5.4.1 Deadlock Freedom

Theorem 3 (Deadlock Freedom). *After any sequence of transitions, if there is a pending request from any core, then at least one transition rule (other than the Downgrade rule) can fire.*

Before proving the theorem, we first introduce and prove several lemmas.

Lemma 4. *If an L1 cacheline is busy, either a GetS or GetM request with the same address exists in its c2pRq buffer or a ToS or ToM response with the same address exists in its p2c buffer.*

Proof. This can be proven through induction on the transition sequence. In the base case, all the L1 cachelines are non-busy and the hypothesis is true. An L1 cacheline can only become busy through the *L1Miss* rule, which enqueues a request to its *c2pRq* buffer. A request can only be dequeued from *c2pRq* through the *ShReq_S* or *ExReq_S* rule, which enqueues a response into the same L1's *p2c* buffer. Finally, whenever a message is dequeued from the *p2c* buffer (*L2Resp* rule), the L1 cacheline becomes non-busy, proving the lemma. □

Lemma 5. *If an L2 cacheline is busy, the cacheline must be in state M.*

Proof. This lemma can be proven by induction on the transition sequence. For the base case, no cachelines are busy and the hypothesis is true. Only *Req_M* makes an L2 cacheline busy but the cacheline must be in the *M* state. Only *WriteBackResp* downgrades an L2 cacheline from the *M* state but it also makes the cacheline non-busy. □

Lemma 6. *For an L2 cacheline in the M state, the id of the master block for the address equals the owner of the L2 cacheline.*

Proof. According to Lemma 1, exactly one master block exists for the address. If the L2 state is *M*, the master block can be a *ToM* response, an L1 cacheline in the *M* state, or a *WBRp*. We prove the lemma by induction on the transition sequence.

The base case is true since no L2 cachelines are in the *M* state. We only need to consider cases wherein a master block is created. When *ToM* is created (*ExReq_S* rule), its *id* equals the owner in the L2 cacheline. When an L1 cacheline in the *M* state is created (*L2Resp* rule), its *id* equals the *id* of the *ToM* response. When a *WBRp* is created (*WriteBackReq* or *Downgrade* rule), its *id* equals the *id* of the L1 cacheline. By the induction hypothesis, the *id* of a newly created master block always equals the *owner* in the L2 cacheline, which does not change as long as the L2 cacheline is in the *M* state. □

Lemma 7. *For a busy cacheline in L2, either a WBRq or a WBRp exists for the address with id matching the owner of the L2 cacheline.*

Proof. We prove the lemma by induction on the transition sequence. For the base case, no cacheline is busy and thus the hypothesis is true. We only need to consider the cases where an L2 cacheline is busy after the current transition, i.e., $\neg busy \Rightarrow busy$ and $busy \Rightarrow busy$.

Only the *Req_M* rule can cause a $\neg busy \Rightarrow busy$ transition and the rule enqueues a *WBRq* into *p2c* with *id* matching the *owner* and therefore the hypothesis is true.

For $busy \Rightarrow busy$, the lemma can only be violated if a *WBRq* or *WBRp* with a matching *id* is dequeued. However, when a *WBRp* is dequeued, the cacheline becomes non-busy in L2 (*WriteBackResp* rule). If a *WBRq* is dequeued and the L1 cacheline is in the *M* state, a *WBRp* is created with a matching *id*. So the only case to consider is when the *WBRq* with a matching *id* is dequeued, and the L1 cacheline is in the *S* or *I* states, and no other *WBRq* exists in the same *p2c* buffer and no *WBRp* exists in the *c2pRp* buffer. We now use contradiction to show such a scenario can never happen.

The L2 cacheline can only become *busy* by sending a *WBRq*. The fact that the dequeued *WBRq* is the only *WBRq* in the *p2c* means that the L2 cacheline has been busy since the dequeued *WBRq* was sent (otherwise another *WBRq* will be sent when the L2 cacheline becomes busy again). Since *p2c* is a FIFO, when the *WBRq* is dequeued, the messages in the *p2c* must be sent after the *WBRq* was sent. By transition rules, the L2 cacheline cannot send *ToM* with the same address while being busy, so no *ToM* with the same address as *WBRq* may exist in the *p2c* buffer when the *WBRq* message dequeues. As a result, no master block exists with $id = owner$. Then, by Lemma 6, no master block exists for the address (L2 is in the *M* state because of Lemma 5), which contradicts Lemma 1. □

Finally, we can prove Theorem 3.

Theorem 3 Proof. If any message exists in the *c2pRp* buffer, the *WriteBackResp* rule can fire. Consider the case where no message exists in *c2pRp* buffer. If any message exists in the *p2c* buffer's head, the *L2Resp* rule can fire, or the *WriteBackReq*, *LoadHit* or *StoreHit* rule can fire. For the theorem to be violated, no messages can exist in the *c2pRp* or *p2c* buffer. Then, according to Lemma 7, all cachelines in L2 are non-busy.

Now consider the case when no message exists in *c2pRp* buffer or *p2c* buffer and a *GetS* or *GetM* request exists in *c2pRq* for an L1 cache. Since the L2 is not busy, one of *ShReq_S*, *ExReq_S* and *Req_M* can fire, which enqueues a message into the *p2c* buffer.

Consider the last case where there is no message in any network buffer. By Lemma 4, all L1 cachelines are non-busy. By the hypothesis, there must be a request in *mRq* for some core. Now if the request is a hit, the corresponding hit rule (*LoadHit* or *StoreHit*) can fire. Otherwise, the *L1Miss* rule can fire, sending a message to *c2pRq*. □

5.4.2 Livelock Freedom

Even though the Tardis protocol correctly follows sequential consistency and is deadlock-free, livelock may still occur if the protocol is not well designed. For example, for an L1 miss, the *Downgrade* rule may fire immediately after the *L2Resp* but before any *LoadHit* or *StoreHit* rule fires. As a result, the *L1Miss* needs to be fired again but the *Downgrade* always happens after the response comes back, leading to livelock. We avoid this possibility by only allowing *Downgrade* to fire when neither *LoadHit* nor *StoreHit* can fire.

To rigorously prove livelock freedom, we need to guarantee that some transition rule should eventually make forward progress and no transition rule can make backward progress. Specifically, the problem in formalized in Theorem 4.

Theorem 4. *After any sequence of transitions, if there exists a pending request from any core, then within a finite number of transitions, some request at some core will dequeue.*

Table 5.4: **Lattice for a Request** – For a load request, $L1.miss \triangleq (L1.state = I \vee (L1.state = S \wedge pts > L1.rts))$. For a store request, $L1.miss \triangleq (L1.state < M)$. *bufferName_exist* means a message exists in the buffer and *bufferName_rdy* means that the message is the head of the buffer. *bufferName_rdy* implies *bufferName_exist*.

1	L1.miss ∧ ¬L1.busy
2	L1.miss ∧ L1.busy ∧ c2pRq_exist ∧ ¬ c2pRq_rdy
3	L1.miss ∧ L1.busy ∧ c2pRq_rdy ∧ L2.state = *M* ∧ ¬L2.busy
4	L1.miss ∧ L1.busy ∧ c2pRq_rdy ∧ L2.state = *M* ∧ L2.busy ∧ p2cRq_exist ∧ ¬p2cRq_rdy
5	L1.miss ∧ L1.busy ∧ c2pRq_rdy ∧ L2.state = *M* ∧ L2.busy ∧ p2cRq_rdy ∧ ownerL1.state = *M*
6	L1.miss ∧ L1.busy ∧ c2pRq_rdy ∧ L2.state = *M* ∧ L2.busy ∧ p2cRq_rdy ∧ ownerL1.state < *M*
7	L1.miss ∧ L1.busy ∧ c2pRq_rdy ∧ L2.state = *M* ∧ L2.busy ∧ ¬p2cRq_exist
8	L1.miss ∧ L1.busy ∧ c2pRq_rdy ∧ L2.state = *S*
9	L1.miss ∧ L1.busy ∧ p2cRp_exist ∧ ¬p2cRp_rdy
10	L1.miss ∧ L1.busy ∧ p2cRp_rdy
11	¬L1.miss

In order to prove the theorem, we will show that for every transition rule, at least one request will make forward progress and move one step towards the end of the request and at the same time no other request makes backward progress; or if no request makes forward or backward progress for the transition, we show that such transitions can only be fired a finite number of times. Specifically, we define forward progress as a lattice of system states where each request in *mRq* (load or store) has its own lattice. Table 5.4 shows the lattice for a request where the lower parts in the lattice correspond to the states with more forward progress. We will prove livelock freedom by showing that for any state transition, a request either moves down the lattice (making forward progress) or stays at the current position but never moves upwards. Moreover, transitions which keep the state of every request staying at the same position in the lattice can only occur a finite number of times. Specifically, we will prove the following lemma.

Lemma 8. *For a state transition except Downgrade, WriteBackReq and WriteBackResp, either a request dequeues from the mRq or at least one request will move down its lattice. For all the state transitions, no request will move up its lattice. Further, the system can only fire Downgrade, WriteBackReq and WriteBackResp for a finite number of times without firing other transitions.*

We need Lemmas 9 to 16 in order to prove Lemma 8.

Lemma 9. *If an L1 cacheline is busy, then exactly one request (GetS or GetM in c2pRq) or response (ToS or ToM in p2c) exists for the address. If the L1 cacheline is non-busy, then no request or response can exist in its c2pRq and p2c.*

Proof. This lemma is a stronger lemma than Lemma 4. We prove this by the induction on the transition sequence. For the base case, all L1s are empty and no message exists and thus the lemma is true.

We only need to consider the cases where the busy flag changes or any request or response is enqueued or dequeued. Only the *L1Miss*, *L2Resp*, *ShReq_S* and *ExReq_S* rules need to be considered.

For *L1Miss*, a request is enqueued to *c2pRq* and the L1 cacheline becomes busy. For *L2Resp*, a response is dequeued and the L1 cacheline becomes non-busy. For *ShReq_S* and *ExReq_S*, a request is dequeued but a response in enqueued. By the induction hypothesis, after the current transition, the hypothesis is still true for all the cases above, proving the lemma. □

Lemma 10. *If an L1 cacheline is busy, there must exist a request at the head of the mRq buffer for the address and the request misses in the L1.*

Proof. For the base case, all L1 cachelines are non-busy and the lemma is true.

We consider cases where the L1 cacheline is busy after the transition. Only *L1Miss* can make an L1 cacheline busy from non-busy and the rule requires a request to be waiting at the head of the *mRq* buffer. If the L1 cacheline stays busy, then no rule can remove the request from the *mRq* buffer. By the induction hypothesis, the lemma is true after any transition. □

Lemma 11. *If an L2 cacheline is busy, there must exist a request with the same address at the head of the c2pRq buffer in L2.*

Proof. The proof follows the same structure as the previous proof for Lemma 10. □

Lemma 12. *For a memory request in a c2pRq buffer, its type and pts equal the type and pts of a pending request to the same address at the head of the mRq at the L1 cache.*

Proof. By Lemmas 9 and 10, the L1 cacheline with the same address must be *busy* and a pending request exists at the head of the *mRq* buffer. Only the *L1Miss* rule sets the *type* and *pts* of a memory request in a *c2pRq* buffer and they equal the *type* and *pts* of the request at the head of *mRq*. □

Lemma 13. *For a memory response in a p2c buffer (i.e., ToM or ToS), its type equals the type of a pending mRq request to the same address at the L1 cache; if type = S, its rts is no less than the pts of the pending mRq request.*

Proof. Similar to the proof of Lemma 12, a request with the same address must exist in *mRq* of the corresponding core. Only the *ShReq_S* and *ExReq_S* rules set the *type* and *rts* of the response, and *type* equals the *type* of a memory request and if *type* = *S*, *rts* is no less than the memory request. Then the lemma is true by Lemma 12. □

Lemma 14. *When the L2Resp rule fires, a request with the same address at the head of mRq will transition from an L1 miss to an L1 hit.*

Proof. Before the transition of *L2Resp*, the L1 cacheline is busy, and a response is at the head of the *p2c* buffer. By Lemma 13, if the pending processor request has *type* = *M*, then the memory response also has *type* = *M* and thus it is an L1 hit. If the pending processor has *type* = *S*, also by Lemma 13, the memory response has *type* = *S* and the *rts* of the response is no less than the *pts* of the pending request. Therefore, LoadHit can also fire. □

Lemma 15 (Coverage)**.** *The union of all the entries in Table 5.4 is True.*

Proof. By Lemma 4, if *L1.busy* we can prove that $c2pRq\_exist \vee p2cRp\_exist \Rightarrow True$.

Then, it becomes obvious that the union of all the entries is true. □

Lemma 16 (Mutually Exclusive)**.** *The intersection of any two entries in Table 5.4 is False.*

Proof. For most pairs of entries, we can trivially check that the intersection is *False*. The only tricky cases are the intersection of entry 9 or 10 with an entry from 3 to 8. These cases can be proven *False* using Lemma 9, which implies that $c2pRq\_exist \wedge p2cRp\_exist \Rightarrow False$. □

Now we can prove Lemma 8.

Lemma 8 Proof. We need to prove two goals. First, for each transition rule except *Downgrade*, *WriteBackReq* and *WriteBackResp*, at least one request will dequeue or move down the lattice. Second, for all transition rules no request will move up the lattice.

We first prove that a transition with respect to address a_1 never moves a request with address a_2 ($\neq a_1$) up its lattice. The only possible way that the transition affects the request with a_2 is by dequeuing from a buffer, which may make a request with a_2 being the head of the buffer and thus becomes ready. However, this can only move the request with a_2 down the lattice.

Also note that each processor can only serve one request per address at a time, because the *mRq* is a FIFO. Therefore, for the second goal we only need to prove that requests with the same address in other processors do not move up the lattice. We prove both goals for each transition rule.

For *LoadHit* and *StoreHit*, a request always dequeues from the *mRq* and the lemma is satisfied.

For the *L1Miss* rule, before the transition, a request must exist and be in entry 1 in Table 5.4. Since $busy = True$ after the transition, it must move down the lattice to one of entries from 2 to 10. Since the L1 cacheline state does not change, no other requests in other processors having the same address move in their lattice.

For the *L2Resp* rule, according to Lemma 14, a request will move from *L1.miss* to *L1.hit*. In the lattice, this corresponds to moving from entry 10 to entry 11, which is a forward movement. For another request to the same address, the only entries that might be affected are entry 4, 5 and 6. However, since *p2c* is a FIFO and the response is ready in the *p2c* buffer before the transition, no *WBRq* can be ready in this *p2c* buffer for other requests with the same address. Therefore, no other requests can be in entry 5 or 6. If another request is in entries 4, the transition removes the response from the *p2c* and this may make the *WBRq* ready in *p2c* and thus the request moves down the lattice. In all cases, no other requests move up the lattice.

For the *ShReq_S* or *ExReq_S* rule to fire, there exists a request in the *c2pRq* buffer, which means the address must be busy in the corresponding L1 (Lemma 9) and thus a request exists in its *mRq* and misses the L1 (Lemma 10). This request, therefore, must be in entry 8 in Table 5.4. The transition will dequeue the request and enqueue a response to *p2c* and thus moves the request down to entry 9 or 10. For all the other requests with the same address, they cannot be ready in the *c2pRq* buffer since the current request blocks them, and thus they are not in entries 3 to 8 in the lattice. For the other entries, they can only possibly be affected by the transition if the current request is dequeued and one of them becomes ready. This, however, only moves the request down the lattice.

The *Req_M* rule can only fire if a request is ready in *c2pRq* and the L2 is in the *M* state. According to Lemma 9 and Lemma 10, there exists a request in one *mRq* that is in entry 3 in a table. After the transition, this request will move to entries 4 or 5 or 6 and thus down the lattice. For all the other requests, similar to the discussion of *ShReq_S* and *ExReq_S*, they either stay in the same entry or move down the lattice.

Finally, we talk about the *Downgrade*, *WriteBackReq* and *WriteBackResp* rules. The *Downgrade* rule can only fire when the L1 cacheline is non-busy, corresponding to entry 1 and 11 if the request is from the same L1 as the cacheline being downgraded. Entry 1 cannot move up since it is the first entry. If a request is in entry 11, since it is an L1 hit now, the *Downgrade* rule does not fire. For a request from a different L1, the *Downgrade* rule may affect entry 5 and 6. However, it can only move the request from entry 5 to 6 rather than the opposite direction.

For the *WriteBackReq* rule, if the L1 cacheline is in the *S* state, then nothing changes but a message is dequeued from the *p2c* buffer, which can only move other requests down the lattice. If the L1 cacheline is in the *M* state, then if a request to the same address exists in the current L1, the request must be a hit and thus *WirteBackReq* cannot fire. For requests from other L1s, they can only be affected if they are in entry 4. Then, the current transition can only move them down the lattice.

For the *WriteBackResp* rule, the L2 cacheline moves from the *M* to the *S* state. All the other requests can only move down their lattice due to this transition.

Now, we prove that *Downgrade*, *WriteBackReq* and *WriteBackResp* can only fire a finite number of times without other transitions being fired. Each time *Downgrade* is fired, an L1 cacheline's state goes down. Since there are only a finite number of L1 cachelines and a finite number of states, *Downgrade* can only be fired a finite number of times. Similarly, each *WriteBackReq* transition consumes a *WBRq* message, which can only be replenished by the *Req_M* rule. And each *WriteBackResp* transition consumes a *WBRp*, which is replenished by *Downgrade* and *WriteBackReq* and thus only has finite count. □

Finally, we can prove Theorem 4.

Theorem 4 Proof. If there exists a pending request from any processor, by Lemma 8, some pending request will eventually dequeue or move down the lattice, which only has a finite number of states. For a finite number of processors, since the *mRq* is a FIFO, only a finite number of pending requests can exist. Therefore, some pending request will eventually reach the end of the lattice and dequeue, proving the theorem. □

5.5 Other Consistency Models

In Section 7.3, we proved that the Tardis protocol follows sequential consistency. In this section, we show that Tardis also supports relaxed consistency models with simple changes. We use *Total Store Order* (TSO) as an example, which is defined in Definition 6 ([132]). We only prove for correctness, but not deadlock and livelock freedom, since the proof is similar to the sequential consistency case.

Definition 6 (TSO). *An execution of a program follows TSO iff*

Rule 1: $\forall$ *address a, b from the same processor,* $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$, $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$, $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$.

Rule 2: *Value of L(a) = Value of* $Max_{<m}\{S(a)|S(a) <_m L(a) \vee S(a) <_p L(a)\}$

Rule 3: $X <_p Fence \Rightarrow X <_m Fence$, $Fence <_p X \Rightarrow Fence <_m X$

Different from the processor model for SC (Definition 3), TSO requires a slightly different model for the processor. We define the following processor model for the TSO consistency model.

Definition 7 (TSO Processor). $\forall$ *address a and b,* $L(a) <_p L(b) \Rightarrow L(a) \leq_{ts} L(b), L(a) <_p S(b) \Rightarrow L(a) \leq_{ts} S(b), S(a) <_p S(b) \Rightarrow S(a) \leq_{ts} S(b)$. $\forall X, Y \in \{Ld, St, Fence\}$, $X <_p Y \Rightarrow X <_{pt} Y$

Furthermore, the rules for the memory system (Table 5.2) also need some changes to fully model TSO. Specifically, we change the LoadHit rule to be the one shown in Table 5.5. When a cacheline is dirty, a load to it can be performed at a timestamp smaller than the current *wts* of the cacheline; therefore, the *pts* does not have to increase.

Table 5.5: **LoadHit Rule for TSO** – The change for TSO is underlined.

Rules and Condition	**Action**
LoadHit **let** (*type*, *addr*, _, *pts*) = *mRq.get_msg*() **let** (*state*, *data*, *busy*, *dirty*, *wts*, *rts*) = *L1* [*addr*] **condition:** $\neg$ *busy* $\wedge$ *type* = *S* $\wedge$ (*state* = *M* $\vee$ (*state* = *S* $\wedge$ *pts* $\leq$ *rts*))	*mRq*.deq() *mRp*.enq(*type*, *addr*, *data*, max(*pts*, *wts*)) **If (*dirty*)** **_mRp_.enq(_type_, _addr_, _data_, _pts_)** **Else** *mRp*.enq(*type*, *addr*, *data*, max(*pts*, *wts*)) **If** (*state* = *M*) *rts* := max(*pts*, *rts*)

Using similar proof techniques as in SC, it is straightforward to show that both Rules 1 and 3 in Definition 6 are obeyed. We show a proof of Rule 2 in this section.

Before proving Rule 2, we need the following lemmas.

Lemma 17. *The wts and rts of the master block of an address never decrease.*

Proof. This lemma is an extension of Lemma 1 which says only one master block can exist for an address. Lemma 17 further states that the timestamps of the block should never decrease. Proving the lemma is very straightforward by induction. All rules that create a master block (i.e., L2Resp, WriteBackReq, ExReq_S, WriteBackResp) copy the *wts* and *rts* from the previous master block, and thus do not decrease the timestamps. All rules that modify the timestamps of a master block (i.e., LoadHit, StoreHit, ShReq_S) will only increase a timestamp but never decrease them. □

Lemma 18. *The wts and rts of cachelines with the same address never decrease within a particular L1.*

Proof. A cacheline within an L1 may or may not be a master block. If it is a master block, it must be in modified state, otherwise, it must be in shared state. We show that between any pair of cachelines (A_1, A_2) that are adjacent in physical time with the same address in the same L1, the cacheline that exists later in time (A_2) has greater or equal *wts* and *rts* compared to the preceding cacheline (A_1), namely $A_1.wts \leq A_2.wts$ and $A_1.rts \leq A_2.rts$.

First, if both A_1 and A_2 are in modified states, the timestamps of A_2 are no less than the timestamps of A_1 due to Lemma 17 and therefore the Lemma is true. If A_1 is in shared state and A_2 is in modified state, then the timestamps of A_1 must be copied from a master block that exists before A_1 and therefore A_2. Therefore, A_1's timestamps must be less than or equal to A_2's timestamps. If A_2 is in shared state, then A_2 might come from a Downgrade, WriteBackReq or L2Resp rule. If it is Downgrade or WriteBackReq, then A_1 must be in modified state and $A_2.wts = A_1.wts$, $A_2.rts = A_1.rts$. Finally, if A_2 comes from the L2Resp rule, then the A_2's timestamps must be copied from a shared L2 cacheline through a ShReq_S rule. Since the *p2c* buffer is a FIFO, A_1 (either in shared or modified state) must come from a shared L2 cacheline earlier than the firing of ShReq_S that creates A_2. Therefore, the lemma is true for all possible cases. □

Now we can prove Rule 2 in Definition 6.

Theorem 5. *For each $L(a)$, Value of $L(a)$ = Value of $Max_{<m}\{S(a)|S(a) <_m L(a) \vee S(a) <_p L(a)\}$*

Proof. First, observe that Lemmas 1, 2 and 3 are still true after the changes are made for TSO. This is because all the three lemmas are about the states of cachelines in the memory system, while the changes made for TSO are only about states of the cores. For the same reason, Invariants 2 and 3 in

Theorem 1 are also true. Namely, $\forall S_1(a), S_2(a), S_1(a) \neq_{ts} S_2(a)$, and $\forall S(a), L(a), S(a) =_{ts} L(a) \Rightarrow S(a) <_{pt} L(a)$.

Given these invariants, we now prove Rule 2 by considering two cases: whether the load is to a dirty cacheline or not.

Case 1. A core i loads a non-dirty cacheline with address a. The cacheline can be in either shared or modified state, but the dirty bit is unset. In this case, the LoadHit rule is exactly the same as before. Following the proof of Theorem 1, we have *Value of* $L(a)$ = *Value of* $Max_{<ts}\{S(a) | S(a) \leq_{ts} L(a)\}$. In order to prove Rule 2 in Definition 6, we need to show that $\neg\exists\ S'(a).(S'(a) <_p L(a) \land \forall S''(a) \in \{S(a) | S(a) <_m L(a)\}.\ S''(a) <_m S'(a))$.

We prove this by contradiction. If such an $S'(a)$ exists, then we have $S'(a) <_p L(a) \land S'(a) >_{ts} L(a)$. This means $S'(a).ts >_{ts} A.wts$, where A is the cacheline with address a in core i's private cache when the load occurs. The existence of $S'(a)$ means there must have existed another L1 cacheline A' with address a in modified state earlier than A (namely, $A' <_{pt} A$) and that $A'.wts >_{ts} A.wts$. This contradicts Lemma 18.

Case 2. A core i loads a dirty cacheline with address a. In this case, the timestamp of the load does not have to be greater than or equal to the *wts* of the cacheline. Because the cacheline is dirty, it must be in modified state in the L1. According to Lemmas 2 and 3, the load returns the data from a store that has the largest timestamp of all of the stores ever occurred to the address when the load occurs. Namely, *Value of* $L(a)$ = *Value of* $Max_{<m}\{S(a)\}$. Therefore, the statement in the Lemma is obviously true. □

5.6 Main Memory

For ease of discussion, we have assumed that all the data fit in the L2 cache, which is not realistic for some shared memory systems. A multicore processor, for example, has an offchip main memory that does not contain timestamps. For these systems, the components in Table 5.6 and transition rules in Table 5.7 need to be added. And for the initial system state, all the data should be in main memory with all L2 cachelines in I state, and with $mrts = mwts = 0$.

Most of the extra components and rules are handling main memory requests and responses and the I state in the L2. However, *mwts* and *mrts* are special timestamps added to represent the largest *wts* and *rts* of the cachelines stored in the main memory. These timestamps guarantee that cachelines loaded from the main memory have proper timestamps.

Table 5.6: **System Components to Model a Main Memory.**

Component	Format	Message Types
MemRq	*MemRq.entry* = (*type*, *addr*, *data*)	*MemRq*
MemRp	*MemRp.entry* = (*addr*, *data*)	*MemRp*
Mem	*Mem* [*addr*] = (*data*)	-
mwts, *mrts*	-	-

Table 5.7: **State Transition Rules for Main Memory.**

Rules and Condition	Action
L2Miss **let** (*id*, *type*, *addr*, *pts*) = *c2pRq.get_msg*() **condition:** *L2.*[*addr*]*.state* = *I*	MemRq.enq(*S*, *addr*, _) *busy* := *True*
MemResp **let** (*addr*, *data*) = MemRp **let** (*state*, *l2data busy*, *owner*, *wts*, *rts*) = *L2.*[*addr*]	*state* := *S* *l2data* := *data* *busy* := *False* *wts* := *mwts* *rts* := *mrts*
L2Downgrade **let** (*state*, *data*, *busy*, *owner*, *wts*, *rts*) = *L2.*[*addr*] **condition:** *state* = *M* ∧ *busy* = *False*	*p2c*.enq(*owner*, *Req*, *addr*, _, _, _, _) *busy* := *True*
L2Evict **let** (*state*, *data*, *busy*, *owner*, *wts*, *rts*) = *L2.*[*addr*] **condition:** *state* = *S*	*MemRq*.enq(*M*, *addr*, *data*) *state* := *I* *mwts* := max(*wts*, *mwts*) *mrts* := max(*wts*, *mrts*)

In this section, we prove that the Tardis protocol with main memory still obeys sequential consistency (SC). Although not included, the proof for TSO is similar. The system can also be shown to be deadlock- and livelock-free using proofs similar to Section 5.4. For the SC proof, Lemma 1 needs to be changed slightly. Instead of stating that "exactly" one master block exists for an address, we change it to "at most" one master block exists for an address. Then, we just need to show that the modified Lemmas 1, 2, and 3 are true after the main memory is added.

In order to prove these lemmas, we need to prove the following Lemma 19.

Lemma 19. *If an L2 cacheline is in the I state, no master block exists for the address.*

Proof. We prove by induction on the transition sequence. The hypothesis is true for the base case since no master block exists. If an L2 cacheline moves from *S* to *I* (through the L2Evict rule), the master block (L2 cacheline in S state) is removed and no master block exists for that address. By the transition rules, while the L2 line stays in the *I* state, no master block can be created. By the induction hypothesis, if an L2 cacheline is in the *I* state after a transition, then no master block can exist for that address. □

For Lemma 1, we only need to include Lemma 19 in the original proof. For Lemmas 2 and 3, we need to show the following properties of *mwts* and *mrts*.

Lemma 20. *If an L2 cacheline with address a is in I state, then the following statements are true.*

- *mwts is greater than or equal to the wts of all the copies of blocks with address a.*
- *mrts is greater than or equal to the rts of all the copies of blocks with address a.*
- *No store has happened to the address at ts such that $ts > mrts$.*
- *The data value of the cacheline B in main memory comes from a store St such that $St.ts \leq mwts$ and St happened before the current physical time. And no other store St' has happened such that $St.ts < St'.ts \leq mrts$.*

Proof. All the statements can be easily proven by induction on the transition sequence. For $S \rightarrow I$ of an L2 cacheline, since *mwts* and *mrts* are no less than the cacheline's *wts* and *rts*, respectively, by Lemmas 2 and 3, all four statements are true after the transition.

Consider the other case where the L2 cacheline stays in *I* state. Since no master block exists (Lemma 19), the copies of the cacheline cannot change their timestamps and no store can happen. By the transition rules, the *mts* never decreases after the transition. So the hypothesis must be true after the transition.

To finish the original proof, we need to consider the final case where the L2 cacheline moves from *I* to *S* state (*MemResp* rule). In this case, *wts* and *rts* are set to *mwts* and *mrts* respectively. By Lemma 20, both Lemma 2 and Lemma 3 are true after the current transition.

□

5.7 Additional Related Work

Both model checking and formal proofs are popular in proving the correctness of cache coherence protocols. Model checking based verification [42, 94, 79, 109, 22, 23, 74, 34, 32, 46, 159, 108, 107, 75, 39, 160] is a commonly used technique, but even with several optimizations, it does not typically scale to automatically verify real-world systems.

Many other works [121, 10, 104, 26, 90] prove the correctness of a cache coherence protocol by proving invariants as we did in this paper. Our invariants are in general simpler and easier to understand than what they had, partly because Tardis is simpler than a directory coherence protocol. Finally, our proofs can be machine-checked along the lines of the proofs for a hierarchical cache coherence protocol [145, 119].

Part II

Scalable Concurrency Control

Chapter 6

Concurrency Control Scalability Study

6.1 Introduction

The number of cores on a single chip has been growing significantly. Processors with hundreds of cores are ready available in the market, and single-chip thousand-core systems may appear within a few years. The scalability of single-node, shared-memory database management systems (DBMSs) is even more important in the many-core era. But if the current DBMS technology does not adapt to this reality, all this computational power will be wasted on bottlenecks, and the extra cores will be rendered useless.

In this section, we take a peek at this future and examine what happens with transaction processing at one thousand cores. We limit our scope to concurrency control since it is one of the system components that is most difficult to scale. With hundreds of threads running in parallel, the complexity of coordinating competing accesses to data will become a major bottleneck to scalability, and will likely dwindle the gains from increased core counts.

We implemented seven concurrency control algorithms in a main memory DBMS and used a high-performance, distributed CPU simulator to scale the system to 1000 cores. Implementing a system from scratch allows us to avoid any artificial bottlenecks in existing DBMSs and instead understand the more fundamental issues in the algorithms. Previous scalability studies used existing DBMSs [77, 81, 117], but many of the legacy components of these systems do not target many-core CPUs. To the best of our knowledge, there has not been an evaluation of multiple concurrency control algorithms on a single DBMS at such large scale, prior to our work.

Our analysis shows that all algorithms fail to scale as the number of cores increases. In each case, we identify the primary bottlenecks that are independent of the DBMS implementation and argue that even state-of-the-art systems suffer from these limitations.

This chapter makes the following contributions:

- A comprehensive evaluation of the scalability of seven concurrency control schemes.
- The first evaluation of an OLTP DBMS on 1000 cores.
- Identification of bottlenecks in concurrency control schemes that are not implementation-specific.

The remainder of this section is organized as follows. We begin in Section 6.2 with an overview of the concurrency control schemes used in our evaluation. Section 6.3 describes the multicore architecture of our study. We present our analysis in Sections 6.4 and 6.5, followed by a discussion of results in Section 6.6. Finally, we survey related work in Section 6.7.

6.2 Concurrency Control Schemes

OLTP database systems support the part of an application that interacts with *end-users*. End-users interact with the front-end application by sending it requests to perform some function (e.g., reserve a seat on a flight). The application processes these requests and then executes transactions in the DBMS. Such users could be humans on their personal computer or mobile device, or another computer program potentially running somewhere else in the world.

A *transaction* in the context of one of these systems is the execution of a sequence of one or more operations (e.g., SQL queries) on a shared database to perform some higher-level function [62]. It is the basic unit of change in a DBMS: partial transactions are not allowed, and the effect of a group of transactions on the database's state is equivalent to any serial execution of all transactions. The transactions in modern OLTP workloads have three salient characteristics: (1) they are short-lived (i.e., no user stalls), (2) they touch a small subset of data using index look-ups (i.e., no full table scans or large joins), and (3) they are repetitive (i.e., executing the same queries with different inputs) [135].

An OLTP DBMS is expected to maintain four properties for each transaction that it executes: (1) atomicity, (2) consistency, (3) isolation, and (4) durability. These unifying concepts are collectively referred to with the *ACID* acronym [68]. Concurrency control permits end-users to access a database in a multi-programmed fashion while preserving the illusion that each of them is executing their transaction alone on a dedicated system [18]. It essentially provides the atomicity and isolation guarantees in the system.

We now describe the different concurrency control schemes that we explored in our many-core evaluation. For this discussion, we follow the canonical categorization that all concurrency schemes are either a variant of *two-phase locking* or *timestamp ordering* protocols [18]. Table 6.1 provides a summary of these different schemes.

6.2.1 Two-Phase Locking

Two-phase locking (2PL) was the first provably correct method of ensuring the correct execution of concurrent transactions in a database system [21, 47]. Under this scheme, transactions have to acquire locks for a particular element in the database before they are allowed to execute a read or write operation on that element. The transaction must acquire a *read lock* before it is allowed to read that element, and similarly it must acquire a *write lock* in order to modify that element. The DBMS maintains locks for either each tuple or at a higher logical level (e.g., tables, partitions) [53].

The ownership of locks is governed by two rules: (1) different transactions cannot simultaneously own *conflicting locks*, and (2) once a transaction surrenders ownership of a lock, it may never obtain additional locks [18]. A read lock on an element conflicts with a write lock on that same element. Likewise, a write lock on an element conflicts with a write lock on the same element.

In the first phase of 2PL, known as the *growing phase*, the transaction is allowed to acquire as many locks as it needs without releasing locks [47]. When the transaction releases a lock, it enters the second phase, known as the *shrinking phase*; it is prohibited from obtaining additional locks at this point. When the transaction terminates (either by committing or aborting), all the remaining locks are automatically released back to the coordinator.

2PL is considered a pessimistic approach in that it assumes that transactions will conflict and thus they need to acquire locks to avoid this problem. If a transaction is unable to acquire a lock for an element, then it can wait until the lock becomes available. If this waiting is uncontrolled (i.e., indefinite), then the DBMS can incur deadlocks [18]. Thus, a major difference among the different variants of 2PL is in how they handle deadlocks and the actions that they take when a deadlock is detected. We now describe the different versions of 2PL that we have implemented in our evaluation framework, contrasting them based on these two details:

2PL with Deadlock Detection (DL_DETECT): The DBMS monitors a *waits-for* graph of transactions and checks for cycles (i.e., deadlocks) [60]. When a deadlock is found, the system must choose a transaction to abort and restart in order to break the cycle. In practice, a centralized deadlock detector is used for cycle detection. The detector chooses which transaction to abort based

Two-Phase Locking (2PL)	DL_DETECT	2PL with deadlock detection.
	NO_WAIT	2PL with non-waiting deadlock prevention.
	WAIT_DIE	2PL with wait-and-die deadlock prevention.
Timestamp Ordering (T/O)	TIMESTAMP	Basic T/O algorithm.
	MVCC	Multi-version T/O.
	OCC	Optimistic concurrency control.
	H-STORE	T/O with partition-level locking.

Table 6.1: **The Concurrency Control Schemes under Evaluation.**

on the amount of resources it has already used (e.g., the number of locks it holds) to minimize the cost of restarting a transaction [18].

2PL with Non-waiting Deadlock Prevention (NO_WAIT): Unlike deadlock detection where the DBMS waits to find deadlocks after they occur, this approach is more cautious in that a transaction is aborted when the system suspects that a deadlock might occur [18]. When a lock request is denied, the scheduler immediately aborts the requesting transaction (i.e., it is not allowed to wait to acquire the lock).

2PL with Waiting Deadlock Prevention (WAIT_DIE): This is a non-preemptive variation of the NO_WAIT scheme where a transaction is allowed to wait for that transaction that holds the lock that it needs if the transaction is older than the one that holds the lock. If the requesting transaction is younger, then it is aborted (hence the term "die") and is forced to restart [18]. Each transaction needs to acquire a unique timestamp before its execution and the timestamp ordering guarantees that no deadlocks can occur.

6.2.2 Timestamp Ordering

Timestamp ordering (T/O) concurrency control schemes generate a serialization order of transactions a priori and then the DBMS enforces this order. A transaction is assigned a unique, monotonically increasing timestamp before it is executed; this timestamp is used by the DBMS to process conflicting operations in the proper order (e.g., read and write operations on the same element, or two separate write operations on the same element) [18].

We now describe the T/O schemes implemented in our test-bed. The key differences between the schemes are (1) the granularity that the DBMS checks for conflicts (i.e., tuples vs. partitions) and (2) when the DBMS checks for these conflicts (i.e., while the transaction is running or at the end).

Basic T/O (TIMESTAMP): Every time a transaction reads or modifies a tuple in the database, the DBMS compares the timestamp of the transaction with the timestamp of the last transaction that reads or writes the same tuple. For any read or write operation, the DBMS rejects the request if the transaction's timestamp is less than the timestamp of the last write to that tuple. Likewise, for a write operation, the DBMS rejects it if the transaction's timestamp is less than the timestamp of the last read to that tuple. In TIMESTAMP, a read query makes a local copy of the tuple to ensure repeatable reads since it is not protected by locks. When a transaction is aborted, it is assigned a new timestamp and then restarted. This corresponds to the "basic T/O" algorithm as described in [18], but our implementation uses a decentralized scheduler.

Multi-version Concurrency Control (MVCC): Under MVCC, every write operation creates a new version of a tuple in the database [19, 20]. Each version is tagged with the timestamp of the transaction that created it. The DBMS maintains an internal list of the versions of an element. For a read operation, the DBMS determines which version in this list the transaction will access. Thus, it ensures a serializable ordering of all operations. One benefit of MVCC is that the DBMS does not reject operations that arrive late. That is, the DBMS does not reject a read operation because the element that it targets has already been overwritten by another transaction [20].

Optimistic Concurrency Control (OCC): The DBMS tracks the read/write sets of each transaction and stores all of their write operations in their private workspace [86]. When a transaction commits, the system determines whether that transaction's read set overlaps with the write set of any concurrent transactions. If no overlap exists, then the DBMS applies the changes from the transaction's workspace into the database; otherwise, the transaction is aborted and restarted. The advantage of this approach for main memory DBMSs is that transactions write their updates to shared memory only at commit time, and thus the contention period is short [144]. Modern implementations of OCC include Silo [144] and Microsoft's Hekaton [41, 95].

T/O with Partition-level Locking (H-STORE): The database is divided into disjoint subsets of memory called *partitions*. Each partition is protected by a lock and is assigned a single-threaded execution engine that has exclusive access to that partition. Each transaction must acquire the locks for all of the partitions that it needs to access before it is allowed to start running. This requires the DBMS to know what partitions that each individual transaction will access before it begins [120]. When a transaction request arrives, the DBMS assigns it a timestamp and then adds it to all of the lock acquisition queues for its target partitions. The execution engine for a partition removes a transaction from the queue and grants it access to that partition if the transaction has the oldest

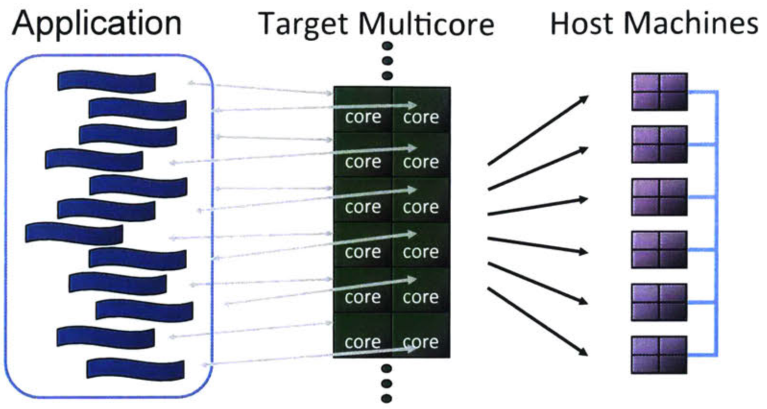


Figure 6-1: **Graphite Simulator Infrastructure** – Application threads are mapped to simulated core threads deployed on multiple host machines.

timestamp in the queue [135]. Smallbase was an early proponent of this approach [72], and more recent examples include H-Store [82].

6.3 Many-Core DBMS Test-Bed

Since many-core chips with up to 1000 cores do not yet exist, we performed our analysis by using Graphite [110], a CPU simulator that can scale up to 1024 cores. For the DBMS, we implemented a main memory OLTP engine that only contains the functionality needed for our experiments. The motivation for using a custom DBMS is two fold. First, we can ensure that no other bottlenecks exist other than concurrency control. This allows us to study the fundamentals of each scheme in isolation without interference from unrelated features. Second, using a full-featured DBMS is impractical due to the considerable slowdown of the simulator (e.g., Graphite has an average slowdown of 10,000$\times$). Our engine allows us to limit the experiments to reasonable time. We now describe the simulation infrastructure, the DBMS engine, and the workloads used in this study.

6.3.1 Simulator and Target Architecture

Graphite [110] is a fast CPU simulator for large-scale multicore systems. Graphite runs off-the-shelf Linux applications by creating a separate thread for each core in the architecture. As shown in Figure 6-1, each application thread is attached to a simulated core thread that can then be mapped to different processes on separate host machines. For additional performance, Graphite relaxes cycle accuracy, using periodic synchronization mechanisms to loosely adjust the timing of different hardware components. As with other similar CPU simulators, it only executes the application and

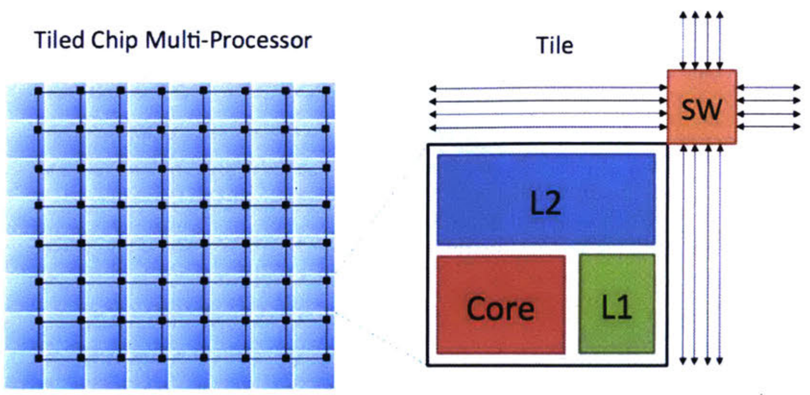


Figure 6-2: **Target Architecture** – Tiled chip multi-processor with 64 tiles and a 2D-mesh network-on-chip. Each tile contains a processing core, L1 and L2 caches, and a network switch (SW).

does not model the operating system. For this paper, we deployed Graphite on a 22-node cluster, each with dual-socket Intel Xeon E5-2670 processors and 64GB of DRAM.

The target architecture is a tiled multicore CPU, where each tile contains a low-power, in-order processing core, 32KB L1 instruction/data cache, a 512KB L2 cache slice, and a network router. This is similar to other commercial CPUs, such as Tilera's Tile64 (64 cores), Intel's SCC (48 cores), and Intel's Knights Landing (72 cores) [6]. Tiles are interconnected using a high-bandwidth, 2D-mesh on-chip network, where each hop takes two cycles. Both the tiles and network are clocked at 1GHz frequency. A schematic of the architecture for a 64-core machine is depicted in Figure 6-2.

We use a shared L2-cache configuration because it is the most common last-level cache design for commercial multicores. In a comparison experiment between shared and private L2-caches, we observe that a shared L2 cache leads to significantly less memory traffic and higher performance for OLTP workloads due to its increased aggregate cache capacity (results not shown). Since L2 slices are distributed among the different tiles, the simulated multicore system is a NUCA (Non-Uniform Cache Access) architecture, where L2-cache latency increases with distance in the 2D-mesh.

6.3.2 DBMS

We implemented our own lightweight main memory DBMS to run in Graphite. It executes as a single process with the number of worker threads equal to the number of cores, where each thread is mapped to a different core. All data in our DBMS is stored in memory in a row-oriented manner. The system supports basic hash table indexes and a pluggable lock manager that allows us to swap in the different implementations of the concurrency control schemes described in Section 6.2. It also allows the indexes and lock manager to be partitioned (as in the case with the H-STORE scheme) or run in a centralized mode.

Client threads are not simulated in our system; instead, each worker contains a fixed-length queue of transactions that are served in order. This reduces the overhead of network protocols, which are inherently difficult to model in the simulator. Each transaction contains program logic intermixed with query invocations. The queries are executed serially by the transaction's worker thread as they are encountered in the program logic. Transaction statistics, such as throughput, latency, and abort rates, are collected after a warm-up period that is long enough for the system to achieve a steady state.

In addition to runtime statistics, our DBMS also reports how much time each transaction spends in the different components of the system [70]. We group these measurements into six categories:

USEFUL WORK: The time that the transaction is actually executing application logic and operating on tuples in the system.

ABORT: The time spent on executing transactions that eventually abort.

TS ALLOCATION: The time that it takes for transactions to acquire unique timestamps from the centralized allocator.

INDEX: The time that the transaction spends in accessing the hash index (e.g., lookups, inserts, and deletes), including the overhead of low-level latching of the buckets in the hash tables.

WAIT: The total amount of time that a transaction has to wait. A transaction may either wait for a lock (e.g., 2PL) or for a tuple whose value is not ready yet (e.g., T/O).

MANAGER: The time that the transaction spends in the lock manager or the timestamp manager. This excludes any time that it has to wait.

6.3.3 Workloads

We next describe the two benchmarks that we implemented in our test-bed for this analysis.

YCSB: The Yahoo! Cloud Serving Benchmark is a collection of workloads that are representative of large-scale services created by Internet-based companies [35]. For all of the YCSB experiments in this thesis, we used a $\sim$20GB YCSB database containing a single table with 20 million records. Each YCSB tuple has a single primary key column and then 10 additional columns each with 100 bytes of randomly generated string data. The DBMS creates a single hash index for the primary key.

Each transaction in the YCSB workload by default accesses 16 records in the database. Each access can be either a read or an update. The transactions do not perform any computation in their program logic. All of the queries are independent from each other; that is, the input of one query does not depend on the output of a previous query. The records accessed in YCSB follows a Zipfian distribution that is controlled by a parameter called *theta* that affects the level of contention in the benchmark [63]. When *theta*=0, all tuples are accessed with the same probability. But when *theta*=0.6 or *theta*=0.8, a hotspot of 10% of the tuples in the database are accessed by ~40% and ~60% of all transactions, respectively.

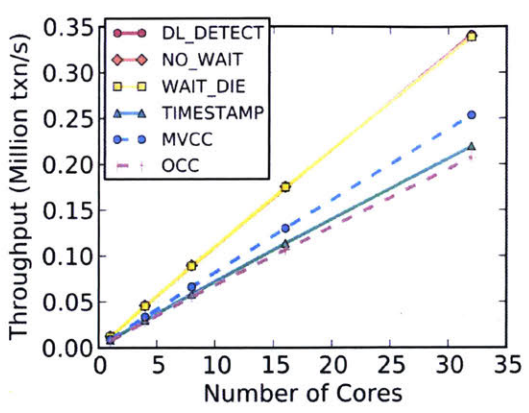
TPC-C: This benchmark is the current industry standard for evaluating the performance of OLTP systems [140]. It consists of nine tables that simulate a warehouse-centric order processing application. All of the transactions in TPC-C provide a warehouse id as an input parameter for the transaction, which is the ancestral foreign key for all tables except `ITEM`. For a concurrency control algorithm that requires data partitioning (i.e., H-STORE), TPC-C is partitioned based on this warehouse id.

Only two (`Payment` and `NewOrder`) out of the five transactions in TPC-C are modeled in our simulation in this chapter. Since these two comprise 88% of the total TPC-C workload, this is a good approximation. Our version of TPC-C is a "good faith" implementation, although we omit the "thinking time" for worker threads. Each worker issues transactions without pausing; this mitigates the need to increase the size of the database with the number of concurrent transactions.

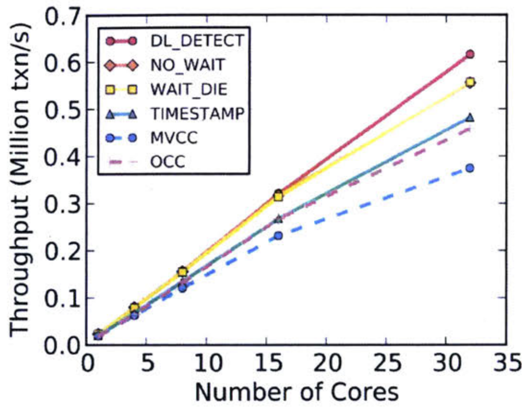
6.3.4 Simulator vs. Real Hardware

To show that using the Graphite simulator generates results that are comparable to existing hardware, we deployed our DBMS on an Intel Xeon E7-4830 and executed a read-intensive YCSB workload with medium contention (*theta*=0.6). We then executed the same workload in Graphite with the same number of cores.

The results in Figure 6-3 show that all of the concurrency control schemes exhibit the same performance trends on Graphite and the real CPU. We note, however, that the relative performance difference between MVCC, TIMESTAMP, and OCC is different in Figure 6-3b. This is because MVCC accesses memory more than the other two schemes and those accesses are more expensive on a two-socket system. Graphite models a single CPU socket and thus there is no inter-socket traffic. In addition to this, the throughput of the T/O-based and WAIT_DIE schemes drops on 32 cores due to the overhead of cross-core communication during timestamp allocation. We discuss this issue in more details in Section 6.4.3.



(a) Graphite Simulation



(b) Real Hardware

Figure 6-3: **Simulator vs. Real Hardware** – Comparison of the concurrency control schemes running in Graphite and a real multicore CPU using the YCSB workload with medium contention (*theta*=0.6).

6.4 Design Choices & Optimizations

One of the main challenges of this study was designing a DBMS and concurrency control schemes that are as scalable as possible. When deploying a DBMS on 1000 cores, many secondary aspects of the implementation become a hindrance to performance. We did our best to optimize each algorithm, removing all possible scalability bottlenecks while preserving their essential functionality. Most of this work was to eliminate shared data structures and devise distributed versions of the classical algorithms [18].

In this section, we discuss our experience with developing a many-core OLTP DBMS and highlight the design choices we made to achieve a scalable system. Additionally, we identify fundamental bottlenecks of both the 2PL and T/O schemes and show how hardware support mitigates these problems. We present our detailed analysis of the individual schemes in Section 6.5.

6.4.1 General Optimizations

We first discuss the optimizations that we added to improve the DBMS's performance across all concurrency control schemes.

Memory Allocation: One of the first limitations we encountered when trying to scale our DBMS to large core counts was the `malloc` function. When using the default Linux version of `malloc`, we found that the DBMS spends most of the time waiting for memory allocation. This is a problem even for read-only workloads, since the DBMS still needs to allocate local records for

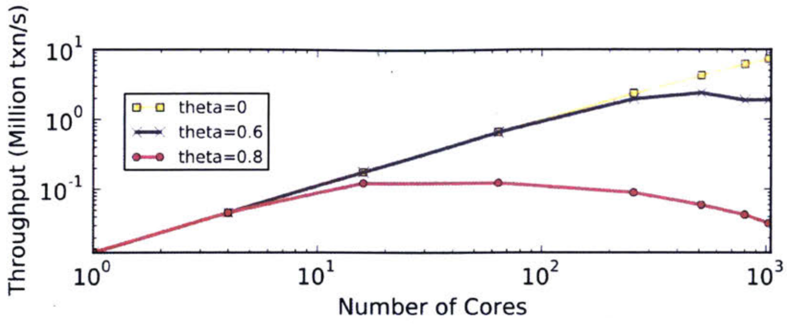


Figure 6-4: **Lock Thrashing** – Results for a write-intensive YCSB workload using the DL_DETECT scheme without deadlock detection. Each transaction acquires locks in their primary key order.

reads in TIMESTAMP and to create internal metadata handles for access tracking data structures. We tried running optimized versions (`TCMalloc` [57], `jemalloc` [48]), but both yielded similar disappointing results at 1000 cores.

We overcame this by writing a custom `malloc` implementation. Similar to `TCMalloc` and `jemalloc`, each thread is assigned its own memory pool. But the difference is that our allocator automatically resizes the pools based on the workload. For example, if a benchmark frequently allocates large chunks of contiguous memory, the pool size increases to amortize the cost for each allocation.

Lock Table: As identified in previous work [81, 124], the lock table is another key contention point in DBMSs. Instead of having a centralized lock table or timestamp manager, we implemented these data structures in a per-tuple fashion where each transaction only latches the tuples that it needs. This improves scalability, but increases the memory overhead because the DBMS maintains additional metadata for the lock sharer/waiter information. In practice, this metadata (several bytes) is negligible for large tuples.

Mutexes: Accessing a mutex lock is an expensive operation that requires multiple messages to be sent across the chip. A central critical section protected by a mutex will limit the scalability of any system (cf. Section 6.4.3). Therefore, it is important to avoid using mutexes on the critical path. For 2PL, the mutex that protects the centralized deadlock detector is the main bottleneck, while for T/O algorithms it is the mutex used for allocating unique timestamps. In the subsequent sections, we describe the optimizations that we developed to obviate the need for these mutexes.

6.4.2 Scalable Two-Phase Locking

We next discuss the optimizations for the 2PL algorithms.

Deadlock Detection: For DL_DETECT, we found that the deadlock detection algorithm is a bottleneck when multiple threads compete to update their *waits-for* graph in a centralized data structure. We solved this by partitioning the data structure across cores to make the deadlock detector completely lock-free. When a transaction updates its *waits-for* graph, its thread inserts the transactions that the current transaction is waiting for into its local queue without holding any locks. This step is local (i.e., no cross-core communication), as the thread does not write to the queues of other transactions.

In the deadlock detection process, a thread searches for cycles in a partial waits-for graph by only reading the queues of related threads but not locking the queues. Although this approach may not discover a deadlock immediately after it forms, the thread is guaranteed to find it on subsequent passes [20].

Lock Thrashing: Even with improved deadlock detection, DL_DETECT still does not scale due to lock thrashing. This occurs when a transaction holds its locks until it commits, blocking all the other concurrent transactions that attempt to acquire those locks. This becomes a problem with high contention and a large number of concurrent transactions. It is the main scalability bottleneck of all 2PL schemes.

To demonstrate the impact of thrashing, we executed a write-intensive YCSB workload (i.e., 50/50% read-write mixture) using a variant of DL_DETECT where transactions acquire locks in primary key order. Although this approach is not practical for all workloads, it removes the need for deadlock detection and allows us to better observe the effects of thrashing. Figure 6-4 shows the transaction throughput as a function of the number of cores for different levels of contention. When there is no skew in the workload (*theta*=0), the contention for locks is low and the throughput scales almost linearly. As the contention level increases, however, thrashing starts to occur. With medium contention (*theta*=0.6), the throughput peaks at several hundred cores and then decreases due to thrashing. At the highest contention level (*theta*=0.8), the DBMS's throughput peaks at 16 cores and cannot scale beyond that. Simulation results show that almost all the execution time is spent on waiting for locks. This indicates that lock thrashing is the key bottleneck of lock-based approaches that limits scalability in high-contention scenarios.

Waiting vs. Aborting: The thrashing problem can be mitigated in DL_DETECT by aborting

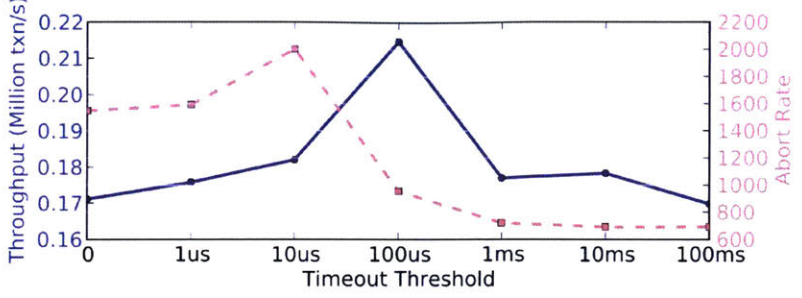


Figure 6-5: **Waiting vs. Aborting** – Results for DL_DETECT with varying timeout threshold running high contention YCSB (*theta*=0.8) at 64 cores.

some transactions to reduce the number of active transactions at any point in time. Ideally, this keeps the system running at the highest throughput for each curve in Figure 6-4. We added a *timeout threshold* in the DBMS that causes the system to abort and restart any transaction that has been waiting for a lock for an amount of time greater than the threshold. Note that when the timeout threshold is zero, this algorithm is equivalent to NO_WAIT.

We ran the same YCSB workload with high contention using different timeout thresholds on a 64-core CPU. We measure both the throughput and abort rate in the DBMS for the DL_DETECT scheme sweeping the timeout from 0–100 ms.

The results in Figure 6-5 indicate that when the CPU has a small number of cores, it is better to use a shorter timeout threshold. This highlights the trade-off between performance and the transaction abort rate. With a small timeout, the abort rate is high, which reduces the number of running transactions and alleviates the thrashing problem. Using a longer timeout reduces the abort rate at the cost of more thrashing. Therefore, in this paper, we evaluate DL_DETECT with its timeout threshold set to 100μs. In practice, the threshold should be based on an application's workload characteristics.

6.4.3 Scalable Timestamp Ordering

Finally, we discuss the optimizations that we developed to improve the scalability of the T/O-based algorithms.

Timestamp Allocation: All T/O-based algorithms make ordering decisions based on transactions' assigned timestamps. The DBMS must therefore guarantee that each timestamp is allocated to only one transaction. A naïve approach to ensure this is to use a *mutex* in the allocator's critical section, but this leads to poor performance. Another common solution is to use an *atomic addi-*

tion operation to advance a global logical timestamp. This requires fewer instructions and thus the DBMS's critical section is locked for a smaller period of time than with a mutex. But as we will show, this approach is still insufficient for a 1000-core CPU. We now discuss three timestamp allocation alternatives: (1) atomic addition with batching [144], (2) CPU clocks, and (3) hardware counters.

With the *batched atomic addition* approach, the DBMS uses the same atomic instruction to allocate timestamps, but the timestamp manager returns multiple timestamps together in a batch for each request.

To generate a timestamp using *clock*-based allocation, each worker thread reads a logical clock from its local core and then concatenates it with its thread id. This provides good scalability as long as all the clocks are synchronized. In distributed systems, synchronization is accomplished using software protocols [111] or external clocks [36]. On a many-core CPU, however, this imposes large overhead and thus requires hardware support. To the best of our knowledge, only Intel CPUs support synchronized clocks across cores at the present moment. It is not clear if synchronized clocks will be continuously supported in the future.

Lastly, the third approach is to use an efficient, built-in *hardware counter*. The counter is physically located at the center of the CPU such that the average distance to each core is minimized. No existing CPU currently supports this. Thus, we implemented a counter in Graphite where a timestamp request is sent through the on-chip network to increment it atomically in a single cycle.

To determine the maximum rate that the DBMS can allocate timestamps for each method, we ran a micro-benchmark where threads continuously acquire new timestamps. The throughput as a function of the number of cores is shown in Figure 6-6. We first note that *mutex*-based allocation has the lowest performance, with $\sim$1 million timestamps per second (ts/s) on 1024 cores. The *atomic addition* method reaches a maximum of 30 million ts/s at 16 cores, but throughput decreases as the number of cores increases. At 1024 cores, the allocation throughput is only 8 million ts/s. This is due to the cache coherence traffic resulting from writing back and invalidating the last copy of the corresponding cache line for every timestamp. This takes one round trip of communication across the chip or $\sim$100 cycles for a 1024-core CPU, which translates to a maximum throughput of 10 million ts/s at 1GHz frequency. Batching these allocations does help, but it causes performance issues when there is contention (see below). The hardware-based solutions are able to scale with the number of cores. Because incrementing the timestamp takes only one cycle with the *hardware counter*-based approach, this method achieves a maximum throughput of 1 billion ts/s. The

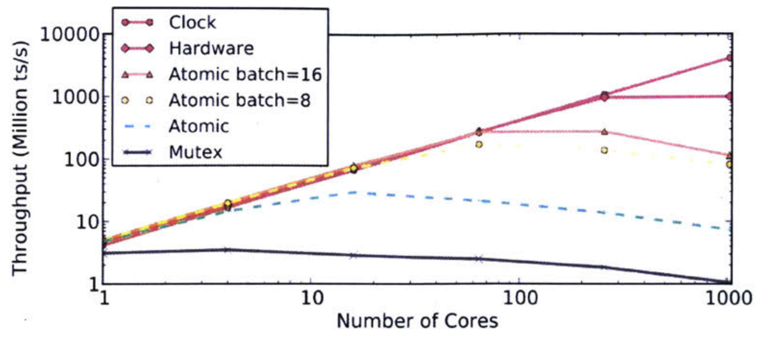


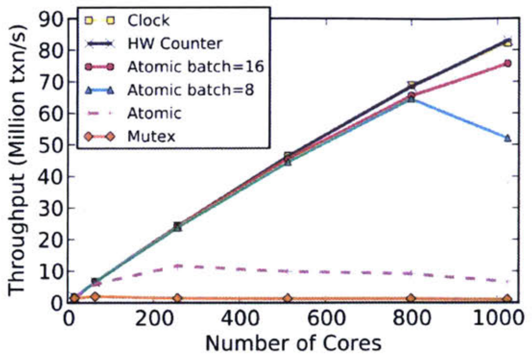
Figure 6-6: **Timestamp Allocation Micro-benchmark** – Throughput measurements for different timestamp allocation methods.

performance gain comes from removing the coherence traffic by executing the addition operation remotely. The *clock*-based approach has ideal (i.e., linear) scaling, since this solution is completely decentralized, so long as synchronized clocks are available.

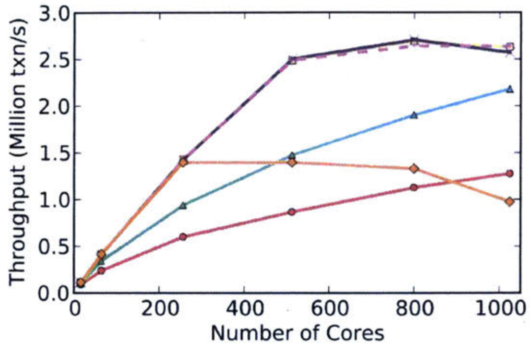
We also tested the different allocation schemes in the DBMS to see how they perform for real workloads. For this experiment, we executed a write-intensive YCSB workload with two different contention levels using the TIMESTAMP scheme. The results in Figure 6-7a show that with no contention, the relative performance of the allocation methods are the same as in Figure 6-6. When there is contention, however, the trends in Figure 6-7b are much different. First, the DBMS's throughput with the *batched atomic addition* method is much worse. This is because in our implementation, when a transaction is restarted due to a conflict, it gets restarted in the same worker thread and is assigned the next timestamp from the last batch. But this new timestamp may still be less than the one for the other transaction that caused the abort. Therefore, the current transaction may continuously restart until the thread fetches a new batch. The non-batched *atomic addition* method performs as well as the *clock* and *hardware counter* approaches. Hence, the DBMS uses *atomic addition* without batching to allocate timestamps because the other approaches require specialized hardware support that is currently not available on all CPUs.

Distributed Validation: The original OCC algorithm contains a critical section at the end of the read phase, where the transaction's read set is compared to previous transactions' write sets to detect conflicts. Although this step is short, as mentioned above, any mutex-protected critical section severely hurts scalability. We solve this problem by using per-tuple validation that breaks up this check into smaller operations. This is similar to the approach used in Hekaton [95] but it is simpler, since we only support a single version of each tuple.

Local Partitions: We optimized the original H-STORE protocol to take advantage of shared



(a) No Contention



(b) Medium Contention

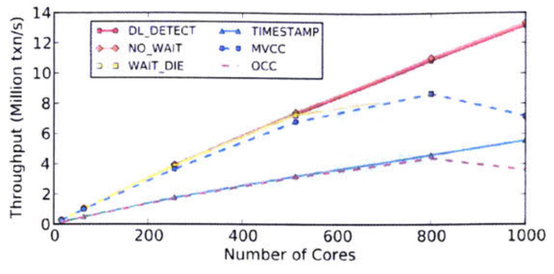
Figure 6-7: **Timestamp Allocation** – Throughput of the YCSB workload using TIMESTAMP with different timestamp allocation methods.

memory. Because the DBMS's worker threads run in a single process, we allow multi-partition transactions to access tuples at remote partitions directly instead of sending query requests that are executed by the remote partitions' worker threads. This allows for a simpler implementation that is faster than using inter-process communication. With this approach, the data is not physically partitioned since on-chip communication latency is low. Read-only tables are accessed by all threads without replication, thus reducing the memory footprint. Finally, we use the same timestamp allocation optimizations from above to avoid the mandatory wait time to account for clock skew [135].

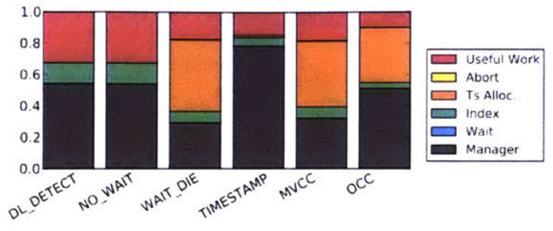
6.5 Experimental Analysis

We now present the results from our analysis of the different concurrency control schemes. Our experiments are grouped into two categories: (1) *scalability* and (2) *sensitivity* evaluations. For the former, we want to determine how well the schemes perform as we increase the number of cores. We scale the number of cores up to 1024 while fixing the workload parameters. With the sensitivity experiments, we vary a single workload parameter (e.g., transaction access skew). We report the DBMS's total simulated throughput as well as a breakdown of the amount of time that each worker thread spends in the different parts of the system listed in Section 6.3.2.

We begin with an extensive analysis of the YCSB workload. The nature of this workload allows us to change its parameters and create a variety of scenarios that stress the concurrency control schemes in different ways. Next, we analyze the TPC-C workload, where we vary the number of warehouses and observe the impact on the throughput of the algorithms. The H-STORE scheme is excluded from our initial experiments and is only introduced in Section 6.5.5 when we analyze



(a) Total Throughput



(b) Runtime Breakdown (1024 cores)

Figure 6-8: **Read-only Workload** – Throughput and runtime breakdown of different concurrency control algorithms for a read-only YCSB workload.

database partitioning.

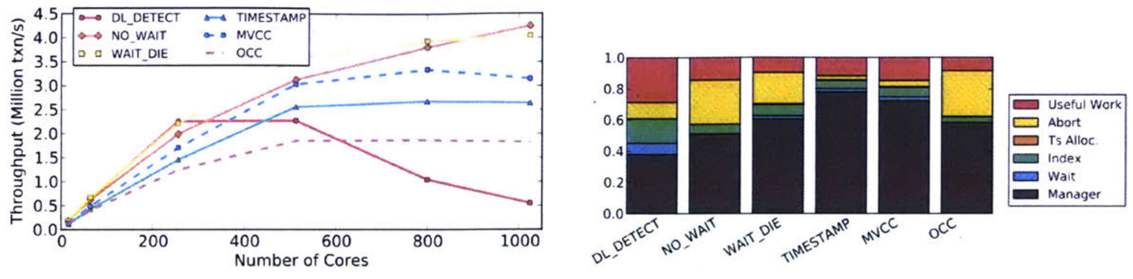
6.5.1 Read-Only Workload

In this first scalability analysis experiment, we executed a YCSB workload comprising read-only transactions with a uniform access distribution. Each transaction executes 16 separate tuple reads at a time. This provides a baseline for each concurrency control scheme before we explore more complex workload arrangements.

In a perfectly scalable DBMS, the throughput should increase linearly with the number of cores. However, this is not the case for the T/O algorithms as shown in Figure 6-8. Performance stops increasing after a few hundred cores. The time breakdown in Figure 6-8b indicates that timestamp allocation becomes the bottleneck with a large core count. OCC hits the bottleneck even earlier since it needs to allocate timestamps twice per transaction (i.e., at transaction start and before the validation phase). Both OCC and TIMESTAMP have significantly worse performance than the other algorithms regardless of the number of cores. These algorithms waste cycles because they copy tuples to perform a read, whereas the other algorithms read tuples in place.

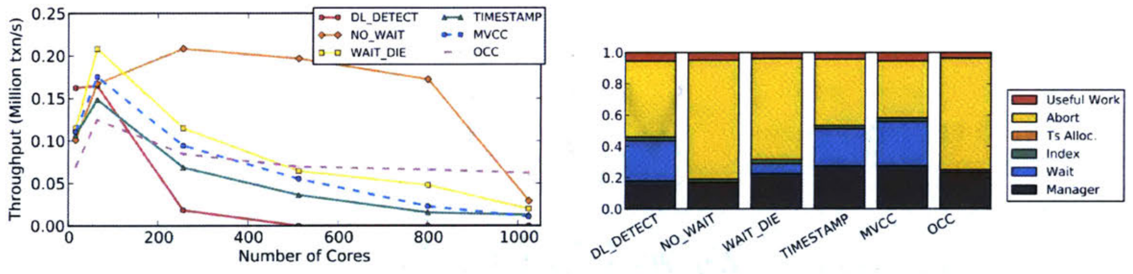
6.5.2 Write-Intensive Workload

A read-only workload represents an optimistic (and unrealistic) scenario, as it generates no data contention. But even if we introduce writes in the workload, the large size of the dataset means that the probability that any two transactions access the same tuples at the same time is small. In reality, the access distribution of an OLTP application is rarely uniform. Instead, it tends to follow a power law distribution, where certain tuples are more likely to be accessed than others. This can be from either skew in the popularity of elements in the database or skew based on temporal locality



(a) Total Throughput (b) Runtime Breakdown (512 cores)

Figure 6-9: **Write-Intensive Workload (Medium Contention)** – Results for YCSB workload with medium contention (*theta*=0.6).



(a) Total Throughput (b) Runtime Breakdown (64 cores)

Figure 6-10: **Write-Intensive Workload (High Contention)** – Results for YCSB workload with high contention (*theta*=0.8).

(i.e., newer tuples are accessed more frequently). As a result, this increases contention because transactions compete to access the same data.

We executed a write-intensive YCSB workload comprising transactions that each accesses 16 tuples. Within each transaction, an access is a write with 50% probability. The amount of skew in the workload is determined by the parameter *theta* (cf. Section 6.3.3). Higher *theta* means more skew. We run transactions with medium and high contention levels.

The medium contention results in Figure 6-9 show that NO_WAIT and WAIT_DIE are the only 2PL schemes that scale past 512 cores. NO_WAIT scales better than WAIT_DIE. For DL_DETECT, the breakdown in Figure 6-9b indicates that the DBMS spends a larger percentage of its time waiting. DL_DETECT is inhibited by lock thrashing at 256 cores. NO_WAIT is the most scalable because it eliminates this waiting. We note, however, that both NO_WAIT and WAIT_DIE have a high transaction abort rate. This is not an issue in our experiments because restarting an aborted transaction has low overhead; the time it takes to undo a transaction is slightly less than the time it takes to re-execute the transaction's queries. But in reality, the overhead may be larger for workloads where transactions have to rollback changes to multiple tables, indexes, and materialized views.

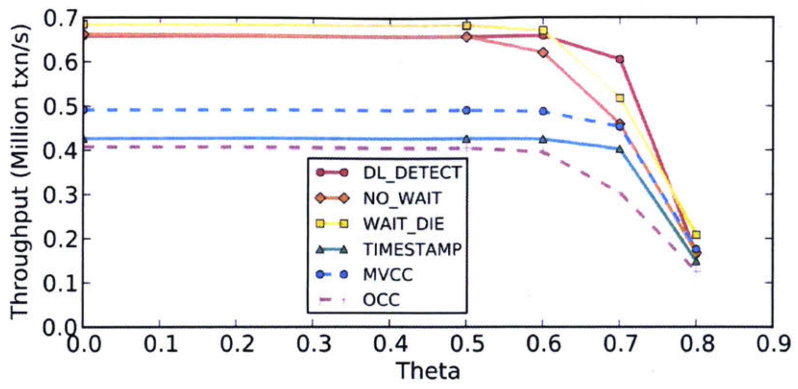
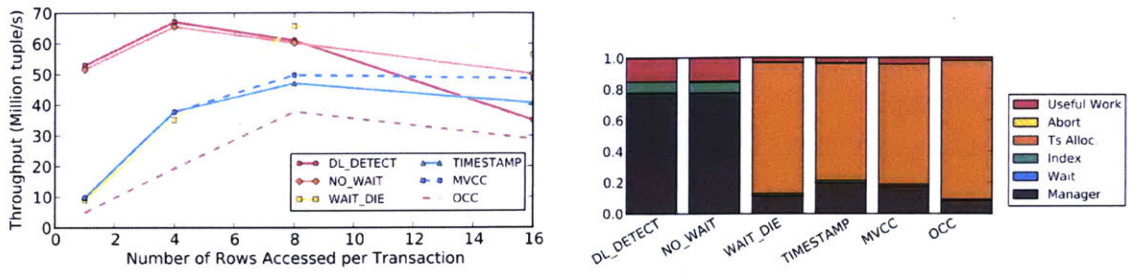


Figure 6-11: **Write-Intensive Workload (Variable Contention)** – Results for YCSB workload with varying level of contention on 64 cores.

The results in Figure 6-9a also show that the T/O algorithms perform well in general. Both TIMESTAMP and MVCC are able to overlap operations and reduce the waiting time. MVCC performs slightly better since it keeps multiple versions of a tuple and thus can serve read requests even if they have older timestamps. OCC does not perform as well because it spends a large portion of its time aborting transactions; the overhead is worse since each transaction has to finish before the conflict is resolved.

With higher contention, the results in Figure 6-10 show that performance of all of the algorithms is much worse. Figure 6-10a shows that almost all of the schemes are unable to scale to more than 64 cores. Beyond this point, the DBMS's throughput stops increasing and there is no performance benefit to the increased core count. NO_WAIT initially outperforms all the others, but then succumbs to lock thrashing (cf. Figure 6-4). Surprisingly, OCC performs the best on 1024 cores. This is because although a large number of transactions conflict and have to abort during the validation phase, one transaction is always allowed to commit. The time breakdown in Figure 6-10b shows that the DBMS spends a larger amount of time aborting transactions in every scheme.

To better understand when each scheme begins to falter with increased contention, we fixed the number of cores to 64 and performed a sensitivity analysis on the skew parameter (*theta*). The results in Figure 6-11 indicate that for *theta* values less than 0.6, the contention has little effect on the performance. But for higher contention, there is a sudden drop in throughput that renders all algorithms non-scalable and the throughput approaches zero for values greater than 0.8.



(a) Total Throughput

(b) Runtime Breakdown (transaction length = 1)

Figure 6-12: **Working Set Size** – The number of tuples accessed per core on 512 cores for transactions with a varying number of queries (*theta*=0.6).

6.5.3 Working Set Size

The number of tuples accessed by a transaction is another factor that impacts scalability. When a transaction's working set is large, it increases the likelihood that the same data is accessed by concurrent transactions. For 2PL algorithms, this increases the length of time that locks are held by a transaction. With T/O, however, longer transactions may reduce timestamp allocation contention. In this experiment, we vary the number of tuples accessed per transaction in a write-intensive YCSB workload. Because short transactions lead to higher throughput, we measure the number of tuples accessed per second, rather than transactions completed. We use the medium skew setting (*theta*=0.6) and fix the core count to 512.

The results in Figure 6-12 show that when transactions are short, the lock contention is low. DL_DETECT and NO_WAIT have the best performance in this scenario, since there are few deadlocks and the number of aborts is low. But as the transactions' working set size increases, the performance of DL_DETECT degrades due to the overhead of thrashing. For the T/O algorithms and WAIT_DIE, the throughput is low when the transactions are short because the DBMS spends a majority of its time allocating timestamps. But as the transactions become longer, the timestamp allocation cost is amortized. OCC performs the worst because it allocates double the number of timestamps as the other schemes for each transaction.

Figure 6-12b shows the time breakdown for transaction length equals to one. Again, we see that the T/O schemes spend most of their execution time allocating timestamps. As the transactions become longer, Figures 6-8b and 6-9b show that the allocation is no longer the main bottleneck. The results in Figure 6-12 also show that the T/O-based algorithms are more tolerant to contention than DL_DETECT.

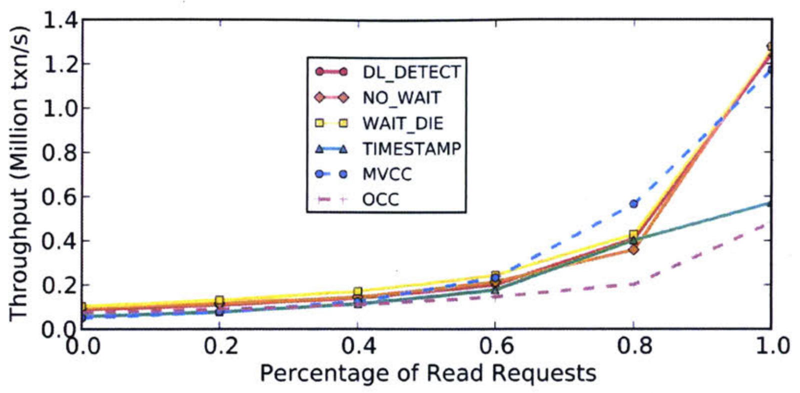


Figure 6-13: **Read/Write Mixture** – Results for YCSB with a varying percentage of read-only transactions with high contention (*theta*=0.8).

6.5.4 Read/Write Mixture

Another important factor for concurrency control is the read/write mixtures of transactions. More writes lead to more contention that affect the algorithms in different ways. For this experiment, we use YCSB on a 64 core configuration and vary the percentage of read queries executed by each transaction. Each transaction executes 16 queries using the high skew setting (*theta*=0.8).

The results in Figure 6-13 indicate that all of the algorithms achieve better throughput when there are more read transactions. At 100% reads, the results match the previous read-only results in Figure 6-8a. TIMESTAMP and OCC do not perform well because they copy tuples for reading. MVCC stand out as having the best performance when there are a small number of write transactions. This is also an example where supporting non-blocking reads through multiple versions is most effective; read queries access the correct version of a tuple based on timestamps and do not need to wait for a writing transaction. This is a key difference from TIMESTAMP, where late arriving queries are rejected and their transactions are aborted.

6.5.5 Database Partitioning

Up to this point in our analysis, we assumed that the database is stored as a single partition in memory and that all worker threads can access any tuple. With the H-STORE scheme, however, the DBMS splits the database into disjoint subsets to increase scalability [135]. This approach achieves good performance only if the database is partitioned in such a way that enables a majority of transactions to only access data at a single partition [120]. H-STORE does not work well when the workload contains multi-partition transactions because of its coarse-grained locking scheme. It also

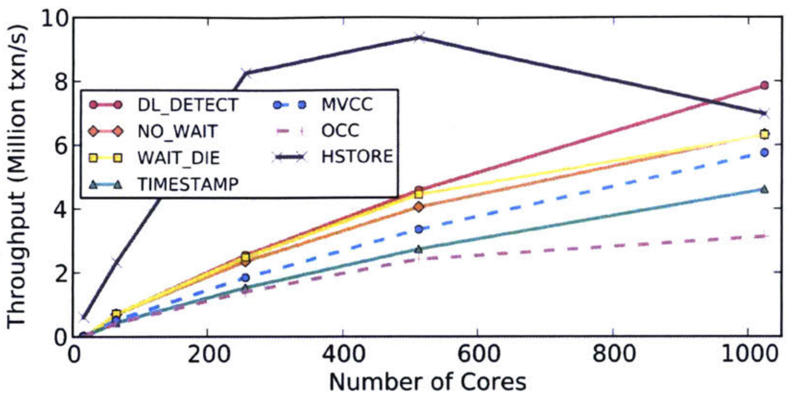


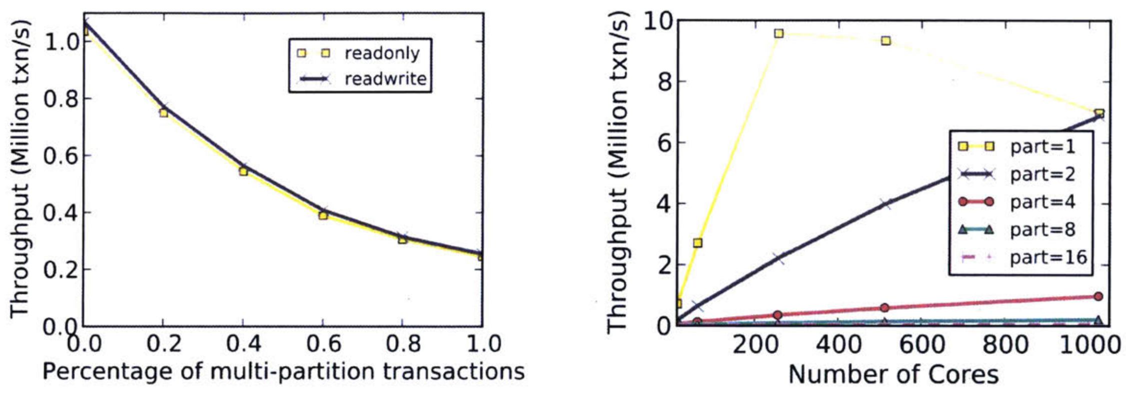
Figure 6-14: **Database Partitioning** – Results for a read-only workload on a partitioned YCSB database. The transactions access the database based on a uniform distribution (*theta*=0).

matters how many partitions each transaction accesses; for example, H-STORE will still perform poorly even with a small number of multi-partition transactions if they access all partitions. To explore these issues in a many-core setting, we first compare H-STORE to the six other schemes under ideal conditions. We then analyze its performance with multi-partition transactions.

We divide the YCSB database into the same number of partitions as the number of cores in each trial. Since YCSB only has one table, we use a simple hashing strategy to assign tuples to partitions based on their primary keys so that each partition stores approximately the same number of records. These tests use a write-intensive workload where each transaction executes 16 queries that all use index look-ups without any skew (*theta*=0.0). We also assume that the DBMS knows what partition to assign each transaction to at runtime before it starts [120].

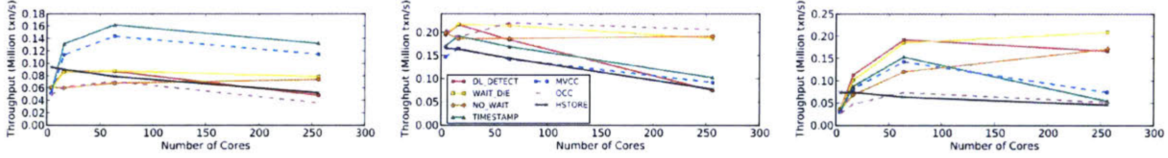
In the first experiment, we executed a workload comprised only of single-partition transactions. The results in Figure 6-14 show that H-STORE outperforms all other schemes up to 800 cores. Since it is specially designed to take advantage of partitioning, it has a much lower overhead for locking than the other schemes. But because H-STORE also depends on timestamp allocation for scheduling, it suffers from the same bottleneck as the other T/O-based schemes. As a result, the performance degrades at higher core counts. For the other schemes, partitioning does not have a significant impact on throughput. It would be possible, however, to adapt their implementation to take advantage of partitioning [124].

We next modified the YCSB driver to vary the percentage of multi-partition transactions in the workload and deployed the DBMS on a 64-core CPU. The results in Figure 6-15a illustrate two important aspects of the H-STORE scheme. First, there is no difference in performance whether or not



(a) Multi-Partition Percentage (b) Partitions per Transaction

Figure 6-15: **Multi-Partition Transactions** – Sensitivity analysis of the H-STORE scheme for YCSB workloads with multi-partition transactions.



(a) `Payment` + `NewOrder` (b) `Payment` only (c) `NewOrder` only

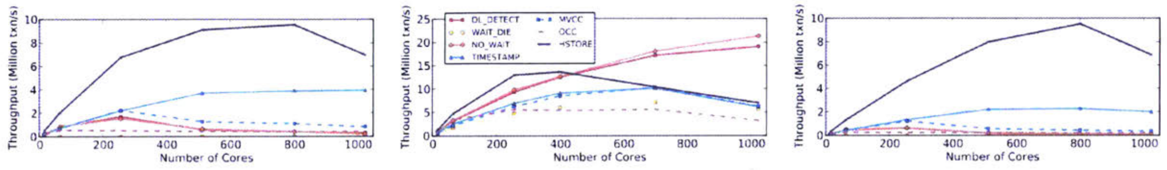
Figure 6-16: **TPC-C (4 warehouses)** – Results for the TPC-C workload running up to 256 cores.

the workload contains transactions that modify the database; this is because of H-STORE's locking scheme. Second, the DBMS's throughput degrades as the number of multi-partition transactions in the workload increases because they reduce the amount of parallelism in the system [120, 144].

Lastly, we executed YCSB with 10% multi-partition transactions and varied the number of partitions that they access. The DBMS's throughput for the single-partition workload in Figure 6-15b exhibits the same degradation due to timestamp allocation as H-STORE in Figure 6-14. This is also why the throughputs for the one- and two-partition workloads converge at 1000 cores. The DBMS does not scale with transactions accessing four or more partitions because of the reduced parallelism and lock contention on the partition.

6.5.6 TPC-C

Finally, we analyze the performance of all the concurrency control algorithms when running the TPC-C benchmark. The transactions in TPC-C are more complex than those in YCSB, and TPC-C is representative of a large class of OLTP applications. For example, they access multiple tables with a *read-modify-write* access pattern and the output of some queries are used as the input for



(a) `Payment` + `NewOrder` (b) `Payment` only (c) `NewOrder` only

Figure 6-17: **TPC-C (1024 warehouses)** – Results for the TPC-C workload running up to 1024 cores.

subsequent queries in the same transaction. TPC-C transactions can also abort because of certain conditions in their program logic, as opposed to only because the DBMS detected a conflict.

The workload in each trial comprises 50% `NewOrder` and 50% `Payment` transactions. These two make up 88% of the default TPC-C mix and are the most interesting in terms of complexity. Supporting the other transactions would require additional DBMS features, such as B-tree latching for concurrent updates. This would add additional overhead to the system.

The size of TPC-C databases are typically measured by the number of warehouses. The warehouse is the root entity for almost all tables in the database. We follow the TPC-C specification where ~10% of the `NewOrder` transactions and ~15% of the `Payment` transactions access a "remote" warehouse. For partitioning-based schemes, such as H-STORE, each partition consists of all the data for a single warehouse [135]. This means that the remote warehouse transactions will access multiple partitions.

We first execute the TPC-C workload on a 4-warehouse database with 100MB of data per warehouse (0.4GB in total). This allows us to evaluate the algorithms when there are more worker threads than warehouses. We then execute the same workload again on a 1024-warehouse database. Due to memory constraints of running in the Graphite simulator, we reduced the size of this database to 26MB of data per warehouse (26GB in total) by removing unimportant attributes from tables. This does not affect our measurements because the number of tuples accessed by each transaction is independent of the database size.

4 Warehouses

The results in Figure 6-16 show that all of the schemes fail to scale for TPC-C when there are fewer warehouses than cores. With H-STORE, the DBMS is unable to utilize extra cores because of its partitioning scheme; the additional worker threads are essentially idle. For the other schemes, the results in Figure 6-16c show that they are able to scale up to 64 cores for the `NewOrder` transaction.

TIMESTAMP, MVCC, and OCC have worse scalability due to high abort rates. DL_DETECT does not scale due to thrashing and deadlocks. But the results in Figure 6-16b show that no scheme scales for the Payment transaction. The reason for this is that every Payment transaction updates a single field in the warehouse (W_YTD). This means that either the transaction (1) must acquire an exclusive lock on the corresponding tuple (i.e., DL_DETECT) or (2) issue a pre-write on that field (i.e., T/O-based algorithms). If the number of threads is greater than the number of warehouses, then updating the warehouse table becomes a bottleneck.

In general, the main problem for these two transactions is the contention when updating the WAREHOUSE table. Each Payment transaction updates its corresponding warehouse entry and each NewOrder will read it. For the 2PL-based algorithms, these read and write operations block each other. Two of the T/O-based algorithms, TIMESTAMP and MVCC, outperform the other schemes because their write operations are not blocked by reads. This eliminates the lock blocking problem in 2PL. As a result, the NewOrder transactions can execute in parallel with Payment transactions.

1024 Warehouses

We next execute the TPC-C workload with 1024 warehouses with up to 1024 cores. Once again, we see in Figure 6-17 that no scheme is able to scale. The results indicate that unlike in Section 6.5.6, the DBMS's throughput is limited by NewOrder transactions. This is due to different reasons for each scheme.

With almost all the schemes, the main bottleneck is the overhead of maintaining locks and latches, which occurs even if there is no contention. For example, the NewOrder transaction reads tuples from the read-only ITEM table, which means for the 2PL schemes that each access creates a shared-lock entry in the DBMS. With a large number of concurrent transactions, the lock meta-data becomes large and thus it takes longer to update them. OCC does not use such locks while a transaction runs, but it does use latches for each tuple accessed during the validation phase. Acquiring these latches becomes an issue for transactions with large footprints, like NewOrder. Although MVCC also does not have locks, each read request generates a new history record, which increases memory traffic. We note, however, that all of this is technically unnecessary work because the ITEM table is never modified.

The results in Figure 6-17b indicate that when the number of warehouses is the same or greater than the number of worker threads, the bottleneck in the Payment transaction is eliminated. This improves the performance of all schemes. For T/O schemes, however, the throughput becomes too high at larger core counts and thus they are inhibited by timestamp allocation. As a result, they are unable to achieve higher than ~10 million txn/s. This is the same scenario as Figure 6-12a where 2PL outperforms T/O for short transactions.

H-STORE performs the best overall due to its ability to exploit partitioning even with ~12% multi-partition transactions in the workload. This corroborates results from previous studies that show that H-STORE outperforms other approaches when less than 20% of the workload comprises multi-partition transactions [120, 144]. At 1024 cores, however, it is limited by the DBMS's timestamp allocation.

6.6 Discussion

We now discuss the results of the previous sections and propose solutions to avoid these scalability issues for many-core DBMSs.

6.6.1 DBMS Bottlenecks

Our evaluation shows that all seven concurrency control schemes fail to scale to a large number of cores, but for different reasons and conditions. Table 6.2 summarizes the findings for each of the schemes. In particular, we identified several bottlenecks to scalability: (1) lock thrashing, (2) preemptive aborts, (3) deadlocks, (4) timestamp allocation, and (5) memory-to-memory copying.

2PL	DL_DETECT	Scales under low-contention. Suffers from lock thrashing.
	NO_WAIT	Has no centralized point of contention. Highly scalable. Very high abort rate.
	WAIT_DIE	Suffers from lock thrashing and timestamp bottleneck.
T/O	TIMESTAMP	High overhead from copying data locally. Non-blocking writes. Suffers from timestamp bottleneck.
	MVCC	Performs well w/ read-intensive workload. Non-blocking reads and writes. Suffers from timestamp bottleneck.
	OCC	High overhead for copying data locally. High abort cost. Suffers from timestamp bottleneck.
	H-STORE	The best algorithm for partitioned workloads. Suffers from multi-partition transactions and timestamp bottleneck.

Table 6.2: **A Summary of Bottlenecks** – A summary of the bottlenecks for each concurrency control scheme evaluated in Section 6.5.

Thrashing happens in any waiting-based algorithm. As discussed in Section 6.4.2, thrashing is alleviated by proactively aborting. This leads to the trade-off between aborts and performance. In general, the results in Section 6.5.2 showed that for high-contention workloads, a non-waiting deadlock prevention scheme (NO_WAIT) performs much better than deadlock detection (DL_DETECT).

Although no single concurrency control scheme performed the best for all workloads, one may outperform the others under certain conditions. Thus, it may be possible to combine two or more classes of algorithms into a single DBMS and switch between them based on the workload. For example, a DBMS could use DL_DETECT for workloads with little contention, but switch to NO_WAIT or a T/O-based algorithm when transactions are taking too long to finish due to thrashing. One could also employ a hybrid approach.

These results also make it clear that new hardware support is needed to overcome some of these bottlenecks. For example, all of the T/O schemes suffer from the timestamp allocation bottleneck when the throughput is high. Using the *atomic addition* method when the core count is large causes the worker threads to send many messages across the chip to modify the timestamp. We showed in Section 6.4.3 how the *clock* and *hardware counter* methods performed the best without the drawbacks of batching. Thus, we believe that they should be included in future CPU architectures.

We also saw that memory issues cause slowdown in some of the schemes. One way to alleviate this problem is to add a hardware accelerator on the CPU to do memory copying in the background. This would eliminate the need to load all data through the CPU's pipeline. We also showed in Section 6.4.1 how `malloc` was another bottleneck and that we were able to overcome it by developing our own implementation that supports dynamic pool resizing. But with a large number of cores, these pools become too unwieldy to manage in a global memory controller. We believe that future operating systems will need to switch to decentralized or hierarchical memory allocation, potentially with hardware support.

6.6.2 Multicore vs. Multi-node Systems

Distributed DBMSs are touted for being able to scale beyond what a single-node DBMS can support [135]. This is especially true when the number of CPU cores and the amount of memory available on a node is small. But moving to a multi-node architecture introduces a new performance bottleneck: *distributed transactions* [18]. Since these transactions access data that may not be on the same node, the DBMS must use an atomic commit protocol, such as *two-phase commit* [61]. The coordination overhead of such protocols inhibits the scalability of distributed DBMSs because the communication between nodes over the network is slow. In contrast, communication between threads in a shared-memory environment is much faster. This means that a single many-core node with a large amount of DRAM might outperform a distributed DBMS for all but the largest OLTP applications [144].

It may be that for multi-node DBMSs two levels of abstraction are required: a shared-nothing implementation between nodes and a multi-threaded shared-memory DBMS within a single chip. This hierarchy is common in high-performance computing applications. More work is therefore needed to study the viability and challenges of such hierarchical concurrency control in an OLTP DBMS.

6.7 Additional Related Work

The work in [139] is one of the first hardware analyses of a DBMS running an OLTP workload. Their evaluation focused on multi-processor systems, such as how to assign processes to processors to avoid bandwidth bottlenecks. Another study [126] measured CPU stall times due to cache misses in OLTP workloads. This work was later expanded in [14] and more recently by [143, 122].

With the exception of H-STORE [53, 72, 135, 151] and OCC [86], all other concurrency control schemes implemented in our test-bed are derived from the seminal surveys by Bernstein et al. [18, 20]. In recent years, there have been several efforts towards improving the shortcomings of these classical implementations [41, 77, 117, 144]. Other work includes standalone lock managers that are designed to be more scalable on multicore CPUs [124, 81]. We now describe these systems in further detail and discuss why they are still unlikely to scale on future many-core architectures.

Shore-MT [77] is a multi-threaded version of Shore [28] that employs a deadlock detection scheme similar to DL_DETECT. Much of the improvements in Shore-MT come from optimizing bottlenecks in the system other than concurrency control, such as logging [78]. The system still suffers from the same thrashing bottleneck as DL_DETECT on high contention workloads.

DORA is an OLTP execution engine built on Shore-MT [117]. Instead of assigning transactions to threads, as in a traditional DBMS architecture, DORA assigns threads to partitions. When a transaction needs to access data at a specific partition, its handle is sent to the corresponding thread for that partition where it then waits in a queue for its turn. This is similar to H-STORE's partitioning model, except that DORA supports multiple record-level locks per partition (instead of one lock per partition) [118]. We investigated implementing DORA in our DBMS but found that it could not be easily adapted and requires a separate system implementation.

The authors of Silo [144] also observed that global critical sections are the main bottlenecks in

OCC. To overcome this, they use a decentralized validation phase based on *batched atomic addition* timestamps. But as we showed in Section 6.4.3, the DBMS must use large batches when deployed on a large number of cores to amortize the cost of centralized allocation. This batching in turn increases the system's latency under contention.

Hekaton [41] is a main memory table extension for Microsoft's SQL Server that uses a variant of MVCC with lock-free data structures [95]. The administrator designates certain tables as in-memory tables that are then accessed together with regular, disk-resident tables. The main limitation of Hekaton is that timestamp allocation suffers from the same bottleneck as the other T/O-based algorithms evaluated in this chapter.

The VLL centralized lock manager uses per-tuple 2PL to remove contention bottlenecks [124]. It is an optimized version of DL_DETECT that requires much smaller storage and computation overhead than our implementation when the contention is low. VLL achieves this by partitioning the database into disjoint subsets. Like H-STORE, this technique only works when the workload is partitionable. Internally, each partition still has a critical section that will limit scalability at high contention workloads.

The work in [81] identified latch contention as the main scalability bottleneck in MySQL. They removed this contention by replacing the *atomic-write-after-read* synchronization pattern with a *read-after-write* scheme. They also proposed to pre-allocate and deallocate locks in bulk to improve scalability. This system, however, is still based on centralized deadlock detection and thus will perform poorly when there is contention in the database. In addition, their implementation requires the usage of global barriers that will be problematic at higher core counts.

Others have looked into using the software-hardware co-design approach for improving DBMS performance. The "bionic database" project [76] focuses on implementing OLTP DBMS operations in FPGAs instead of new hardware directly on the CPU. Other work is focused on OLAP DBMSs and thus is not applicable to our problem domain. For example, an FPGA-based SQL accelerator proposed in [40] filters in-flight data moving from a data source to a data sink. It targets OLAP applications by using the FPGA to accelerate the projection and restriction operations. The Q100 project is a special hardware co-processor for OLAP queries [154]. It assumes a column-oriented database storage and provides special hardware modules for each SQL operator.

Chapter 7

TicToc Concurrency Control

7.1 Introduction

In the previous chapter, we saw how traditional concurrency control algorithms failed to scale in many-core shared-memory systems due to various fundamental scalability bottlenecks. In this chapter, we present a new concurrency control algorithm that eliminates these bottlenecks. Interestingly, the fundamental idea behind the new concurrency control algorithm is the same as the one behind Tardis cache coherence protocol (Chapters 3, 4 and 5), namely, logical leases. Different from Tardis which applies logical leases to multicore hardware, in this chapter we apply logical leases to software, and more specifically, to concurrency control algorithms.

The new concurrency control algorithm we developed is called **TicToc** which combines ideas from both Timestamp Ordering (T/O) and Optimistic Concurrency Control (OCC) algorithms. It can achieve higher concurrency than state-of-the-art T/O schemes and completely eliminates the timestamp allocation bottleneck, which is a main obstacle of scaling as discussed in Chapter 6. The key idea of TicToc is a technique that we call *data-driven timestamp management*: instead of assigning timestamps to each transaction independently of the data it accesses, TicToc embeds the necessary timestamp information (in the form of logical leases) in each tuple to enable each transaction to compute a valid commit timestamp after it has run, right before it commits. This approach has two benefits. First, each transaction infers its timestamp from metadata associated with each tuple it reads or writes. No centralized timestamp allocator exists, and concurrent transactions accessing disjoint data do not communicate. Second, by determining timestamps *lazily* at commit time, TicToc finds a logical-time order that enforces serializability even among transactions that overlap in physical time and would cause aborts in other T/O-based protocols. In essence, Tic-

Toc allows commit timestamps to *move in logical time* to uncover more concurrency than existing schemes without violating serializability.

We present a high-performance, OCC-based implementation of TicToc, and prove that it enforces serializability. We also design several optimizations that further improve TicToc's scalability. Finally, we compare TicToc with four other modern concurrency control schemes in the **DBx1000** main-memory DBMS [2], using two different OLTP workloads on a multi-socket, 40-core system. Our results show that TicToc achieves up to 92% higher throughput than prior algorithms under a variety of workload conditions.

7.2 The TicToc Algorithm

Like other T/O-based algorithms, TicToc uses timestamps to indicate the serial order of the transactions. But unlike these previous approaches, it does not assign timestamps to transactions using a centralized allocator. Instead, a transaction's timestamp is calculated *lazily* at its commit time in a *distributed* manner based on the tuples it accesses. There are two key advantages of this timestamp management policy. First, its distributed nature avoids all of the bottlenecks inherent in timestamp allocation [157], making the algorithm highly scalable. Second, laziness makes it possible for the DBMS to find the most appropriate order among transactions that can minimize aborts.

7.2.1 Lazy Timestamp Management

To see why lazy timestamp management can reduce conflicts and improve performance, we consider the following example involving two concurrent transactions, A and B, and two tuples, x and y. The transactions invoke the following sequence of operations:

1. A read(x)
2. B write(x)
3. B commits
4. A write(y)
5. A commits

This interleaving of operations does not violate serializability because the end result is identical to sequentially executing A before B, even though A commits after B in physical time order.

Traditional T/O algorithms assign timestamps to transactions statically, essentially agreeing on a fixed sequential schedule for concurrent transactions. This eases conflict detection, but limits concurrency. In this example, if transaction A is assigned a lower timestamp than transaction B, then both transactions can commit since the interleaving of operations is consistent with the timestamp order. However, if transaction A is assigned a higher timestamp than transaction B, A must eventually abort since committing it would violate the schedule imposed by timestamp order. In practice, the timestamp allocation process does not know the data access pattern of transactions. What timestamp a transaction gets may be completely random, making it very hard to avoid unnecessary aborts due to bad timestamp allocation.

By contrast, TicToc does not allocate timestamps statically, so it does not restrict the set of potential orderings. It instead calculates the timestamp of each transaction lazily at the transaction's commit time by inspecting the tuples it accessed. In our example, when transaction A reaches its commit point, TicToc calculates the commit timestamp using the versions of the tuples x and y it actually reads/writes rather than their latest version in the database, which might have changed since the transaction accesses those tuples. And since the version of tuple x read by A is older than the one written by B, A will be ordered before B and both transactions can commit.

To encode the serialization information in each tuple, each data version in TicToc has a logical lease represented using the write timestamp (*wts*) and the read timestamp (*rts*). Conceptually, a particular version is created at timestamp *wts* and is valid until timestamp *rts*, in logical time. A version read by a transaction is valid if and only if that transaction's commit timestamp is in between the version's *wts* and *rts*. And a write by a transaction is valid if and only if the transaction's commit timestamp is greater than the *rts* of the previous version. Formally, the following invariant must hold for transaction T to commit:

$$\begin{aligned}
&\exists\ commit\_ts,\\
&\quad (\forall t \in \{tuples\ read\ by\ T\}, t.wts \leq commit\_ts \leq t.rts)\\
&\quad \wedge(\forall t \in \{tuples\ written\ by\ T\}, t.rts < commit\_ts)
\end{aligned}$$

This policy leads to serializable execution because all the reads and writes within a transaction occur at the same timestamp. Intuitively, the commit timestamp order determines the serialization order. A read always returns the version valid at that timestamp and a write is ordered after all the reads to older versions of the tuple. We will prove that TicToc follows serializability in Section 7.3.

7.2.2 Protocol Specification

Like standard OCC algorithms, each transaction in TicToc accesses the database without acquiring locks during normal operation. This is known as the *read phase*. When the transaction invokes the commit operation, the protocol then takes the transaction through the *validation phase* to check whether it should be allowed to commit. If it does, then it enters the *write phase* where the transaction's changes are applied to the shared database.

We now discuss these phases in further detail.

Read Phase

The DBMS maintains a separate read set and write set of tuples for each transaction. During this phase, accessed tuples are copied to the read set and modified tuples are written to the write set, which is only visible to the current transaction. Each entry in the read or write set is encoded as $\{tuple, data, wts, rts\}$, where *tuple* is a pointer to the tuple in the database, *data* is the data value of the tuple, and *wts* and *rts* are the timestamps copied from the tuple when it was accessed by the transaction. For a read set entry, TicToc maintains the invariant that the data version is valid from *wts* to *rts* in logical time; namely, no transaction can write to the tuple at a timestamp between the tuple's current *wts* and *rts*.

Algorithm 4 shows the procedure for a tuple access request in the read phase. A new entry in the read set is allocated. The pointer to the accessed tuple, data value and timestamps of the tuple are stored in the entry. Note that the data value and timestamps must be loaded atomically to guarantee that the value matches the timestamps. We explain in Section 7.2.6 how to efficiently perform this atomic operation in a lock-free manner.

Algorithm 4: Read Phase of TicToc.

```
Data: read set RS, tuple t
1 r = RS.get_new_entry()
2 r.tuple = t
  # Atomically load wts, rts, and value
3 < r.value = t.value, r.wts = t.wts, r.rts = t.rts >
```

Validation Phase

In the validation phase, TicToc uses the timestamps stored in the transaction's read and write sets to compute its commit timestamp. Then, the algorithm checks whether the tuples in the transaction's read set are valid based on this commit timestamp.

The first step for this validation, shown in Algorithm 5, is to lock all the tuples in the transaction's write set in their primary key order to prevent other transactions from updating the same tuples concurrently. Using this fixed locking order guarantees that there are no deadlocks with other

```
Algorithm 5: Validation Phase
  Data: read set RS, write set WS
  # Step 1 – Lock Write Set
  for w in sorted(WS) do
      lock(w.tuple)
  end
  # Step 2 – Compute the Commit Timestamp
  commit_ts = 0
  for e in WS ∪ RS do
      if e in WS then
          commit_ts = max(commit_ts, e.tuple.rts + 1)
      else
          commit_ts = max(commit_ts, e.wts)
      end
  end
  # Step 3 – Validate the Read Set
  for r in RS do
      if r.rts < commit_ts then
          # Begin atomic section
          if r.wts ≠ r.tuple.wts or (r.tuple.rts ≤ commit_ts and isLocked(r.tuple) and r.tuple not
            in W) then
              abort()
          else
              r.tuple.rts = max(commit_ts, r.tuple.rts)
          end
          # End atomic section
      end
  end
```

transactions committing at the same time. This technique is also used in other OCC algorithms (e.g., Silo [144]).

The second step in the validation phase is to compute the transaction's commit timestamp using the *wts* and *rts* of each accessed tuple, which are available in the read and write sets. As discussed in Section 7.2.1, for a tuple in the read set but not in the write set, the commit timestamp should be no less than the tuple's *wts* since the tuple would have a different version before this timestamp. For a tuple in the transaction's write set, however, the commit timestamp needs to be no less than its current $rts + 1$ since the previous version was valid till *rts*.

In the last step, the algorithm validates the tuples in the transaction's read set. If the transaction's *commit_ts* is less than or equal to the *rts* of the read set entry, then the invariant $wts \leq commit\_ts \leq rts$ holds. This means that the tuple version read by the transaction is valid at *commit_ts*, and thus no further action is required. If the entry's *rts* is less than *commit_ts*, however, it is not clear whether the local value is valid or not at *commit_ts*. It is possible that another transaction has modified the tuple at a logical time between the local *rts* and *commit_ts*, which means the transaction has to abort. It is also possible that no other transaction has modified the tuple, in which case the *rts* can be extended to *commit_ts*, making the version valid at *commit_ts*.

Specifically, the local *wts* is first compared to the latest *wts*. If they are different, the tuple has already been modified by another transaction and thus it is impossible to extend the *rts* of the local version. If *wts* matches, but the tuple is already locked by a *different* transaction (i.e., the tuple is locked but it is not in the transaction's write set), it is not possible to extend the *rts* either. If the *rts* is extensible, the *rts* of the tuple can be extended to *commit_ts*. Note that the whole process must be done atomically to prevent interference from other transactions. The DBMS does not validate tuples that are only in the write set but not in the read set, since they are already protected by the locks acquired at the beginning of the validation phase, and no other transactions can possibly change a locked tuple's timestamps.

In TicToc, there is no centralized contention point during transaction execution. The locks and atomic sections protect only the tuples that a transaction touches. In Section 7.4, we present optimizations to further reduce the contention caused by these operations.

Write Phase

Finally, if all of the tuples that the transaction accessed pass validation, then the transaction enters the write phase. As shown in Algorithm 6, in this phase the transaction's write set is written to the database. For each tuple in the transaction's write set, the DBMS sets both its *wts* and *rts* to *commit_ts*, indicating a new version of the tuple. All locks that were acquired during the validation phase are then released, making the tuple accessible to all other transactions.

Algorithm 6: Write Phase

```
Data: write set WS, commit timestamp commit_ts
for w in WS do
    write(w.tuple.value, w.value)
    w.tuple.wts = w.tuple.rts = commit_ts
    unlock(w.tuple)
end
```

7.2.3 Example

We now revisit the example in Section 7.2.1 and explain how TicToc is able to commit both transactions *A* and *B* even though previous OCC algorithms could not. Figure 7-1 shows a step-by-step

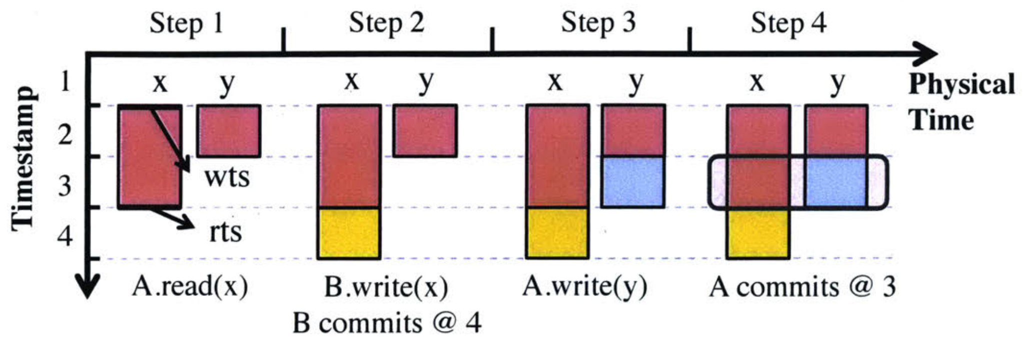


Figure 7-1: **An Example of Two Transactions Executing in TicToc** – The logic of the two transactions are shown in Section 7.2.1.

diagram. In this example, one operation occurs in each physical step. The *wts* and *rts* for tuples *x* and *y* are encoded as the start and end point of the vertical bands.

Step 1: Transaction *A* reads tuple *x*. The current version of *x* and its timestamps ($wts = 2$ and $rts = 3$) are stored in *A*'s read set.

Step 2: Transaction *B* writes to tuple *x* and commits at timestamp 4. The version of *x* will be overwritten and both the *wts* and *rts* of the new version will become 4.

Step 3: Transaction *A* writes to tuple *y*. Since the previous version of *y* has $rts = 2$, the new version can be written at timestamp 3. At this point, the new version of *y* is only stored in the write set of transaction *A* and is not visible to other transactions yet.

Step 4: Transaction *A* enters the validation phase. According to Algorithm 5, the commit timestamp should be the maximum of the read tuple's *wts* and write tuple's $rts + 1$, which is timestamp 3 in this example. Then, transaction *A* validates the read set by checking whether the version of tuple *x* it read is valid at timestamp 3. Since transaction *A*'s version of tuple *x* is valid from timestamp 2 to 3, it passes the validation phase and commits.

Note that when the DBMS validates transaction *A*, it is not even aware that tuple *x* has been modified by another transaction. This is different from existing OCC algorithms (including Hekaton [95] and Silo [144]), which always recheck the tuples in the read set. These algorithms would abort transaction *A* in this example because tuple *x* has already been modified since transaction *A* last read it. TicToc is able to commit such transactions because it uses logical timestamps to determine the serialization order, which can be different from the physical time order in which transactions commit.

7.2.4 Discussion on Aborts

A transaction may not always be able to validate its read set during the validation phase, which leads to aborts. For example, if tuple y in Figure 7-1 was originally valid from timestamps 1 to 4, then transaction A's *commit_ts* has to be 5. Since x's *rts* cannot be extended to 5, A has to abort. In this case, transaction A aborts not because of the conflicts between transactions A and B, but because of the timestamps.

In general, the reason that these aborts occur is because serializability *may be* violated due to other concurrently running transactions that change the tuples' timestamps. For the example above, imagine that there exists a transaction C reading tuple x and y after B commits but before A commits. C is able to commit at timestamp 4 as it observes B's write to x and the original value of y. C will extend the *rts* of y to 4. This now means that A cannot commit without violating serializability because there is a dependency cycle between A, B and C[1]. Note that when A enters the validation phase, it does not know whether C exists or not. The way timestamps are managed in TicToc guarantees serializability, which requires certain transactions to be aborted.

7.2.5 Discussion

Beyond scalability and increased concurrency, TicToc's protocol has two other distinguishing features. Foremost is that the transaction's logical commit timestamp order may not agree with the physical commit time order. In the example shown in Figure 7-1, transaction A commits physically after transaction B, but its commit timestamp is less than transaction B's commit timestamp. This means that A precedes B in the serial schedule. The flexibility to move transactions back and forth in logical time allows TicToc to commit more transactions that would abort in other concurrency control algorithms.

Another feature of TicToc is that logical timestamps grow more slowly than the number of committed transactions. Moreover, the rate at which the logical timestamp advances is an indicator of the contention level in the workload. This is because different transactions may commit with the same logical timestamp. Such a scenario is possible if two transactions have no conflicts with each other, or if one transaction reads a version modified by the other transaction. At one extreme, if all transactions are read-only and thus there is no contention, all transactions will have the same commit timestamp (i.e., they all commit at timestamp 0). At the other extreme, if all the transactions update

[1] $A<B$ due to write-after-read on x, $B<C$ due to read-after-write on x, and $C<A$ due to write-after-read on y.

the same tuple, each commit would increase the tuple's *wts* by one, and the logical timestamp would increase at the same rate as the number of committed transactions. Since most OLTP workloads have some contention, the DBMS's logical timestamps will increase more slowly than the number of committed transactions; the higher the contention, the faster logical timestamps advance. We will show this in Section 7.5.5.

7.2.6 Implementation

As shown in Algorithms 4 and 5, both the read and validation phases require the DBMS to *atomically* read or write tuples' timestamps. But implementing these atomic sections using locks would degrade performance. To avoid this problem, TicToc adopts an optimization from Silo [144] to encode a lock bit and a tuple's *wts* and *rts* into a single 64-bit word (`TS_word`) in the following form:

`TS_word [63]`: *lock bit* (1 bit).
`TS_word [62:48]`: $delta = rts - wts$ (15 bits).
`TS_word [47:0]`: *wts* (48 bits).

The highest-order bit is used as the lock bit. The *wts* is stored as a 48-bit counter. To handle *wts* overflows, which happens at most once every several weeks for the most active workloads, the tuples in the database are periodically loaded in the background to reset their *wts*. This process is infrequent and can be performed concurrently with normal transactions, so its overhead is negligible.

Algorithm 7 shows the lock-free implementation to *atomically* load the data value and timestamps for a tuple, which is part of Algorithm 4. `TS_word` is loaded twice, before and after loading the data. If these two `TS_word` instances are the same and both have the lock bit unset, then the data value must not have changed and is still consistent with the timestamps. Otherwise, the process is repeated until the two loads of `TS_word` are consistent. There are no writes to shared memory during this process. To avoid starvation, one could revert to more heavy-weight latching if this check repeatedly fails.

Algorithm 7: Atomically Load Tuple Data and Timestamps

```
Data: read set entry r, tuple t
1 do
2     v1 = t.read_ts_word()
3     read(r.data, t.data)
4     v2 = t.read_ts_word()
5 while v1 ≠ v2 or v1.lock_bit == 1;
6 r.wts = v1.wts
7 r.rts = v1.wts + v1.delta
```

Similarly, Algorithm 8 shows the steps to atomically extend a tuple's *rts* in TicToc's validation phase (Algorithm 5). Recall that this operation is called if *commit_ts* is greater than the local *rts*; the DBMS makes the local version valid at *commit_ts* by extending the *rts* of the tuple. The first part of the algorithm is the same as explained in Algorithm 3; validation fails if the tuple's *rts* cannot possibly be extended to *commit_ts*.

Algorithm 8: Read-Set Validation

```
Data: read set entry r, write set W, commit timestamp commit_ts
1 do
2     success = true
3     v2 = v1 = r.tuple.read_ts_word()
4     if r.wts ≠ v1.wts or (v1.rts ≤ commit_ts and isLocked(r.tuple)) and r.tuple not in W then
5         Abort()
6     end
      # Extend the rts of the tuple
7     if v1.rts ≤ commit_ts then
          # Handle delta overflow
8         delta = commit_ts − v1.wts
9         shift = delta − delta ∧ 0x7fff
10        v2.wts = v2.wts + shift
11        v2.delta = delta − shift
          # Set the new TS word
12        success = compare_and_swap(r.tuple.ts_word, v1, v2)
13    end
14 while not success;
```

Since we only encode *delta* in 15 bits in `TS_word`, it may overflow if *rts* and *wts* grow far apart. If an overflow occurs (line 8-11), we increase *wts* to keep *delta* within 15 bits. Increasing *wts* this way does not affect the correctness of TicToc. Intuitively, this is because increasing the *wts* can be considered a dummy write to the tuple at the new *wts* with the same data. Inserting such a dummy write does not affect serializability. Increasing *wts*, however, may increase the number of aborts since another transaction may consider the version as being changed while it has not actually changed. This effect is more problematic if *delta* uses fewer bits. Our experiments indicate that 15 bits is enough for the overflow effect to be negligible.

Finally, the new *wts* and *delta* are written to a new `TS_word` and atomically applied to the tuple. The DBMS uses an atomic *compare-and-swap* instruction to make sure that the `TS_word` has not been modified by other transactions simultaneously.

7.2.7 Logging and Durability

The TicToc algorithm can support logging and crash recovery in a similar way as traditional concurrency control algorithms. The DBMS can use the canonical ARIES approach if there is only a single log file [113]. But ARIES cannot provide the bandwidth required in today's multicore systems [162]. Implementing arbitrarily scalable logging is out of the scope of this thesis and is left for future work. In this section, we briefly discuss one idea of implementing parallel logging with multiple log files on TicToc.

Parallel logging has been studied in other DBMSs [147, 162]. The basic idea is to perform logging in batches. All transactions in a previous batch must be ordered before any transaction in a later batch, but the relative ordering among transactions within the same batch can be arbitrary. For each batch, logs are written to multiple files in parallel. A batch is considered durable only after all the logs within that batch have been written to files.

In TicToc, the batching scheme requires that transactions in a later batch must have commit timestamps greater than transactions in a previous batch. This can be achieved by setting a minimum commit timestamp for transactions belonging to the new batch. To start a new batch, each worker thread should coordinate to compute the minimum commit timestamp that is greater than the commit timestamps of all transactions in previous batches. Each transaction in the new batch has a commit timestamp greater than this minimum timestamp. Setting a minimum timestamp does not affect the correctness of TicToc since timestamps only increase in this process, and transactions are properly ordered based on their timestamps. The performance of this parallel logging scheme should be the same as with other concurrency control algorithms [147, 162].

7.3 Proof of Correctness

In this section, we prove that the TicToc algorithm enforces serializability.

7.3.1 Proof Idea

To prove that a schedule is serializable in TicToc, we need to show that the schedule is equivalent to another schedule where all the transactions are executed sequentially. Previous T/O concurrency control algorithms use the transactions' unique timestamps to determine the serial order. In TicToc, however, the commit timestamp is derived from the accessed tuples and no global coordination takes place, and thus two transactions may commit with the same timestamp. Therefore, transactions cannot be fully ordered based on their commit timestamps alone.

We show that in TicToc, the equivalent sequential order is actually the physiological time (cf. Section 3.3) order among transactions. More formally, we define the global sequential order in Definition 8.

Definition 8 (Serial Order). *Using $<_s$, $<_{ts}$ and $<_{ps}$ to indicate serial order, commit timestamp order, and physical commit time order, respectively, the serial order between transaction A and B is defined as follows:*

$$A <_s B \triangleq A <_{ts} B \vee (A =_{ts} B \wedge A \leq_{pt} B)$$

The serial order defined in Definition 8 is a total order among all the transactions. Transaction A is ordered before transaction B if A has a smaller commit timestamp or if they have the same commit timestamp but A commits before B in physical time. If A and B both have the same logical and physical commit time, then they can have arbitrary serial order, because they do not conflict.

The goal of the correctness proof is summarized as follows:

Theorem 6. *Any schedule in TicToc is equivalent to the serial schedule defined in Definition 8.*

To prove this, we show that a read in the actual schedule always returns the value of the last store in the serial schedule. We also prove that transactions having the same commit timestamp *and* physical commit time do not conflict.

7.3.2 Formal Proofs

We first prove a useful lemma that will be used to prove subsequent Lemmas 22 and 23.

Lemma 21. *Transactions writing to the same tuple must have different commit timestamps.*

Proof. According to Algorithms 5 and 6, a tuple is locked while being written, therefore only one transaction can write to that tuple at any time. According to line 3 in Algorithm 6, both *wts* and *rts* of the modified tuple become its commit timestamp.

According to line 7 in Algorithm 5, if another transaction writes to the same tuple at a later time, its commit timestamp must be strictly greater than the tuple's current *rts*. Since *rts* never decreases in the TicToc algorithm, the commit timestamp of the later transaction must be greater than the commit timestamp of the earlier transaction. Therefore, transactions writing to the same tuple must have different commit timestamps. □

Two properties must be true in order to prove Theorem 6. First, transactions that have the same commit timestamp and physical time must not conflict. Second, a read always returns the latest write in the serial schedule. We now prove these two properties of TicToc:

Lemma 22. *Transactions that commit at the same timestamp and physical time do not conflict with each other.*

Proof. According to Lemma 21, write-write conflicting transactions must have different commit timestamps. Therefore, we only need to show that all read-write or write-read conflicting transactions commit at different logical *or* physical time.

Consider a pair of transactions committing at the same physical time. One reads a tuple and the other writes to the same tuple. Then, the commit timestamp of the reading transaction must be less than or equal to the tuple's current *rts*. And the commit timestamp of the writing transaction must be greater than the tuple's current *rts*. Therefore, they have different commit timestamps. □

Lemma 23. *A read operation from a committed transaction returns the value of the latest write to the tuple in the serial schedule.*

Proof. We first prove that if a committed transaction's read observes another transaction's write, then the reading transaction must be ordered after the writing transaction in the serial schedule.

A tuple's *wts* is always updated together with its value, and the *wts* is always the commit timestamp of the transaction that writes the value. Line 9 in Algorithm 5 states that if another transaction reads the value, then its commit timestamp must be greater than or equal to *wts*. If the commit timestamp equals *wts*, then the reading transaction still commits after the writing transaction in physical time because the writing transaction only makes its writes globally visible after its physical commit time. By Definition 8, the reading transaction is always ordered after the writing transaction in the serial schedule.

We next show that the write observed by the following read is the latest write in the serial schedule. In other words, if the writing transaction has timestamp ts_1 and the reading transaction has timestamp ts_2, no other write to the same tuple commits at timestamp ts such that $ts_1 \leq ts \leq ts_2$.

According to Algorithm 5, when the reading transaction commits, it can observe a consistent view of the tuple's `TS_word` with *wts* and *rts*, where $ts_1 = wts$ and $ts_2 \leq rts$. This implies that so far in physical time, no write to the same tuple has happened between t_1 and t_2 in logical time because otherwise the *wts* of the tuple would be greater than t_1. No such write can happen in the future either, because all future writes will have timestamps greater the tuple's *rts* and thus greater than t_2. □

Proof of Theorem 6. According to Lemma 22, transactions with the same commit timestamp and physical commit time do not conflict. Thus, they can have any serial order among them and the result of the serial schedule does not change.

According to Lemma 23, for transactions with different commit timestamps or physical commit time, a read in a transaction always returns the latest write in the serial schedule. According to Lemma 21, only one such latest write can exist so there is no ambiguity. Then, for each transaction executed in the actual schedule, all the values it observes are identical to the values it would observe in the serial schedule. Hence, the two schedules are equivalent. □

7.4 Optimizations

The TicToc algorithm as presented so far achieves good performance when tested on a multi-socket shared memory system (cf. Section 7.5). There are, however, still places in the validation phase that may create unnecessary contention and thus hurt performance. For example, locking the transaction's write set during the validation phase may cause thrashing for write-intensive benchmarks.

In this section, we discuss several optimizations that we developed for TicToc to minimize contention. We also discuss how TicToc works for weaker isolation levels for those applications that find serializability too strong.

7.4.1 No-Wait Locking in Validation Phase

In TicToc's validation phase (Algorithm 5), tuples in a transaction's write set are locked following the primary key order. The process of locking the write set is essentially a Two-Phase-Locking (2PL) protocol. As discussed in Chapter 6, lock waiting leads to lock thrashing at high core counts,

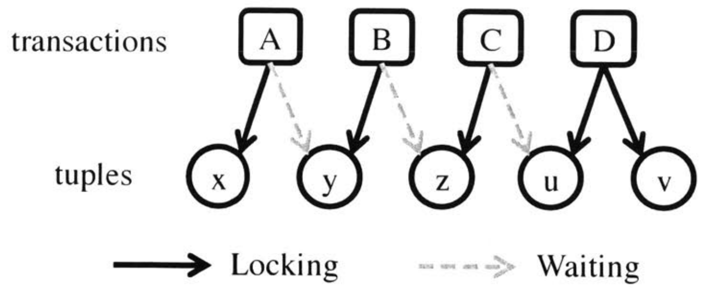


Figure 7-2: **An Example of Lock Thrashing.**

leading to significant performance degradation, even if the locks are acquired in the primary key order [157]. Thrashing happens when a transaction already holds locks and waits for the next lock. The locks it already holds, however, may block other transactions.

Consider a pathological case shown in Figure 7-2 where each transaction tries to lock two tuples. Transaction *D* has already acquired both of the locks that it needs, while transactions *A*, *B*, and *C* are waiting for locks held by other transactions. When transaction *D* commits, *C* is able to acquire the lock and make forward progress. Transactions *A* and *B*, however, still need to wait. The end result is that the four transactions commit sequentially.

Note that, in this particular example, it is actually possible to abort transactions *A* and *C* so that *B* and *D* can acquire the locks and run in parallel. After they finish, *A* and *C* can also run in parallel. This schedule only takes half the execution time compared to the pathological schedule. In practice, however, it is very challenging to quickly find out what transactions to abort at runtime to maximize parallelism.

A better approach to avoid the thrashing problem is to use a 2PL variant based on non-waiting deadlock prevention in TicToc's validation phase [18]. This protocol optimization, which we refer to as **No-Wait**, is like running a mini concurrency control algorithm inside of the TicToc algorithm. With No-Wait, if a transaction fails to acquire a lock for a tuple in its write set during the validation phase, the validation is immediately aborted by releasing all locks the transaction already holds. Then, TicToc restarts the validation phase. The transaction sleeps for a short period (1 μs) before retrying to reduce thrashing. Our experiments show that the algorithm's performance is not overly sensitive to the length of this sleep time as long as it is not too large.

The No-Wait optimization minimizes lock thrashing and allows more transactions to validate simultaneously. In the example in Figure 7-2, if No-Wait is used, then *A* or *B* may run in parallel with *D*.

7.4.2 Preemptive Aborts

The first step of the validation phase (cf. Algorithm 5) locks the tuples in the transaction's write set before it examines the read set. If a transaction ends up aborting because the read set validation fails, then this locking becomes an unnecessary operation. Furthermore, these locks may block other transactions from validating, which hurts overall performance. We observe that for some transactions the decision to abort can actually be made before locking the write set tuples. We call this optimization **preemptive abort**. A transaction that aborts preemptively does not need to go through the whole validation phase. This saves CPU time and reduces contention.

To better understand this, consider a transaction with one tuple in its read set. This transaction will fail the validation phase if this tuple's local *rts* is less than the transaction's commit timestamp and its local *wts* of the tuple in the read set does not match the tuple's latest *wts* in the shared database. A tuple's latest *wts* can be atomically read from the `TS_word` of the tuple. The transaction's commit timestamp, however, cannot be accurately determined before the write set is locked because a tuple's *rts* in the write set might be changed by a different transaction. The key observation here is that we just need to find an approximate commit timestamp. Thus, this optimization allows us to abort some transactions early without executing all the logic in the validation phase.

We compute the approximate commit timestamp using the local *wts* and *rts* in the read and write sets. For each tuple in the read set, the approximate commit timestamp is no less than the tuple's local *wts*; for each tuple in the write set, the approximate commit timestamp is no less than the tuple's local $rts + 1$. Note that the actual commit timestamp is no less than the approximate commit timestamp, because the latest timestamps in the tuples cannot be less than the local timestamps. Once an approximate commit timestamp is determined, it is used to decide if the transaction should be preemptively aborted.

7.4.3 Timestamp History

TicToc always aborts a transaction if its local *rts* of a tuple is less than *commit_ts* and the local *wts* does not match the latest *wts*. There are some cases, however, where the latest *wts* is greater than the transaction's *commit_ts* but the local version is still valid at *commit_ts*. Such transactions may commit without violating serializability, but the TicToc algorithm discussed so far always aborts them.

Figure 7-3 shows such an example. Transaction A first reads tuple x with $wts = 2$ and $rts = 2$.

Later, tuple *x*'s *rts* is extended to timestamp 3 due to the validation of transaction *B*. Then, tuple *x* is modified by transaction *C* and thus the latest *wts* and *rts* both become 4. Finally, transaction *A* enters the validation phase and validates tuple *x* at $commit\_ts = 3$ (not 2 because transaction *A* accessed other tuples not shown in the figure). At this point, transaction *A* only has the local timestamps of *x* ($wts = 2$ and $rts = 2$) and knows that the local version is valid at timestamp 2, but does not know if it is still valid at timestamp 3. From transaction *A*'s perspective, it is possible that the local version has been extended to timestamp 3 by some other transaction; it is also possible, however, that some other transaction did a write that is only valid at timestamp 3. Based on all the information transaction *A* has, these two situations are indistinguishable.

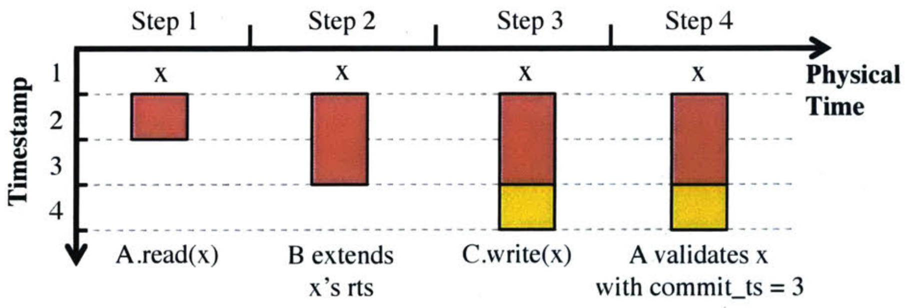


Figure 7-3: **An Example of the Timestamp History Optimization** – The DBMS uses a tuple's timestamp history to avoid aborting a transaction.

To prevent these unnecessary aborts, we can extend TicToc to maintain a history of each tuple's *wts* rather than just one scalar value. When a new version is created for a tuple, the *wts* of the old version is stored in a structure called the *history buffer*. The value of the old version of the tuple does not need to be stored since transactions in TicToc always read the latest data version. Therefore, the storage overhead of this optimization is just a few bytes per tuple. In our implementation, the history buffer is a per-tuple array that keeps a fixed number of the most recent *wts*'s. This way, the DBMS does not perform garbage collection.

During a transaction's validation phase, if the *rts* of a tuple in the read set is less than *commit_ts* and the *wts* does not match the latest *wts*, the DBMS checks if the *wts* matches any version in the tuple's history buffer. If so, the valid range of that version is from the local *wts* to the next *wts* in the history buffer. If *commit_ts* falls within that range, the tuple can pass validation. This way, some unnecessary aborts can be avoided.

7.4.4 Lower Isolation Levels

The TicToc algorithm described in Section 7.2 provides serializable isolation, which is the strictest isolation level in ANSI SQL. With minimal changes, TicToc can also support lower isolation levels for those applications that are willing to sacrifice isolation guarantees in favor of better performance and lower abort rate.

Snapshot Isolation: This level mandates that all of a transaction's reads see a consistent snapshot of the database, and that the transaction will commit only if it does not conflict with any concurrent updates made since that snapshot. In other words, all the read operations should happen at the same timestamp (*commit_rts*) and all the write operations should happen at a potentially later timestamp (*commit_wts*), and the written tuples are not modified between *commit_rts* and *commit_wts*.

To support snapshot isolation, instead of using a single *commit_ts* and verifying that all reads and writes are valid at this timestamp, two commit timestamps are used, one for all the reads (*commit_rts*) and one for all the writes (*commit_wts*). The algorithm verifies that all reads are valid at *commit_rts* and all writes are valid at *commit_wts*. It also guarantees that before the transaction writes to a tuple, its previous *wts* is less than or equal to *commit_rts*. All of these can be implemented with minor changes to Algorithm 5.

Repeatable Reads: With this weaker isolation level, a transaction's reads do not need to happen at the same timestamp even though writes should still have the same commit timestamp. This means there is no need to verify the read set in the validation phase. For a tuple read and updated by the same transaction, however, the DBMS still needs to guarantee that no other updates happened to that tuple since the transaction last read the value.

7.5 Experimental Evaluation

We now present our evaluation of the TicToc algorithm. For these experiments, we use the same DBx1000 OLTP DBMS [2] discussed in Chapter 6.

Since DBx1000 includes a pluggable lock manager that supports different concurrency control schemes, we can easily compare different concurrency control algorithms in the same system. Specifically, we implemented five concurrency control algorithms for our evaluation.

TICTOC:	Time traveling OCC with all optimizations
SILO:	Silo OCC [144]

HEKATON:	Hekaton MVCC [95]
DL_DETECT:	2PL with deadlock detection
NO_WAIT:	2PL with non-waiting deadlock prevention

We deployed DBx1000 on a 40-core machine with four Intel Xeon E7-4850 CPUs and 128 GB of DRAM. Each core supports two hardware threads, for a total of 80 threads. The experiments with more than 40 threads (shaded areas in the throughput graphs) use multiple threads per core, and thus may scale sub-linearly due to contention. To minimize memory latency, we use `numactl` to ensure each thread allocates memory from its own socket.

7.5.1 Workloads

We next describe the two benchmarks that we implemented in the DBx1000 DBMS for this analysis.

TPC-C: This workload is the current industry standard to evaluate OLTP systems [140]. It consists of nine tables that simulate a warehouse-centric order processing application. Only two (`Payment` and `NewOrder`) out of the five transactions in TPC-C are modeled in our evaluation in this section, with the workload comprised of 50% of each type. These two transaction types make up 88% of the default TPC-C mix and are the most interesting in terms of complexity for our evaluation.

YCSB: The Yahoo! Cloud Serving Benchmark is representative of large-scale on-line services [35]. Each query accesses a single random tuple based on a power law distribution with a parameter (*theta*) that controls the contention level in the benchmark [63]. We evaluate three different variations of this workload:

1. **Read-Only:** A transaction contains only read queries following a uniform access distribution (*theta*=0).
2. **Medium Contention:** 16 queries per transaction (90% reads and 10% writes) with a hotspot of 10% tuples that are accessed by ~60% of all queries (*theta*=0.8).
3. **High Contention:** 16 queries per transaction (50% reads and 50% writes) with a hotspot of 10% tuples that are accessed by ~75% of all queries (*theta*=0.9).

For all of the YCSB experiments in this chapter, we used a ~10 GB database containing a single table with 10 million records. Each tuple has a single primary key column and then 10 additional columns each with 100 bytes of randomly generated string data.

7.5.2 TPC-C Results

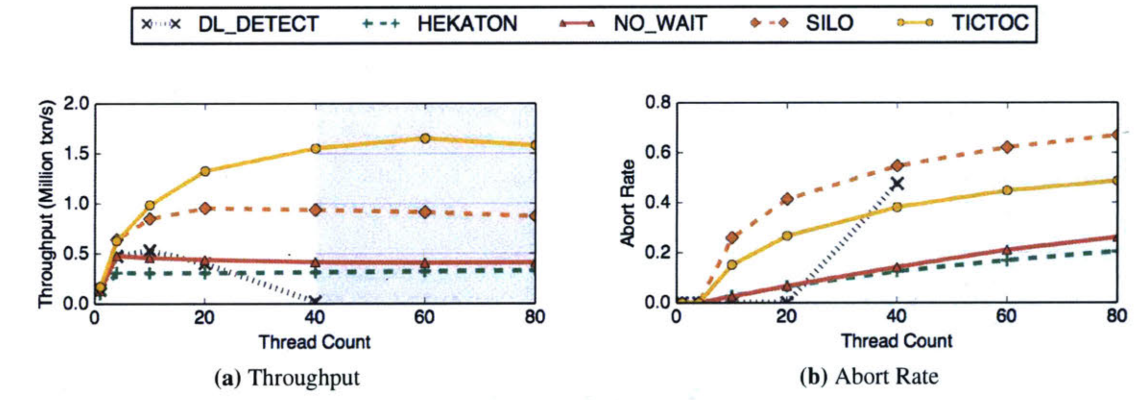
We first analyze the performance of all the concurrency control algorithms on the TPC-C benchmark.

The number of warehouses in TPC-C determines both the size of the database and the amount of concurrency. Each warehouse adds ~100 MB to the database. The warehouse is the root entity for almost all of the tables in the database. We follow the TPC-C specification where ~10% of the `NewOrder` transactions and ~15% of the `Payment` transactions access a "remote" warehouse.

We first run TPC-C with four warehouses, as this is an example of a database that has a lot of contention. We then run an experiment where we fix the number of threads and scale the number of warehouses in the database. This measures how well the algorithms scale when the workload has more parallelism opportunities.

4 Warehouses

The results in Figure 7-4 show that the performance improvements of additional threads are limited by contention on the `WAREHOUSE` table. Each `Payment` transaction updates a per-warehouse tuple in this table and each `NewOrder` transaction reads that tuple. Since there are only four such tuples in the entire database, they become the bottleneck of the whole system.



(a) Throughput (b) Abort Rate

Figure 7-4: **TPC-C (4 Warehouses)** – Scalability of different concurrency control algorithms on the TPC-C workload with 4 warehouses.

The `Payment` transaction is simpler and faster than `NewOrder` transactions. In SILO, when the `NewOrder` transaction enters the validation phase, it is likely that a `Payment` transaction has already modified the tuple in the `WAREHOUSE` table that is read by the `NewOrder` transaction. Therefore, SILO (like other traditional OCCs) frequently aborts these `NewOrder` transactions.

In TICTOC, the `NewOrder` transaction would also see that the `WAREHOUSE` tuple has been modified. But most of the time the transaction can find a logical commit timestamp that satisfies all the tuples it accesses and thus is able to commit. As shown in Figure 7-4, TICTOC achieves 1.8× better throughput than SILO while reducing its abort rate by 27%. We attribute this to TICTOC's ability to achieve better parallelism by dynamically selecting the commit timestamp.

The figure also shows that DL_DETECT has the worst scalability of all the algorithms. This is because DL_DETECT suffers from the thrashing problem discussed in Section 7.4.1. Thrashing occurs because a transaction waits to acquire new locks while holding other locks, which causes other transactions to block and form a convoy. NO_WAIT performs better than DL_DETECT as it avoids this thrashing problem by not waiting for locks. But NO_WAIT still performs worse than TICTOC and SILO due to the overhead of using locks. HEKATON also supports non-blocking reads since it is a multi-version concurrency control algorithm. A read to an earlier version of a tuple can be performed in parallel with a write to the new version of the same tuple. However, HEKATON is slower than TICTOC and SILO due to the overhead of maintaining multiple versions.

Variable Number of Warehouses

As we increase the number of warehouses while fixing the number of worker threads, the contention in the system will decrease. In Figure 7-5, the number of warehouses is swept from 4 to 80 and the number of worker threads is fixed to 80.

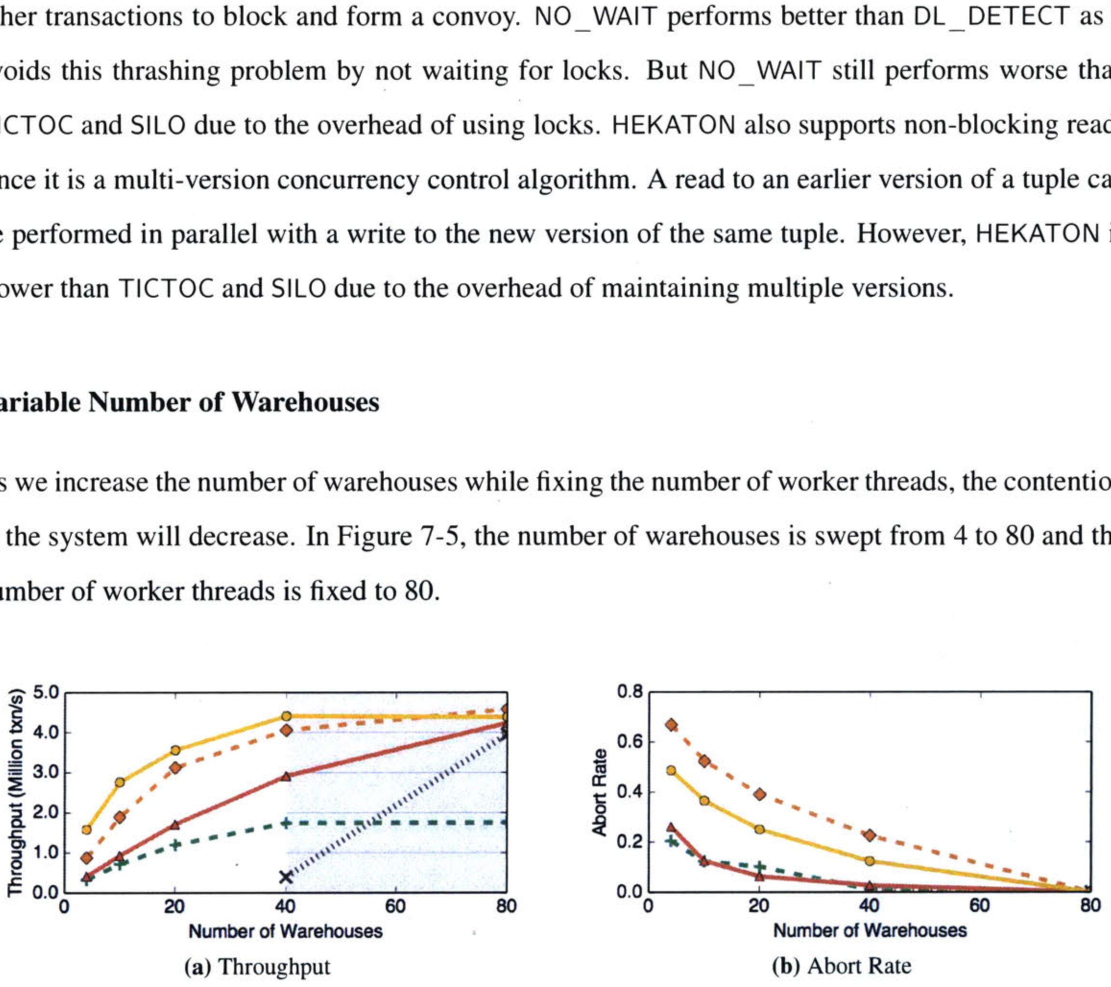


Figure 7-5: **TPC-C (Variable Warehouses)** – Scalability of different concurrency control algorithms on TPC-C as the number of warehouses change. The number of worker threads in DBx1000 is fixed at 80.

When the number of warehouses is small and contention is high, TICTOC performs consistently better than SILO for the same reason as in Section 7.5.2. As the number of warehouses grows, parallelism in TPC-C becomes plentiful, so the advantage of TICTOC over SILO decreases and eventually disappears at 80 warehouses, at which point transactions have little contention. With respect to the scheme's measured abort rate, shown in Figure 7-5b, TICTOC has consistently fewer aborts than SILO for fewer than 80 warehouses because it is able to dynamically adjust the logical commit timestamp for transactions to reduce the number of aborts.

7.5.3 YCSB Results

We now compare TicToc to other concurrency control schemes under the YCSB workload with different parameter settings.

Read-Only

We executed a YCSB workload comprising read-only transactions with a uniform access distribution. This provides a baseline for each concurrency control scheme before we explore more complex workload arrangements.

The results in Figure 7-6 show that all of the algorithms except for HEKATON scale almost linearly up to 40 threads. Beyond that point, scaling is sub-linear as the threads executing on the same physical core contend for the CPU. TICTOC and SILO achieve better absolute performance than the other algorithms because they do not have locking overheads. HEKATON is limited by its centralized timestamp allocation component. It uses a single *atomic add* instruction on a global counter, which causes threads accessing the counter from different cores to incur cache coherence traffic on the chip. In our 4-socket system, this limits HEKATON to ~5 million timestamps per second.

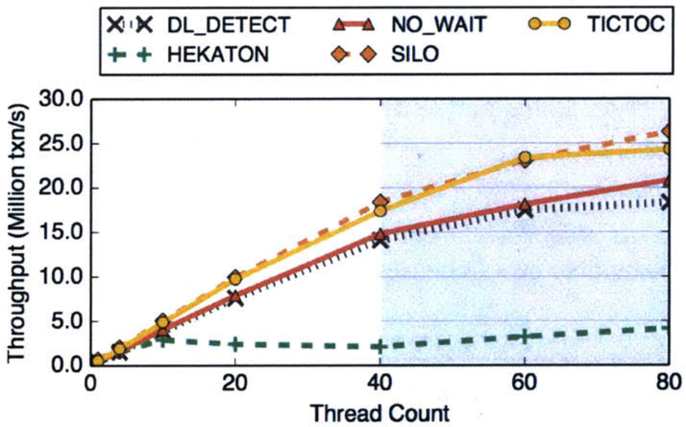


Figure 7-6: **YCSB (Read-Only)** – Results of a read-only YCSB workload for different concurrency control schemes.

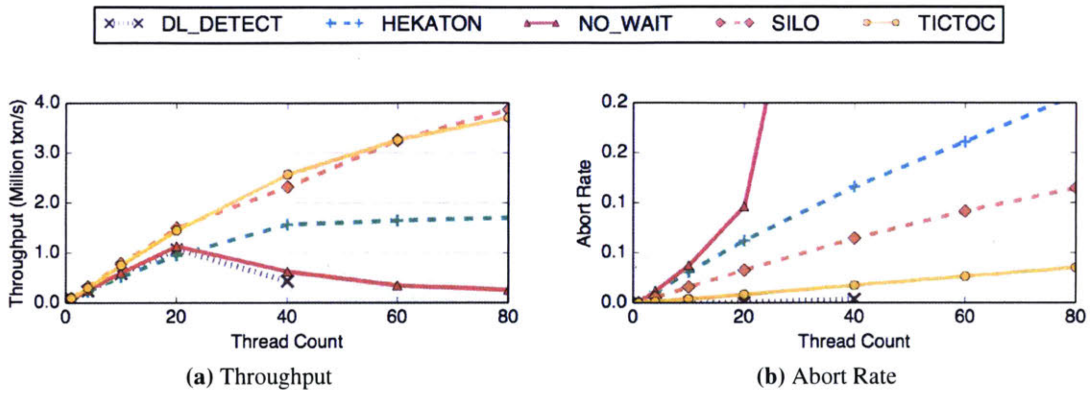


Figure 7-7: **YCSB (Medium Contention)** – Results for a read-write YCSB workload with medium contention. Note that DL_DETECT is only measured up to 40 threads.

Medium Contention

In a read-only workload, transactions do not conflict with each other and thus any algorithm without artificial bottlenecks should scale. For workloads with some contention, however, the ways that the algorithms handle conflicts affect the DBMS's performance.

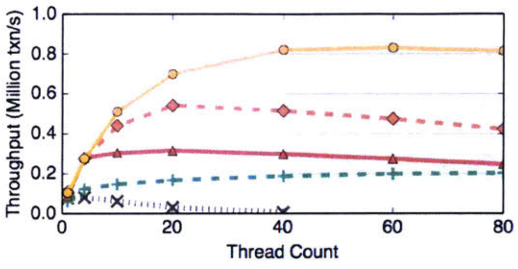
Figure 7-7 shows the throughput and abort rate of the medium-contention YCSB workload. The results in Figure 7-7a show that SILO and TICTOC both scale well and achieve similar throughput. But the graph in Figure 7-7b shows that TICTOC has a ~3.3× lower abort rate than SILO. This is due to TICTOC's data-driven timestamp management, as transactions can commit at the proper timestamp that is not necessarily the largest timestamp so far.

The throughput measurements show that DL_DETECT again has the worst scalability of all the algorithms due to lock trashing. NO_WAIT does better since transactions can get immediately restarted when there is a conflict. HEKATON performs better than the 2PL schemes since multiple versions allows more read operations to succeed (since they can access older versions) which leads to fewer transaction aborts. But this adds overhead that causes HEKATON to perform worse than TICTOC and SILO.

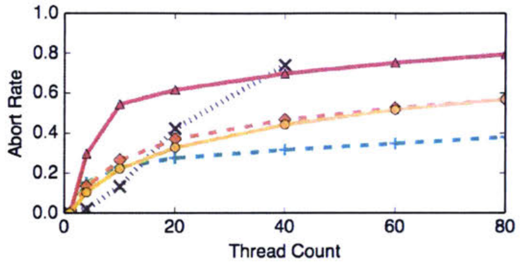
High Contention

We now compare the algorithms on a YCSB workload with high contention. Here, conflicts are more frequent and the workload has lower inherent parallelism, which stresses the DBMS and allows us to more easily identify the main bottlenecks in each algorithm.

As expected, the results in Figure 7-8 show that all algorithms are less scalable than in the medium-contention workload. We note, however, that the performance difference between TICTOC



(a) Throughput



(b) Abort Rate

Figure 7-8: **YCSB (High Contention)** – Results for a read-write YCSB workload with high contention. Note that DL_DETECT is only measured up to 40 threads.

and SILO is more prominent. As we discuss next, this performance gain comes partially from the optimizations we presented in Section 7.4.

Both TICTOC and SILO have similar abort rates in Figure 7-8b under high contention. The timestamp management policy in TICTOC does not reduce the abort rate because the workload is too write-intensive and the contention level is so high that both algorithms have similar behaviors in terms of aborting transactions.

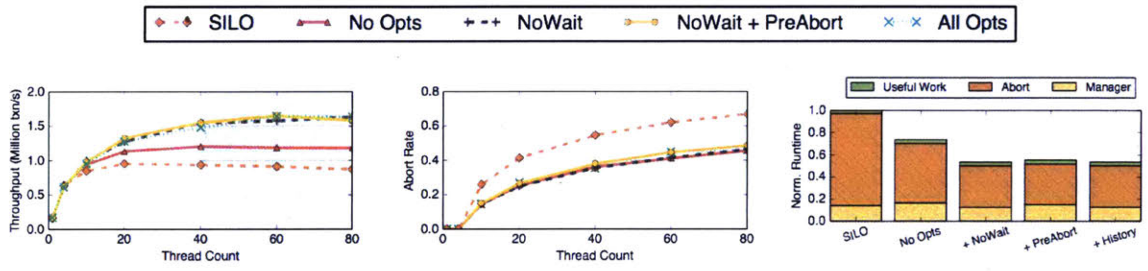
7.5.4 TicToc Optimizations

We now evaluate the optimizations we presented in Section 7.4 to determine their individual effect on TicToc's overall performance. To do this, we run DBx1000 multiple times using TicToc but enable the optimizations one-at-a-time. We use four different configurations in this experiment:

1. **No Opts:** TicToc without any optimizations.
2. **NoWait:** TicToc with No-Wait locking described in Section 7.4.1.
3. **NoWait + PreAbort:** TicToc with No-Wait locking and preemptive aborts described in Section 7.4.2.
4. **All Opts:** TicToc with No-Wait locking, preemptive aborts, and timestamp history described in Section 7.4.3.

Recall that this last configuration is the default setting for TicToc in all of the other experiments in this thesis. We also include the performance measurements for SILO from Figure 7-8a for comparison.

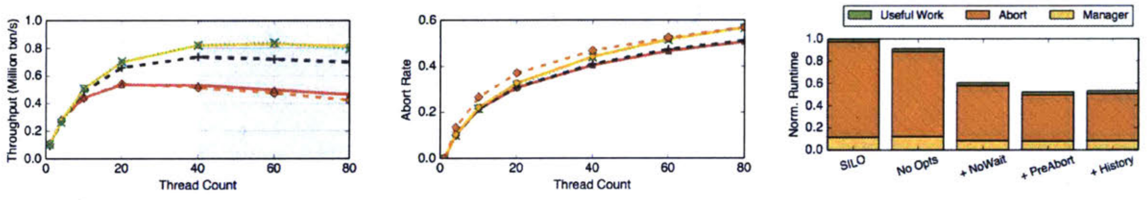
The results in Figure 7-9 show the performance, abort rate, and time breakdown (at 80 cores) for the TPC-C workload with four warehouses. At 80 threads, TICTOC without optimizations achieves 35.6% higher throughput than SILO, and has a 32% lower abort rate. This gain comes from the greater parallelism exploited by the TICTOC's timestamp management policy. Using the No-Wait optimization for locking transactions' write sets provides another 38% performance gain at 80 threads, while not affecting the abort rate. In Figure 7-9c, we see that the gains of basic TICTOC over SILO mainly come from reducing the abort rate. Optimizations do not reduce TICTOC's abort rate further, but they do reduce the amount of time wasted in aborting transactions. These optimizations effectively make each abort take a shorter amount of time.



(a) Throughput **(b)** Abort Rate **(c)** Execution Time Breakdown (80 threads)

Figure 7-9: **TicToc Optimizations (TPC-C)** – Throughput measurements of TicToc using different optimizations from Section 7.4. The system runs with the TPCC workload with 4 warehouses.

Figure 7-10 shows the same experiments on the high contention YCSB workload. Here, we see similar performance improvement as in TPC-C but it comes from different aspects of TICTOC. TICTOC without any optimizations only performs 10% better than SILO, and most of the performance improvement comes from the No-Wait and preemptive abort optimizations. In contrast to Figure 7-9, using preemptive aborts provides a larger performance gain in YCSB. This is partly because in YCSB each transaction locks more tuples during the validation phase. Preemptive aborts alleviate the contention caused by these locks.



(a) Throughput **(b)** Abort Rate **(c)** Execution Time Breakdown (80 threads)

Figure 7-10: **TicToc Optimizations (YCSB)** – Throughput measurements of TicToc using different optimizations from Section 7.4. The system runs with the high-contention YCSB workload.

We observe that the timestamp history optimization does not provide any measurable performance gain in either workload. This was initially surprising to us, but upon further investigation we are convinced that this is indeed correct. In a way, TICTOC without this optimization already stores multiple versions of a tuple in each transaction's private workspace. This means that each transaction can commit in parallel using its own version. Although in theory timestamp history can enable more concurrency, in practice there is no clear performance benefit for the workloads we evaluated.

7.5.5 Logical Time Analysis

We analyze how TicToc's commit timestamps grow over time under different levels of contention. If timestamps grow slower relative to the total number of committed transactions, then the amount of synchronization among transactions is less infrequent, which means the contention level is low. For this experiment, we execute the medium and high contention YCSB workloads from Section 7.5.3 and track the commit timestamps of transactions over time. We also include TS_ALLOC as a baseline where each transaction is assigned a unique timestamp using an atomic add instruction. This is representative of the timestamp growth rate in other T/O-based algorithms, such as HEKATON.

Figure 7-11 shows the relationship between logical timestamps and the number of committed transactions for the three configurations. With the TS_ALLOC protocol, the number of committed transactions and logical timestamps increase at the same rate. In TicToc, however, logical timestamps increase at a slower rate: $64\times$ and $10\times$ slower than the number of committed transactions, respectively, for low and high contention levels in YCSB. What is interesting about these measurements is that the rate of growth of logical timestamps indicates the inherent level of parallelism in a workload that can be exploited by TicToc. In the high contention workload, for example, this ratio is $10\times$. This corroborates our results in Figure 7-8a, which show the DBMS is only able to achieve $7.7\times$ (which is smaller than $10\times$) better throughput when executed on multiple threads compared to a single thread, in the high contention YCSB workload.

7.5.6 Isolation Levels

All of the experiments so far have used serializable isolation. Serializable isolation is the strictest isolation level and thus usually has less concurrency than lower isolation levels. We now compare the DBMS's performance when transactions execute under snapshot isolation (SI) and repeatable read isolation (RR) levels versus the default serializable isolation (SR). All five algorithms support the RR level. For SI, we are able to run it only with the TICTOC and HEKATON algorithms. This is because supporting SI requires the DBMS to maintain multiple versions for each tuple, which is difficult for the other algorithms. We use the medium- and high-contention YCSB workloads from

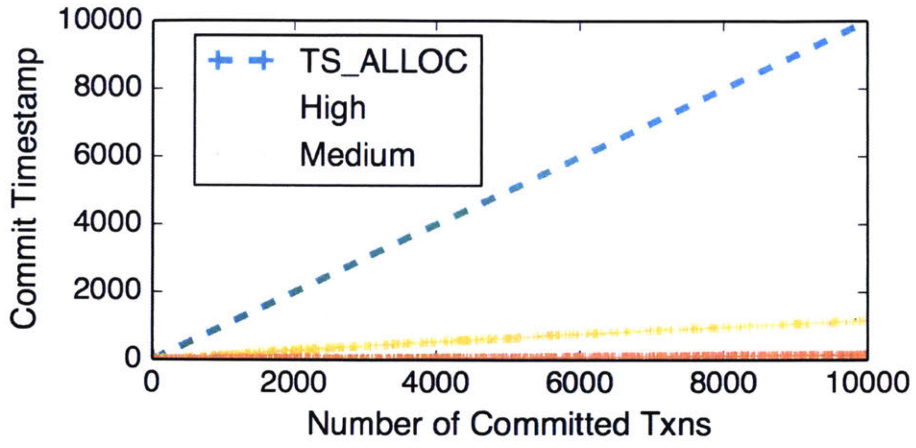


Figure 7-11: **Logical Time Analysis** – Comparison of the growth rate of the timestamps in TICTOC versus TS_ALLOC.

Section 7.5.3 for our experiments.

The medium-contention YCSB results are shown in Table 7.1. For this setting, the workload has enough parallelism and thus all the optimistic T/O-based algorithms only see small improvements when running at a lower isolation level (4.7% for TICTOC and 5.6% for SILO), whereas for the pessimistic 2PL algorithms the improvement is more pronounced (67.4% for DL_DETECT and 200.0% for NO_WAIT). HEKATON only has a 21.3% improvement from SR to RR. The abort rate measurements in Table 7.1b show that the lower isolation levels achieve lower abort rates because there are fewer conflicts between transactions. As expected, all the algorithms have the fewest number of aborted transactions under RR since it is the most relaxed isolation level.

	DL_DETECT	HEKATON	NO_WAIT	SILO	TICTOC
SR	0.43	1.55	0.63	2.32	2.57
SI	–	1.78	–	–	2.69
RR	0.72	1.88	1.89	2.45	2.69

(a) Throughput (Million txn/s)

	DL_DETECT	HEKATON	NO_WAIT	SILO	TICTOC
SR	0.35%	11.6%	63.2%	6.47%	1.76%
SI	–	1.96%	–	–	1.54%
RR	0.10%	1.94%	9.9%	0.71%	0.72%

(b) Abort Rate

Table 7.1: **Isolation Levels (Medium Contention)** – Performance measurements for the concurrency control schemes running YCSB under different isolation levels with 40 threads.

The high-contention YCSB results are shown in Table 7.2. Lower isolation levels have better performance than serializable isolation. Again, the throughput of the RR isolation level is slightly better than SI's. In general, for this workload setting we found that different isolation levels do not cause large reductions in abort rates due to the significant amount of contention on hotspot tuples.

	DL_DETECT	HEKATON	NO_WAIT	SILO	TICTOC
SR	0.005	0.18	0.30	0.52	0.82
SI	–	0.23	–	–	0.90
RR	0.010	0.23	0.35	0.80	1.04

(a) Throughput (Million txn/s)

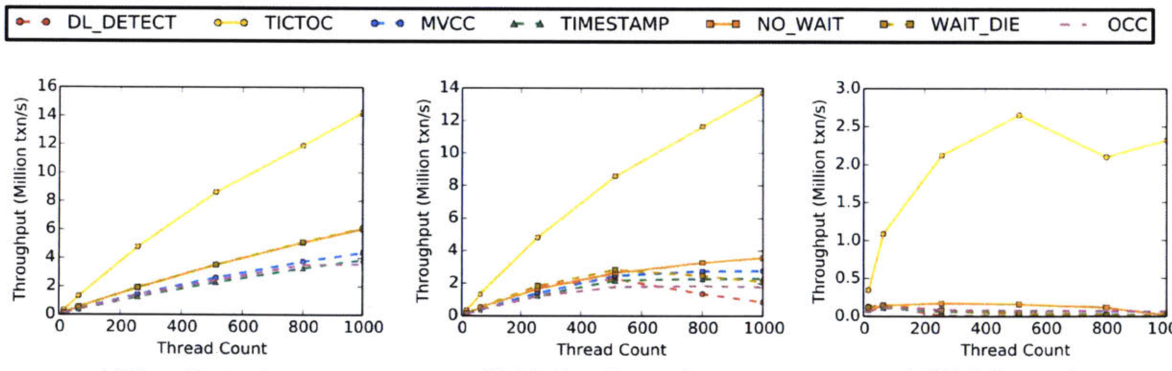
	DL_DETECT	HEKATON	NO_WAIT	SILO	TICTOC
SR	74.0%	34.4%	69.9%	46.8%	44.3%
SI	–	30.9%	–	–	40.1%
RR	74.3%	30.4%	71.3%	42.3%	39.7%

(b) Abort Rate

Table 7.2: **Isolation Levels (High Contention)** – Performance measurements for the concurrency control schemes running YCSB under different isolation levels with 40 threads.

7.5.7 TicToc on 1000 Cores

In this section, we study the scalability of TicToc at a larger scale, by running TicToc on the same simulated 1000-core processor as we used in Chapter 6. In Figure 7-12, we present the throughput (in million transactions per second) of TicToc and six other traditional concurrency control algorithms at different levels of contention (low, medium, and high contention). The database is unpartitioned in this experiment, and thus HSTORE is not included.



(a) Low Contention (b) Medium Contention (c) High Contention

Figure 7-12: **TicToc on 1000 Cores (YCSB)** – Throughput measurements of TicToc compared to classic concurrency control algorithms that are evaluated in Chapter 6. The system runs with the different contention level on the YCSB workload.

We observe that TicToc significantly outperforms other concurrency control algorithms, especially when the contention level is high. In the high contention case (Figure 7-12c), TicToc outperforms other algorithms by an order of magnitude. This is because TicToc allows more transactions to commit in parallel by dynamically determining their commit order. The performance of TicToc plateaus after 512 cores due to the limited parallelism in the workload.

Chapter 8

Sundial Distributed Concurrency Control

8.1 Introduction

In Chapter 6, we saw how concurrency control can be a serious scalability bottleneck in a OTLP DBMS when deployed on a shared memory multicore machine. We also saw how logical leases can completely remove the scalability bottleneck and improve performance of an OLTP DBMS in Chapter 7. In this chapter, we extend the idea of logical leases beyond a single multicore machine. We show how logical leases can improve the performance of a distributed database management system (DDBMS).

When the computational and storage demands of an OLTP application exceed the resources of a single-node system, an organization must turn to a distributed DBMS. Such systems split the database into disjoint subsets, called partitions, that are separately stored across multiple shared-nothing nodes. If transactions only need to access data at a single node, these systems achieve great scalability [3, 4]. The trouble, however, arises when transactions have to access multiple nodes. These *distributed transactions* incur two problems. First, distributed concurrency control over the network imposes a performance and scalability bottleneck. Second, reading data from a remote node incurs significant latency and network traffic, and caching remote data locally requires complex protocols to handle updates.

Recent work on overcoming these problems in distributed DBMSs can be grouped into three categories. First, protocol-level improvements to reduce synchronization among transactions [103, 142, 49]. However, these schemes still limit the concurrency of the DBMS or incur extra overhead. We will make qualitative and quantitative comparisons to some of these schemes. Second, hardware improvements, including using atomic clocks to order transactions [36] or optimized networks that enable low-latency remote memory access [44, 150]. However, the hardware is not widely available or is cost prohibitive for some organizations. Third, there have been proposals to restrict the programming model for transactions [115, 142]. Such limitations make it difficult to port existing applications to these systems. Ideally, we want concurrency control protocols that achieve good performance on commodity hardware using an unrestricted programming interface.

To hide the long network latency, previous work has proposed to replicate hotspot shared data across multiple database nodes [37], such that some distributed transactions become single-node transactions. However, this requires profiling the database workload or manual identification of hot data, which adds extra complexity for the users. It would be easier if the database can dynamically replicate data across nodes during normal operation through the caching mechanism to reduce distributed transactions [7], but the complexity of maintaining coherence across caches is prohibitive [123], leading to limited adoption of the idea.

To address these issues, we introduce **Sundial**, a new distributed concurrency control protocol that outperforms existing approaches through two main contributions. First, Sundial uses a new distributed concurrency control algorithm that eliminates the need for transactions to wait for locks and reduces false aborts from read-write conflicts. Second, Sundial natively provides strongly consistent data caching without requiring an additional protocol or external system, and uses cached data only when beneficial.

The fundamental principle behind both components in Sundial is *logical leases*. Similar to TicToc, logical leases allow the DBMS to dynamically determine the logical order among transactions while enforcing serializability guarantees. A transaction dynamically computes its commit timestamp to overlap with leases of all the accessed tuples. Different from TicToc, Sundial uses logical leases not only for concurrency control, but also for caching data that map to a remote server in the local server's main memory. In a sense, Sundial combines ideas in both TicToc and Tardis and applies them to a distributed DBMS.

To evaluate Sundial, we implemented it in a distributed DBMS testbed and compared it to three state-of-the-art protocols: MaaT [103], Google F1 [130], and two-phase locking [69]. We used two OLTP workloads with different contention settings and cluster configurations. Sundial achieves up to 57% better throughput with 41% lower latency than the alternative approaches. We also show that Sundial's caching feature improves throughput by up to 3× for certain workloads with minimal extra complexity or overhead.

In summary, this chapter makes the following contributions.

- We apply logical leases to a distributed system and build Sundial. The new distributed concurrency control algorithm in Sundial avoids lock waiting and significantly reduces aborts due to read-write conflicts among transactions.
- We implement a lightweight caching protocol, which allows Sundial to cache data from remote nodes. The caching mechanism is tightly integrated into the concurrency control algorithm with very low complexity.
- Evaluated over YCSB and TPCC benchmarks, Sundial significantly improves performance over other baselines. Overall, Sundial's new distributed concurrency control delivers 57% performance improvement. The caching mechanism can deliver up to 3× performance improvement for workloads with hot read-intensive tables.

The rest of the chapter is organized as follows. Section 8.2 presents the design of the Sundial concurrency control scheme. Section 8.3 introduces Sundial's distributed caching technique. Section 8.4 discusses possible causes of aborts in Sundial and qualitatively compares Sundial with other concurrency control algorithms. Finally, Section 8.5 presents quantitative evaluation.

8.2 Sundial Concurrency Control

In this section, we discuss the Sundial protocol in detail. In a high-level overview, Sundial uses logical leases (Section 8.2.1) to seamlessly integrate caching with the concurrency control algorithm to improve performance of distributed transactions. Sundial uses a hybrid single-version concurrency control algorithm that handles write-write conflicts using 2PL and read-write conflicts using OCC (Section 8.2.2). The detailed concurrency control protocol will be presented in Section 8.2.3. Section 8.2.4 and 8.2.5 discuss optimizations and extensions.

Our discussion assumes that the database is deployed over a homogeneous cluster of server nodes. Each node is able to initiate a transaction on behalf of an external user, as well as to process remote queries on behalf of peer nodes. The node initiating the transaction is called the *coordinator* of that transaction. A distributed transaction accessing a remote node will execute a sub-transaction on that node.

8.2.1 Logical Leases in Sundial

Logical leases allow the DBMS to determine a *partial logical order* among concurrent operations on shared data. The management of logical leases in Sundial is similar to that in TicToc (cf. Chapter 7). A logical lease is associated with each data element (e.g., each tuple), and is represented using *wts* and *rts*. A data element can be read only at a logical time within its current lease; a lease can be extended in order to read the tuple at a later logical time. A write to a tuple creates a new lease after the end of the current lease.

To achieve serializability, the DBMS must compute a single *commit timestamp* for a transaction, a logical time that is within the leases of all tuples it accesses. From the perspective of logical time, all operations of a transaction are *atomically* performed at the commit timestamp. The DBMS can calculate this commit timestamp using only the leases of tuples accessed by the transaction. No centralized or inter-transaction coordination is required.

Figure 8-1 shows an example of two transactions running under Sundial. *T1* reads `A` and `B` with leases of `[0, 1]` and `[1, 2]`, respectively. *T1* also writes to `D` with a new lease of `[1, 1]`, which is greater than the current lease of `D` at `[0, 0]`. *T1* commits at timestamp `1` since it overlaps with all the leases it has seen. Note that before *T1* commits, *T2* has already modified `A`. But *T1* does not abort due to this conflict since Sundial computes the global serialization order using logical time. The conflict between *T1* and *T2* enforces *T1* to commit before *T2* in the timestamp order, but neither of the transactions has to abort.

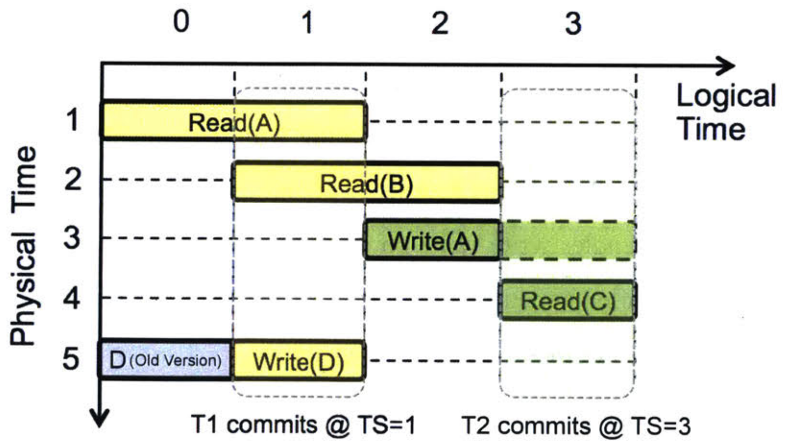


Figure 8-1: **Logical Lease Example** – Example schedule of two transactions (*T1* and *T2*) accessing `A`, `B`, `C`, and `D`. *T1* and *T2*'s operations are highlighted in yellow and green, respectively.

T2 creates a new lease of [2, 2] and writes `A`. It also reads `C` that has a lease of `[3, 3]`. The two leases, however, do not overlap. In this case, Sundial extends the lease on A from 2 to 3. The DBMS is allowed to perform such an extension because there is no valid version of A at timestamp 3. In contrast, reversing C's lease from 3 to 2 can also make the two leases overlap, but this is not allowed since a previous version of C is already valid at timestamp 2. After the lease extension, the DBMS will commit *T2* at timestamp 3 since it overlaps with leases of both A and C after the lease extension.

8.2.2 Conflicts

Sundial handles different types of conflicts among transactions using different methods [18, 138]. Conflicts are either *write-write* (i.e., two transactions writing the same tuple) or *read-write* (i.e., one transaction reads and the other writes the same tuple).

For write-write conflicts, we assume each write is actually a read-modify-write update (blind writes will be discussed later in Section 8.2.5). There is no parallelism opportunity as the two writes must happen sequentially. For an OCC algorithm, among the transactions that have write-write conflicts with each other, at most one of them is able to commit while the others have to abort. This leads to performance degradation and waste of system resources. Sundial uses the more conservative 2PL concurrency control for write-write conflicts to avoid excessive aborts that can occur with OCC [158].

For read-write conflicts, for ease of discussion, we ignore the order between the read and write operations in the two transactions. In this case, OCC is able to exploit more parallelism than 2PL by avoiding waiting for locks. For example, the DBMS can execute a transaction that updates a tuple in parallel with another transaction that reads an old value of that same tuple. As long as the reading transaction is logically ordered before the writing transaction, the DBMS can commit both of them simultaneously without requiring either one to wait for a lock. Existing OCC algorithms, however, unnecessarily abort transactions with read-write conflicts due to their limitation in the validation process [158]. Consider the example shown in Figure 8-2 where transaction *T1* first reads tuple A and then *T2* modifies that same tuple. The DBMS successfully commits *T2* before *T1* starts to validate. When *T1* finishes execution, the DBMS observes that A was already modified since its last read and therefore it will have to abort *T1*.

In this example, *T1*'s abort may be unnecessary. If a serializable order exists among all transactions such that *T1* is logically ordered before *T2*, then both transactions can commit. It is non-trivial, however, for the DBMS to discover whether such a logical order exists with low algorithmic complexity at runtime. Existing techniques typically have extra requirements for the DBMS, such as

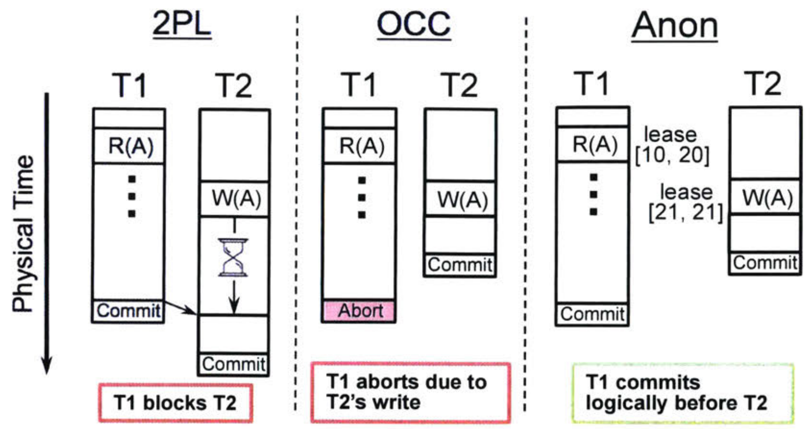


Figure 8-2: **Read-Write Conflict Example** – Example schedule of two transactions with a read-write conflict in 2PL, OCC, and Sundial.

multi-versioning [50, 148] and deterministic execution [142]. As we now discuss, Sundial does not have these restrictions and works for any single-version database with low complexity.

8.2.3 Sundial Concurrency Control Protocol

As shown in Figure 8-3, the lifecycle of a distributed transaction in Sundial consists of three phases: (1) **execution**, (2) **prepare**, and (3) **commit**. The application initiates a new transaction in the coordinator. In the first phase (execution), the coordinator executes the transaction's program logic and sends query requests to other nodes involved in the transaction's logic. When the transaction completes its operations, it reaches the commit point and enters the prepare phase. This is where the DBMS begins the *two-phase commit* (2PC) process, which determines whether the transaction is able to commit, and if so, commits the transaction atomically. The 2PC contains two phases, the *prepare phase* and the *commit phase*. In the prepare phase, the coordinator sends a commit request to each participating remote node involved in the transaction. Each remote node can respond to this request with either "OK" or "Abort". If all nodes agree that the transaction can commit, then the transaction enters the commit phase, where the coordinator sends a message to all the remote nodes to complete the transaction. Otherwise, the transaction is aborted and all of its modifications are rolled back. To ensure that all of a transaction's modifications persist after a node failure, the DBMS flushes log records at each node at different points during the 2PC protocol (shown as $*$ in Figure 8-3).

During the three phases (i.e., execution, prepare and commit), Sundial performs additional coordination between the coordinator and remote nodes to calculate a transaction's commit timestamp

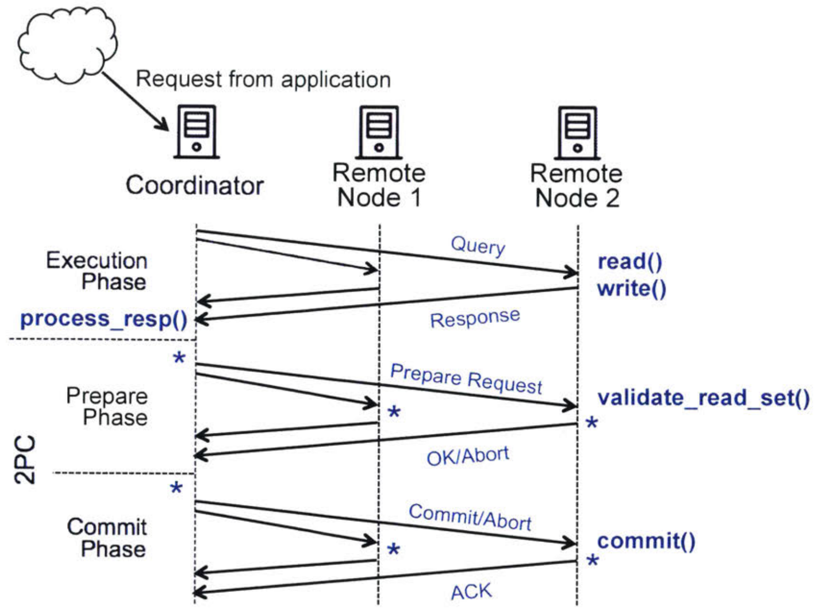


Figure 8-3: **Distributed Transaction Lifecycle** – A distributed transaction goes through the execution phase and two-phase commit (2PC) which contains the prepare and commit phases.

and to extend leases. Sundial does not need to send additional messages to do this and instead piggybacks this coordination information on normal request and response messages.

We now describe how Sundial executes the three phases in more detail.

Execution Phase

During the execution phase, a transaction reads and writes tuples in the database and performs computation. Each tuple contains additional metadata that the DBMS uses to track the logical leases of transactions. A tuple's *write timestamp* (*wts*) and *read timestamp* (*rts*) represent the start and end of a logical lease. Sundial uses the 2PL Wait-Die algorithm [18] to handle write-write conflicts, which requires the DBMS to also maintain the current *lock owner* and a *waitlist* of transactions waiting for the lock. Each tuple has the following format, where DB is the database and the subscript n denotes the identifier of the node containing the tuple.

$$DB_n[key] = \{wts, rts, owner, waitlist, data\}$$

Since Sundial handles read-write conflicts using OCC, it maintains the *read set* (RS) and the *write set* (WS) for each transaction at each participating node, like how other OCC algorithms do. The two sets are modeled as follows. The `data` field contains a local copy of the database tuple that can be read (in RS) or written (in WS). An entry in the read set also contains the *wts* and *rts* of the tuple read by the transaction. The subscript n is included because a transaction's RS or WS can be spread across multiple servers; and each server maintains a subset of the RS or WS for the transaction.

$$RS_n[key] = \{wts, rts, data\}$$

$$WS_n[key] = \{data\}$$

At runtime, the DBMS either processes the transaction's queries locally at the coordinator or sends the queries to the node that contains the target tuples. In either case, the logic to process a read or a write query is the same. Algorithm 9 shows the operations performed by Sundial on node *n* when a read or write query is executed.

Algorithm 9: Execution Phase

```
1  Function read(key) @ node n
2      if key not in $RS_n$ then
3          # Atomically copy lease and data
4          $RS_n$[key].{wts, rts, data} = $DB_n$[key].{wts, rts, data}
5          $commit\_ts_n$ = Max($commit\_ts_n$, wts)
6      end
7      return $RS_n$[key].data
8  Function write(key, new_data) @ node n
9      if key not in $WS_n$ then
10         if $DB_n$[key].lock() == Success then
11             $WS_n$[key].data = $DB_n$[key].data
12             $commit\_ts_n$ = Max($commit\_ts_n$, $DB_n$[key].rts + 1)
13         else
14             Abort()
15         end
16     end
17     # Updates $WS_n$[key].data using new_data
18     $WS_n$[key].update(new_data)
19 # handle response from remote node n
20 Function handle_resp($commit\_ts_n$) @ coordinator
21     commit_ts = Max(commit_ts, $commit\_ts_n$)
```

For a read, if the tuple is already in the transaction's read set, then the DBMS just returns it without updating any metadata (line 2). Otherwise, the DBMS atomically copies the tuple's *wts*, *rts*, and *D* into the transaction's local read set (line 4). It also updates the transaction's $commit\_ts_n$ at node *n* to be at least the *wts* of the tuple (line 5). $commit\_ts_n$ cannot be smaller than *wts* of the tuple since the tuple is invalid earlier than that logical time. Then, the data in the read set is returned to the transaction (line 7). If the transaction reads that tuple again, it will just access that version in its read set without having to retrieve it from the database again.

For a write, if the tuple is already in the transaction's write set, then the DBMS just applies the update to this cached copy and returns immediately (line 9). Otherwise, the DBMS will attempt to acquire the lock on that tuple for the transaction (line 10). If this lock acquisition is successful (potentially after waiting for some period of time), then the DBMS copies the tuple into the transaction's write set (line 11) and advances the transaction's $commit\_ts_n$ to be at least $rts + 1$ (line 12). Note at this point the updated tuple is in the transaction's private write set; its changes are not visible to other active transactions. If the lock acquisition fails (i.e., because a transaction with a higher priority owns it), then the DBMS immediately aborts the transaction (line 14). After the data is in the write set, the update operation is performed (line 18) and the transaction execution continues.

Once the DBMS completes the operation for the transaction at node n, it sends the results as well as the current $commit\_ts_n$ back to the coordinator. The coordinator executes *handle_resp* after receiving this response. The coordinator only keeps the largest *commit_ts* it has seen so far. By the end of the execution phase, the *commit_ts* of the transaction at the coordinator is the minimum timestamp at which the transaction can possibly commit. This final *commit_ts* will be used in the 2PC protocol.

An important feature of Sundial is that a read operation from one transaction does not block any write operation from another transaction during the execution phase (and vice versa). The tuple lock held by one transaction only prevents another transaction from modifying that tuple's lease; it does not stop another transaction from reading the tuple's lease metadata and the data itself, as long as the data and metadata are consistently read from the tuple. If a transaction reads a locked tuple, then the transaction will have a commit timestamp smaller than that of the transaction holding the lock, but both transactions may be able to commit.

Prepare Phase

The goal of the prepare phase is for the DBMS to discover whether it is safe to commit the transaction at *every* node that it accessed. The coordinator sends the transaction's *commit_ts* to each participating node in the prepare messages. Each participating node (including the coordinator) then executes the *validate_read_set(commit_ts)* function shown in Algorithm 10.

For a write operation, since the transaction already holds the lock on the tuple in the execution phase, the lease is unchanged since the last access from the transaction. Therefore, the lease of a tuple in the write set must be valid at the transaction's *commit_ts*. For a tuple in the read set, however, the *commit_ts* may be greater than the end of the lease of the tuple, making it unclear whether the tuple is still valid at the *commit_ts* of the transaction. Thus, Sundial will validate the

Algorithm 10: Prepare Phase

```
1  Function validate_read_set(commit_ts) @ server n
2      for key in $RS_n$.keys() do
3          if commit_ts > $RS_n$[key].rts then
4              # Begin atomic section
5              if $RS_n$[key].wts != $DB_n$[key].wts or $DB_n$[key].is_locked() then
6                  Abort()
7              else
8                  $DB_n$[key].rts = Max($DB_n$[key].rts, commit_ts)
9              end
10             # End atomic section
11         end
12     end
13     return OK
```

transaction's read set of each participating node and extend the leases when necessary.

For each tuple in the transaction's read set (line 2), if *commit_ts* is already within the lease of the tuple, then no updates are needed (line 3). Otherwise, Sundial tries to extend the tuple's lease. If the current *wts* of the tuple is different from the *wts* observed by the transaction during the execution phase, or if the tuple is locked, then the DBMS cannot extend the lease and the transaction aborts (lines 5-6). Otherwise, the *rts* of the tuple is updated to be at least *commit_ts* (line 8). Note that lease extension is the only way to change a tuple's *rts* in Sundial.

If a remote node successfully validates the transaction, then the DBMS sends an *OK* response to the coordinator. If the validation succeeds at all of the participating nodes (including the coordinator), the transaction enters the commit phase; otherwise the DBMS aborts the transaction.

Commit Phase

When the transaction enters this phase, the coordinator sends the commit messages to all of the participating remote nodes. Algorithm 11 shows the logic executed at each remote node. For each modified tuple in the transaction's local write set at each node (line 2), the DBMS copies the version from the write set to the database (line 3). The *wts* and *rts* of the tuple are both set to the *commit_ts* of the transaction (line 4-5) and the tuple is unlocked (line 6).

No special operation is required for the read set during the commit phase. This means that if a transaction did not modify any tuples at a remote node, then the coordinator can skip it during the commit phase. If the transaction was entirely read-only on all remote nodes, then the DBMS can completely skip the commit phase.

Finally, for either committed or aborted transactions, each node drops the transaction's local

Algorithm 11: Commit Phase

```
Function commit(commit_ts) @ server n
    for key in $WS_n$.keys() do
        $DB_n$[key].data = $WS_n$[key].data
        $DB_n$[key].wts = commit_ts
        $DB_n$[key].rts = commit_ts
        $DB_n$[key].unlock()
    end
```

read and write sets.

8.2.4 Indexes

Indexes require special treatment in terms of concurrency control. This is because the index's physical structure is allowed to change during a transaction's lifetime as long as its logical contents are consistent. One challenge of enforcing consistency in indexes is the *phantom read* problem. A phantom read occurs if a transaction reads a set of tuples from the index but then that set changes because another transaction modifies the index. For serializability, the DBMS must guarantee that an index returns the same results even if a transaction accesses it multiple times. It must also guarantee that modifications to the index follow the same logical order of transactions.

Conceptually, we can simplify the interface to an index by considering a lookup or a scan as a read operation to the index, and an insert or a delete as a write operation to the index. With this analogy, the basic principle of a concurrency control algorithm can be applied to indexes. In 2PL, for example, the affected keys in the index are locked in the corresponding modes. For a scan, the next key after the scanned set is also locked to prevent inserts into the gap between the next key and the scanned set [113]. This guarantees that a lookup or a scan cannot occur simultaneously with an insert or delete to the same set of tuples, eliminating the phantom anomaly. In OCC, the phantom anomaly can be avoided by having the transaction rescanning the index structure in the validation phase to check that the scanned set has not changed. Rescanning an index is similar to rereading a tuple to verify that it has not been changed.

In Sundial, we apply concurrency control to indexes by considering an index node (e.g., a leaf node in a B-tree index or a bucket in a hash index) as a tuple. Therefore, a lookup or a scan copies the lease of the index node; and an insert or a delete locks the index node. But lookup/scan and insert/delete do not block each other, in the same way that reads and writes do not block each other in the basic Sundial protocol.

With a range scan, for example, a transaction maintains the *wts* and *rts* of all the index nodes it scans and may extend *rts* for some nodes during validation. Another transaction inserting/deleting tuples in that range will be performed at a logical time after the *rts* of the affected index node. If the scanning transaction finds a *commit_ts* less than or equal to this *rts*, it can commit logically before the inserting/deleting transaction; and both transactions may commit. Otherwise, the scanning transaction will abort due to failed lease extension. To guarantee repeated scans return consistent results, the current implementation of Sundial makes a local copy of each scanned index node. An alternative, potentially more memory-efficient design is to only store the *wts* of the scanned index nodes (instead of storing the whole content of the node). If a transaction rescans a same index node at a later time, it compares the local *wts* and the current *wts* of the index node to make sure they are the same.

8.2.5 Blind Writes

The Sundial algorithm discussed so far supports reads and updates. In practice, a query may completely overwrite a tuple without reading it first. Such a write is called a *blind write*. Compared to read-modify-write updates, blind writes provide more parallelism opportunities for the DBMS since multiple transactions can write to the same tuple in parallel; only the last write remains while previous writes are ignored.

Sundial supports parallel blind writes with small extensions to the protocol presented so far. During the execution phase, a blind write locks the tuple. Different from a read-modify-write operation, however, the locks for blind writes are compatible with each other. Namely, multiple transactions blindly writing to the same tuple can share the same lock. During the commit phase, the DBMS checks whether a transaction's *commit_ts* is greater than the *wts* of the tuple that it blindly writes to. If so, the DBMS updates the data and the tuple's *wts* to the transaction's *commit_ts*. Note that the DBMS did not update the tuple's *rts* since the transaction never reads that tuple. If the transaction's *commit_ts* is less than *wts*, however, the tuple has already been updated by another blind write at a later logical time. Therefore, the current write is simply ignored; this is similar to the Thomas Write Rule in T/O protocols [141]. According to the protocol described above, the transaction must have $commit\_ts > rts$ in this case, which means ignoring the write does not affect any other transaction's reads. Finally, to handle the case where *commit_ts* equals *wts*, the DBMS can use any static metric to break a tie. For example, the writing transaction's ID can be stored together with each *wts* in a tuple so that the (*wts*, ID) pairs are always different.

With blind writes, a tuple may have its $wts > rts$. This means the latest version is created by a blind write and has not been read by any transaction yet. This does not affect the correctness of normal read and write operations.

8.3 Sundial Data Caching

If a distributed database contains some hotspot data that is read by transactions throughout the cluster, transactions accessing the data have to touch multiple servers regardless of where the hotspot data is mapped to. If the hotspot data is mostly read-only, one solution to reduce distributed transactions is to replicate the readonly hotspot data across all the servers so that every transaction can access it locally [37, 120]. This solution, however, has several downsides. First, the database administrator (DBA) needs to either profile the workloads or manually specify what data should be replicated. This is a daunting task in a rapidly evolving database. Second, when replicated data is updated, all nodes need to be notified of the update to maintain consistency. This incurs extra complexity and overhead. Third, full table replication increases memory footprint, which can be problematic if the replicated tables are large.

A more flexible solution to handle hotspot data is caching. Specifically, the DBMS can automatically decide what remote data to cache in a server's local memory without the involvement of DBAs or database users. A query finding the data in a local cache does not have to contact the remote server, saving both network latency and traffic.

Implementing caching in an OLTP DBMS, however, is challenging. The main difficulty comes from the complexity to ensure coherence among distributed caches. Namely, when a write happens in the system, the write needs to propagate to all the caches that store the data, such that serialization is not violated due to a transaction reading stale data from its local cache. Cache coherence is notoriously difficult to design and verify. It has been widely studied in distributed shared memory systems in both hardware [132] and software [83, 97]. It has also been studied in the context of client-server object-oriented databases in the 1990s [52, 9, 85]. Traditional cache coherence protocols are *invalidation-based*: when a tuple is modified, messages are sent to all caches holding copies to invalidate their now-stale copy of the tuple [123]. The protocol is complex, incurs coherence traffic in the network, and is difficult to implement efficiently [133]. Given the already-complex concurrency control protocol, the extra complexity of caching has led to limited adoption of the idea in distributed OLTP DBMSs.

Caching, however, can be very efficiently supported in Sundial without an extra complex protocol. Sundial is based on logical leases, which is what our Tardis cache coherence protocol in Chapter 3 is based on. Therefore, implementing logical-lease-based caching on top of the Sundial distributed concurrency control algorithm (cf. Section 8.2) is relatively straightforward.

In this section, we show how caching is supported in Sundial. We present the overall caching architecture in Section 8.3.1, the coherence protocol in Section 8.3.2, different caching policies in Section 8.3.3 and optimization techniques in Section 8.3.4.

8.3.1 Caching Architecture

Figure 8-4 shows an overview of Sundial's caching architecture. The system only caches tuples for read operations but not for writes. If a transaction attempts to write to a cached tuple, then it updates the data and leases of both the remote tuple and the locally cached copy. The DBMS maintains the cache as a network-side buffer at each node. For a read query, the DBMS always checks the coordinator's local cache. If the requested tuple exists in the cache, then the DBMS may read that local copy (which contains both the data and the lease). Otherwise, the DBMS sends the query to the appropriate remote node and the response is stored in the coordinator's cache for future accesses.

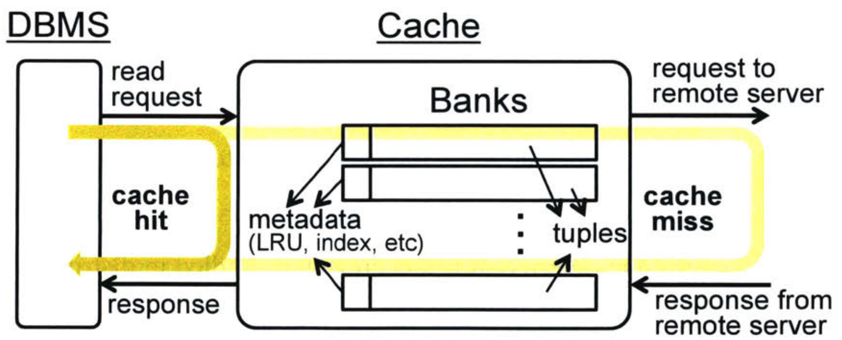


Figure 8-4: **Caching Architecture** – The cache is a network side buffer organized in multiple banks.

To avoid any centralized bottleneck, the DBMS organizes the cache in multiple banks. Each bank maintains the metadata of a fraction of cached tuples. The metadata includes a small index indicating what tuples are cached in the bank. It also keeps track of the occupancy of the bank. When the bank is full, tuples are replaced following the LRU replacement policy.

8.3.2 Cache Coherence with Logical Leases

The main challenge of coherent distributed caches is to ensure that the DBMS maintains a transaction's consistency when it reads cached data. To guarantee that cached data is always up-to-date, existing caching protocols either need 1) an invalidation mechanism to update *all* cached copies for each tuple write, or 2) to check the freshness of data for each local cache hit. The former means that all the caches always have the latest version of a tuple. But broadcasting on each tuple write incurs significant network latency and bandwidth overhead. The latter solution does not have the broadcasting overhead, but incurs a remote request for each cache hit, which offsets the gain that caching provides.

In Sundial, an invalidation mechanism is not required due to its logical leases (which is similar to Tardis, Chapter 3). It also avoids the necessity of contacting the remote node for each cache hit. Sundial caches a remote tuple by storing both the data and the lease locally. A transaction can read the cached tuple as long as the lease can satisfy the transaction's *commit_ts*. Conceptually, Sundial's cache coherence protocol has two major differences from existing coherence protocols. First, it is lease-based. A tuple write is not responsible for invalidating cached copies, instead, each read is responsible for getting the correct data version by taking a lease. A transaction can read a cached tuple provided that the lease is still valid (i.e., *commit_ts* falls within the lease). Second, since the lease is logical, any node caching a tuple does not have to detect a remote write immediately after it happens in physical time. A transaction may continue reading the physically stale cached version for as long as it is logically valid (i.e., the lease can satisfy its *commit_ts*). If the DBMS discovers that the lease is too old when it validates the transaction, it sends a request to that tuple's home node to extend the lease.

Compared to other caching protocols [52, 85], caching in Sundial has lower complexity due to the elimination of multicasting invalidations. In Sundial, caching does not introduce new messages, but only removes or reduces the size of existing messages.

The caching mechanism discussed so far works for `SELECT`s with an equality predicate on the table's primary key. But the same technique can also be applied to range scans or secondary index lookups. Since the index nodes also maintain leases, the DBMS can cache the index nodes in the same way as it caches tuples. An index insert/delete is similar to a tuple write, and is detected by the transaction when the end of the lease on an index node is smaller than *commit_ts* of the transaction.

8.3.3 Policies

There are different ways to manage the cache at each node. We discuss a few possibilities in this section and show their advantages and disadvantages.

Always Reuse: The simplest approach is to always return the cached tuple to the transaction for each cache hit. This works well for read-only tuples, but can hurt performance for tuples that are modified often since it leads to more transaction aborts. If a cached tuple has an old lease, then it is possible that the tuple has already been modified by another transaction at its home node. If a transaction reads the stale cached tuple, then the DBMS may try to extend the lease when the transaction attempts to commit in the validation phase. But the lease extension may fail because another transaction modified the tuple at the home node, which causes the current transaction to abort.

Always Request: An alternative policy is where the DBMS always sends a request to the remote node to retrieve the data even for a cache hit. Caching still provides some benefit in this scheme. The DBMS sends a query request to a remote node that contains the query as well as the *wts* of the expected tuple(s) that the query will return (if they exist in its cache). If the tuple at the home node has the same *wts* as the cached tuple, then this means that the requesting node contains the correct version and the DBMS does not need to return the data in the response. Although the transactions' latencies do not change, this technique does reduce the amount of network traffic in the system compared to the no caching scheme.

Hybrid: Lastly, Sundial uses a hybrid caching policy that strives to achieve the benefits of both the always request and always reuse policies. At each node, the DBMS counts the number of remote reads and writes that it sends for each table. Then, the DBMS periodically checks to see whether a table is read intensive (i.e., the ratio between remote reads and writes is greater than a certain threshold), and then switches to the always reuse scheme. Otherwise, the DBMS simply uses the always request scheme to avoid increasing the number of transaction aborts.

8.3.4 Read-Only Table Optimizations

We can extend Sundial's caching scheme to optimize accesses to read-only tables. The problem is that the lease of a cached read-only tuple may be smaller than a transaction's *commit_ts*, which means the DBMS has to continually extend the tuple's lease at its home node. These frequent lease extensions are essentially wasted work that increases the amount of network traffic in the system. We now describe two techniques that Sundial uses to reduce the number of lease extension requests for such read-only tables. We evaluate their efficacy in Section 8.5.4.

The first optimization is to track and extend leases at the table granularity to amortize the cost of lease extension. The DBMS can discover that a table is read-only or read-intensive because it has a large ratio between reads and writes. For each table, the DBMS maintains a *max_wts* which is the largest *wts* of all its tuples. *max_wts* is updated whenever one of its tuples advances *wts*. A read-only table also maintains *max_rts*; this means that the leases for all of the table's tuples are at least *max_rts*, even if the tuple's *rts* shows a smaller value. If any tuple is modified, its new *wts* must be at least *Max(rts +1, max_rts +1)*. When the DBMS requests a lease extension for a tuple in a read-only table, all the leases in the table are extended by advancing *max_rts*. The *max_rts* is returned to the requesting node's cache. A future cache hit in that table considers the end of the lease to be *Max(max_rts, rts)*.

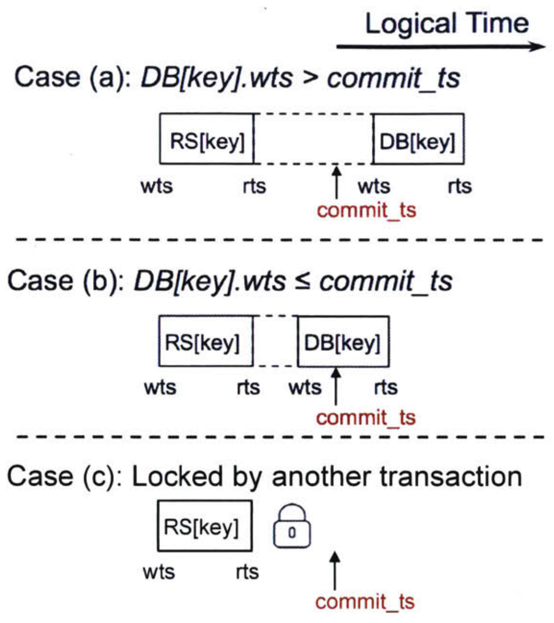
Another technique to amortize the lease extension cost is to speculatively extend the lease to a larger timestamp than what is asked for. Instead of extending the *rts* (or *max_rts*) to the *commit_ts* of the requesting transaction, Sundial extends *rts* to $commit\_ts + lease$ for presumed read-only tables. *lease* increases as more confidence is gained that the table is indeed read-only. This reduces the number of lease extensions since each one is more effective. The two optimizations discussed above are orthogonal to each other and both are implemented in Sundial.

8.4 Discussion

We now discuss two additional aspects of Sundial's concurrency control protocol and caching scheme described in the previous sections. We first characterize the types of transaction aborts that can occur at runtime and discuss ways for the DBMS to reduce them. We also compare Sundial with dynamic timestamp allocation [17] algorithms and show the advantages that Sundial's logical leases provide.

8.4.1 Transaction Aborts

As we described in Section 8.2.2, write-write conflicts are difficult for a DBMS to prevent and thus they are not the main focus of Sundial. Therefore, we focus our discussion on aborts caused by



Logical Time
Case (a): DB[key].wts > commit_ts
RS[key]
DB[key]
wts
rts
wts
rts
commit_ts
Case (b): DB[key].wts ≤ commit_ts
RS[key]
DB[key]
wts
rts
wts
rts
commit_ts
Case (c): Locked by another transaction
RS[key]
0
wts
rts
commit_ts

Figure 8-5: **Transaction Aborts** – Three examples of when the DBMS will abort a transaction under Sundial due to read-write conflicts.

read-write conflicts.

There are three conditions in Sundial that a transaction's *commit_ts* must satisfy before the DBMS is allowed to commit it:

$$commit\_ts \geq tuple.wts, \qquad \forall tuple \in RS \qquad (8.1)$$

$$commit\_ts \leq tuple.rts, \qquad \forall tuple \in RS \qquad (8.2)$$

$$commit\_ts \geq tuple.rts + 1, \qquad \forall tuple \in WS \qquad (8.3)$$

Condition (8.1) is always satisfied since *commit_ts* is updated for each read. Likewise, Condition (8.3) is also always satisfied because each write updates the *commit_ts* and no other transaction can modify the tuple's *rts* because of the lock. Therefore, the DBMS only aborts a transaction if it fails Condition (8.2) in the prepare phase; this is because the transaction fails to extend a lease since the tuple was locked or modified by another transaction (cf. Algorithm 10).

There are three scenarios where a transaction fails Condition (8.2) above, as illustrated in Figure 8-5.

Case (a): In this scenario, the tuple's *wts* in the database is greater than or equal to transaction *T1*'s *commit_ts*. The DBMS must abort *T1* because it is unknown whether or not it can extend the tuple's *rts* to *commit_ts*. It is possible that another transaction modified the tuple after *RS*[*key*].*rts* but before *commit_ts*, in which case the DBMS aborts *T1*. But it is also possible that no version was created before *commit_ts* such that the version in *T1*'s read set is still valid at *commit_ts* and therefore *T1* can commit. This uncertainty can be resolved by maintaining a history of recent *wts*'s in each tuple [158].

Case (b): Another transaction already wrote the latest version of the tuple to the database before *T1*'s *commit_ts*. The DBMS is therefore unable to extend the lease of the transaction's local version to *commit_ts*. As such, the DBMS has to abort *T1*.

Case (c): Lastly, in this example the DBMS is unable to extend the tuple's *rts* because another transaction holds the lock for it. Again, this will cause DBMS to abort *T1*.

For the second and third scenarios, Sundial is potentially able to avoid them if the DBMS lets each read operation extend the tuple's lease during the execution phase. This may reduce the number of renewals during the prepare phase, thereby leading to fewer aborts. But speculatively extending the leases in this manner also causes the transaction that updates the tuple to jump further ahead in logical time, leading to even more extensions and potential aborts. We defer to future work the understanding of under what workload conditions such optimizations would improve performance in Sundial.

8.4.2 Sundial vs. Dynamic Timestamp Allocation

There are previous proposals for concurrency control protocols that use dynamic timestamp allocation (DTA) to determine transactions' logical commit timestamps [17]. In DTA, the DBMS assigns each transaction a timestamp range when the transaction starts (e.g., 0 to infinity). Then, when the DBMS detects a conflict, it shrinks the transaction's range and also pushes the boundary of ranges of transactions in conflict with it such that ranges of conflicting transactions do not overlap. If a transaction's timestamp range is not empty when it commits, then the DBMS can pick any timestamp from its range as the commit timestamp. Otherwise, the DBMS will abort the transaction. Similar to Sundial, DTA dynamically orders transactions with read-write conflicts to minimize aborts. DTA has been applied to OCC algorithms in real-time DBMSs [91, 96], single-node multi-version concurrency control [100], and distributed single-version concurrency control protocols [103].

There is one fundamental drawback of DTA algorithms that makes them less efficient than Sundial in a distributed environment. The key observation is that DTA requires the DBMS to *explicitly coordinate* transactions to shrink their timestamp ranges when a conflict occurs. In contrast, with Sundial the DBMS only considers the tuples that a transaction accessed to determine a transaction's commit timestamp. Although DTA's coordination overhead is acceptable in a single-node DBMS setting [100], we find that distributed transactions that conflict with transactions running at other nodes increase the complexity of timestamp range coordination and therefore degrade performance.

We use MaaT [103], a distributed DTA-based concurrency control protocol, as an example to illustrate the problem. In MaaT, the DBMS assigns a transaction with the initial timestamp range of $[0, +\infty)$. The DBMS maintains the range at each node accessed by the transaction so that it can be adjusted locally when a conflict occurs. When a transaction begins the validation process after execution, the DBMS needs to determine whether its timestamp range is empty or not. This is done by locking the range at each participating node (through the prepare phase) and taking the union of the ranges at the coordinator. Locking is necessary to avoid other conflicting transactions from modifying the range. The problem, however, is that many transactions in MaaT have ranges with an upper bound of $+\infty$. Thus, after they locked their timestamp ranges in the prepare phase, transactions that depend on them will not have a feasible range and have to be aborted. This problem is not MaaT-specific, but fundamental to all DTA protocols. As we will show in Section 8.5, these aborts severely limit the performance of MaaT.

8.5 Experimental Evaluation

We now evaluate Sundial's performance for OLTP workloads. We implemented Sundial in a distributed DBMS testbed based on the DBx1000 DBMS [157]. We extended it to work in a distributed setting. Each node in the system has one input and one output thread for inter-node communication. The DBMS designates all other threads as workers that communicate with input/output threads through asynchronous buffers. The testbed's workload driver submits transaction requests in a blocking manner with one open transaction at a time per worker thread.

At runtime, the DBMS puts transactions that abort due to contention (e.g., lock acquisition failure, validation failure) into an abort buffer. It then restarts these transactions after a small back-off time (randomly selected between 0–1 ms). The DBMS does not restart transactions caused by user-initiated aborts.

Most of the experiments are performed on a cluster of four nodes running Ubuntu 14.04. Each node contains two Intel Xeon E5-2670 CPUs (8 threads each $\times$ 2 with HT) and 64 GB of DRAM. The nodes are connected together with a 10 GigE network. For the datacenter experiments in Sections 8.5.8 and 8.5.9, we use the Amazon EC2 platform. For each experiment, the DBMS runs for a warm-up period of 30 seconds, and then results are collected for the next 30 seconds run.

8.5.1 Workloads

We use two different OLTP workloads in our evaluation. All transactions execute as stored procedures that contain program logic intermixed with queries.

YCSB: The Yahoo! Cloud Serving Benchmark [35] is a synthetic benchmark modeled after cloud services. It contains a single table that is partitioned across servers in a round-robin fashion. Each partition contains 10 GB data with 1 KB tuples. Each transaction accesses 16 tuples as a mixture of reads (90%) and writes (10%) with on average 10% of the accesses being remote (selected uniformly at random). The queries access tuples following a power law distribution characterized by a theta parameter. By default, we use theta = 0.9 which means 75% of all accesses go to 10% of hot data.

TPC-C: This is the standard benchmark for evaluating the performance of OLTP DBMSs [140]. It models a warehouse-centric ordering processing application that contains five transaction types. All the tables except `ITEM` are partitioned based on the warehouse ID. By default, the `ITEM` table is replicated in all nodes' main memory. We use a single warehouse per node to model high contention.

8.5.2 Concurrency Control Algorithms

We implemented the following concurrency control algorithms in our testbed for our evaluation:

2PL: We used a deadlock prevention variant of 2PL called *Wait-Die* [18]. A transaction is allowed to wait to acquire a lock if its priority is higher than the current lock owner; otherwise the DBMS will abort it. We used the current wall clock time attached with the thread id as the metric of priority. This algorithm is similar to the approach used in Google Spanner [36].

Google F1: This is an OCC-based algorithm used in Google's F1 DBMS [130]. During the read-only execution phase, the DBMS tracks a transaction's read and write set. When the transaction goes to commit, the DBMS then locks all of the tuples accessed by the transaction. It will then abort the transaction if it fails to acquire any of these locks or if the latest version of any tuple is different from the version it saw during the execution phase.

MaaT: As described in Section 8.4.2, this is a state-of-the-art DTA protocol [103]. We integrated the original MaaT source code into our testbed. We also improved the MaaT implementation by (1) reducing unnecessary network messages, (2) adding multi-threading support, and (3) improving its garbage collection.

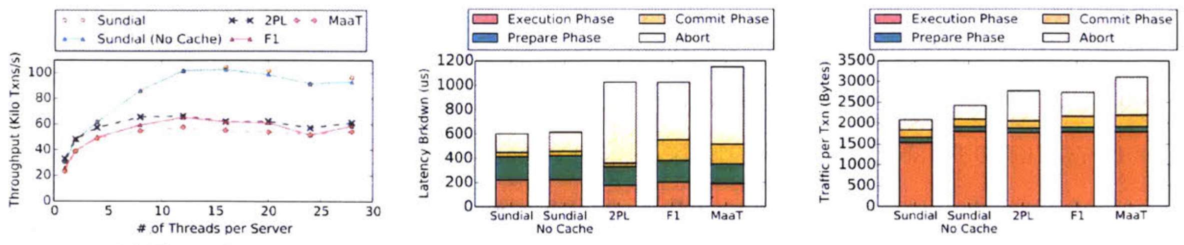
Sundial: Our approach from Section 8.2. We enable all of Sundial's caching optimizations from Section 8.3 unless otherwise stated for a particular experiment. Each node maintains a local cache of 1 GB. Sundial by default uses the hybrid caching policy with a threshold of 1% writes to determine whether a table is read intensive or not.

8.5.3 Performance Comparison

We now quantitatively compare Sundial to the other baseline concurrency control algorithms using the YCSB and TPC-C workloads on four servers. For each workload, we report throughput as we sweep the number of worker threads from 1 to 28. After 28 threads, the DBMS's performance drops due to context switching. We run Sundial with and without caching enabled (using the hybrid policy).

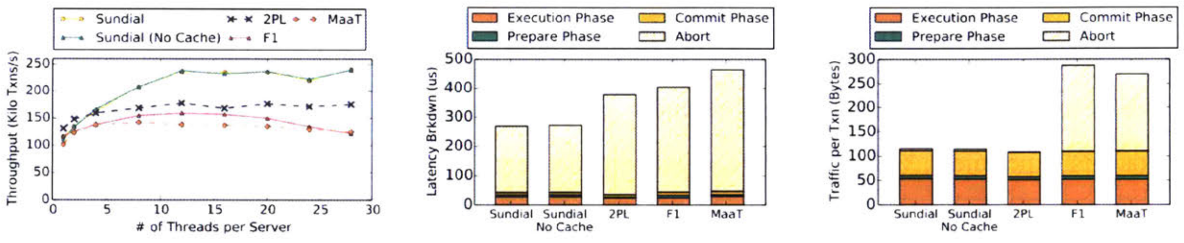
In addition to throughput measurements, we also provide a breakdown of transactions' latency measurements and network traffic. These metrics are divided into Sundial's three phases (i.e., execution, prepare, and commit), and when the DBMS aborts a transaction.

The results in Figures 8-6a and 8-7a show that Sundial outperforms the best evaluated baseline algorithm (i.e., 2PL) by 57% in YCSB and 34% in TPC-C. Caching does not improve performance in these workloads in the current configuration. For YCSB, this is because the transactions both read and write to the only table in the database and the fraction of write queries is high. Therefore, a remote query always sends a request message to the remote node even for a cache hit. As such, a transaction has the same latency regardless of whether caching is enabled. In TPC-C, all remote requests are updates instead of reads, therefore Sundial's caching does not help. In these experiments, we configured the DBMS to manually replicate the ITEM table on every node, so all accesses to ITEM are local.



(a) Throughput (b) Latency Breakdown (c) Traffic Breakdown

Figure 8-6: **Performance Comparison (YCSB)** – Runtime measurements when running the concurrency control algorithms for the YCSB workload.



(a) Throughput (b) Latency Breakdown (c) Traffic Breakdown

Figure 8-7: **Performance Comparison (TPC-C)** – Runtime measurements when running the concurrency control algorithms for the TPC-C workload.

Figures 8-6b and 8-6c show the latency and network traffic breakdown of different algorithms on YCSB at 16 threads. We attribute Sundial's lower latency and network traffic compared to other algorithms to the fewer number of transaction aborts that the DBMS incurs, which is due to the dynamic timestamp assignment for read-write conflicts. Enabling Sundial's caching scheme further reduces traffic in the execution phase. This is because for a cache hit that contains the up-to-date tuple, although the transaction sends a request to the remote node, the data is not sent back to the coordinator (cf. Section 8.3.3). We will provide a more detailed analysis of caching in Sections 8.5.4 and 8.5.5.

Another interesting observation in Figure 8-6b is that F1 and MaaT both incur considerable latency in the commit phase while 2PL and Sundial do not. This is because in both 2PL and Sundial, if the sub transaction is read-only on a remote node, the commit phase on that node can be skipped. In F1 and MaaT, however, this optimization cannot be applied since they have to either release locks (F1) or clear timestamp ranges (MaaT) in the commit phase; a round trip message is required in the commit phase to perform these operations.

The latency and traffic breakdown of TPC-C (Figures 8-7b and 8-7c) show a trend similar to

YCSB in that there are significant gains from reducing the cost of aborts. Since only one warehouse is modeled per node in this experiment, there is high contention on the single row in the WAREHOUSE table. As a result, all algorithms spent significant time on aborted transactions. But both 2PL and Sundial do not incur much traffic for aborted transactions. This is because contention on the WAREHOUSE table happens at the beginning of each transaction. In 2PL, the DBMS resolves conflicts immediately (by letting transactions wait or abort) before sending out any remote queries. Similar to 2PL, Sundial resolves write-write conflicts early as well; for read-write conflicts, Sundial's logical leases allow it to resolve most conflicts by dynamically determine the commit order, without aborting transactions.

8.5.4 Caching with Read-Only Tables

We next measure effectiveness of Sundial's caching scheme on databases with read-only tables. For this experiment, we use the TPC-C benchmark because it contains a table (ITEM) that is read-only and is shared by all the database partitions. To avoid remote queries on the ITEM, our TPC-C implementation so far replicates the table across all of the partitions. Table replication is a workload-specific optimization that requires extra effort from DBAs [120, 37]. In contrast, caching is more general and completely transparent, and therefore much easier to use. We study the performance of Sundial's different caching policies using the TPC-C benchmark. For each caching policy, we show performance as the number of warehouses changes in the workload.

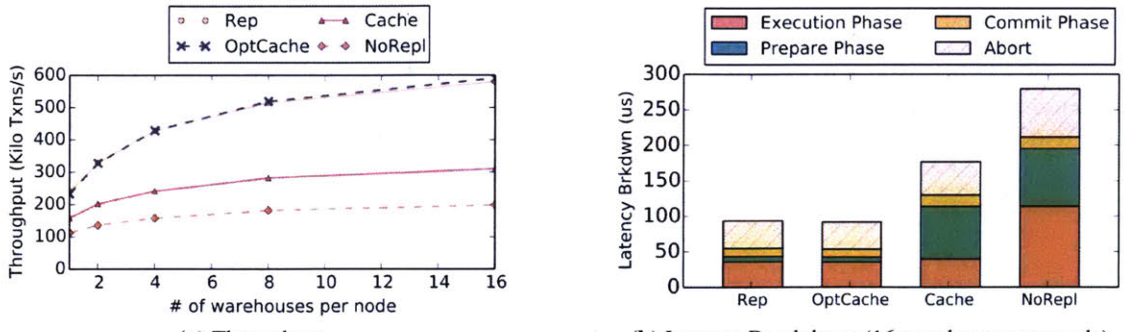
For this experiment, we use two configurations for the ITEM table:

- **Replication (Rep):** The DBMS replicates the table across all the partitions, thereby all accesses to the table are local.
- **No Replication (NoRep):** The DBMS hash partitions the ITEM on its primary key. A significant portion of queries on this table have to access a remote node.

For the configuration without table replication, we test two different caching configurations:

- **Default Caching (Cache):** The original caching scheme described in Section 8.3.2.
- **Caching with Optimizations (OptCache):** Sundial's caching scheme with the two read-only optimizations from Section 8.3.4.

According to Figure 8-8, not replicating the ITEM table shows a significant performance penalty. This is because the table is intensively read by a large fraction of transactions (i.e., all NewOrder transactions that are 45% of the workload). Most of them are distributed if ITEM is not replicated.



(a) Throughput (b) Latency Breakdown (16 warehouses per node)

Figure 8-8: **Caching with Read-Only Tables** – Performance of different TPC-C configurations in Sundial with caching support.

With the caching support in Sundial, however, the performance gap can be completely closed. Since the ITEM table is small, Sundial caches the full ITEM table at each node. Essentially, caching achieves the same benefit as manual table replication but hides the complexity from the database users.

From Figure 8-8, we observe that the read-only table optimizations are important for performance. Without the optimizations, Sundial achieves modest performance gain through caching, mainly by reducing the traffic overhead (but not latency overhead) of fetching remote data during the execution phase (Figure 8-8b). However, a cached tuple in the ITEM table needs frequent lease extension during the prepare phase. All of these extensions are successful. But the latency incurred hurts performance. With the read-only table optimization, all leases in the ITEM table are extended together when one of them needs to be, thereby greatly amortizing the cost.

8.5.5 Caching Policies

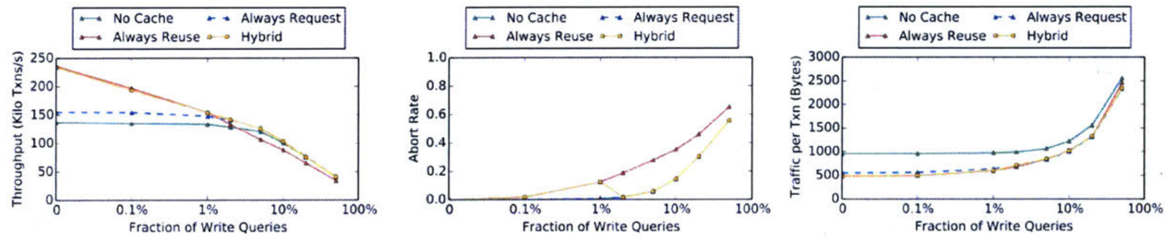
We now evaluate Sundial's caching policies for read-write tables that we presented in Section 8.3.3:

- **No Cache:** The DBMS's caching scheme is disabled.
- **Always Reuse:** The DBMS always reads a cached tuple if it exists in its local cache.
- **Always Request:** The DBMS always sends a request to retrieve the latest version of the tuple even if it exists in its local cache.
- **Hybrid:** The DBMS always reuses cached tuples for read-intensive tables and always requests otherwise. A table is considered to be read-intensive if less than 1% of accesses to the table are writes.

For these experiments, we use YCSB and sweep the percentage of write queries in transactions.

This changes whether or not the DBMS designates the single table in YCSB as read-intensive.

The results in Figure 8-9 show that reusing the cache improves performance when the data is read-intensive. In this case, cached tuples are unlikely to be stale, therefore reading them does not cause many unnecessary aborts. As the number of writes increases, however, many of the transactions abort because they read stale cached tuples. This makes the performance of always reuse even worse than just disabling the cache when more than 1% of queries are writes.



(a) Throughput (b) Abort Rate (c) Network Traffic

Figure 8-9: **Caching Policies** – Sweeping the percentage of write queries in YCSB.

In contrast, the DBMS never performs worse when always requesting compared to no caching. It has the same abort rate as no caching since a transaction always reads the latest tuples. But the DBMS also incurs lower network traffic than no caching since cache hits on fresh tuples do not require data transfer. Thus, when a table is read intensive, this leads to some performance improvement but not as much as the always reusing configuration.

Lastly, the hybrid scheme combines the best of both worlds by adaptively choosing between always reusing and always requesting. This allows the DBMS to achieve the best throughput of all the schemes with the lowest network traffic.

8.5.6 Measuring Aborts

We designed the next experiment to better understand how transaction aborts occur in the Sundial and MaaT protocols. For this, we executed YCSB with the default workload mixture for transactions. We instrumented DBx1000 to record the reason why the system aborts a transaction due to a conflict. A transaction is counted multiple times if it is aborted and restarted multiple times. To ensure that each protocol has the same amount of contention in the system, we keep a constant number of active transactions running during the experiment.

The tables in Figure 8-10 show the percentage of transactions that were aborted from all transactions executed. We see the abort rate of Sundial is 3.3× lower than that of MaaT. The main cause of

Abort Cases	Abort Rate
Case (a): $commit\_ts < DB[key].wts$	1.79%
Case (b): $commit\_ts \geq DB[key].wts$	1.56%
Case (c): tuple locked	6.60%
Aborts by W/W Conflicts	4.05%
Total	14.00%

(a) Sundial

Abort Cases	Abort Rate
Conflict with $[x, \infty)$	42.21%
Empty range due to other conflicts	4.45%
Total	46.66%

(b) MaaT

Figure 8-10: **Measuring Aborts** – The different types of aborts that occur in Sundial and MaaT for the YCSB workload. For Sundial, we classify the aborts due to read-write conflicts into the three categories from Section 8.4.1.

aborts in MaaT is due to conflicts with locked range of $[x, \infty)$ where x is some constant. As discussed in Section 8.4.2, this happens when a transaction reads a tuple and enters the prepare phase with a timestamp range of $[x, \infty)$. While the transaction is preparing, a subsequent transaction that writes to the same tuple has to abort due to an empty timestamp range (cf. Section 8.4.2). In Sundial, most of the aborts are caused by case (c) in read-write conflicts, where a transaction tries to extend a lease that is locked by another writing transaction. There are also many aborts due to write-write conflicts. The number of aborts due to case (a) and case (b) are about the same and lower than the other two cases.

8.5.7 Dynamic vs. Static Timestamp Assignment

One salient feature of Sundial is its ability to dynamically determine the commit timestamp of a transaction to minimize aborts. Many existing concurrency control protocols, in contrast, assign a static timestamp to each transaction when it starts and use multi-versioning to avoid aborting read queries that arrive late [41, 1, 98]. The inability to flexibly adjust commit order, however, can lead to unnecessary aborts due to writes that arrive late (i.e., a write request to a tuple arrives after a read to the tuple has been performed with a timestamp larger than the timestamp of the writing transaction).

In this experiment, we compare Sundial without caching against a multi-version concurrency control (MVCC) protocol with varying amounts of clock skew between nodes (using `ptp` [5]). Our MVCC implementation is idealized as it does not maintain the data versions, and therefore does not have associated overhead in memory and computation (e.g., garbage collection) [155]. This allows us to just compare the amount of concurrency enabled by Sundial and MVCC.

In Figure 8-11, we observe that even with no skew (skew less than 10 μs), MVCC has slightly worse performance than Sundial. This degradation is mostly caused by the extra aborts due to writes that arrive late. In Sundial, the dynamic timestamp assignment can move these writes to a later timestamp and thus reduce these aborts. Increasing the clock skew further reduces the throughput of the MVCC protocol. This is because writes from nodes that fall behind in time will always fail due to reads to the same tuples from other nodes. The DBMS's performance with Sundial does not suffer with higher amounts of clock skew since its timestamps are logical.

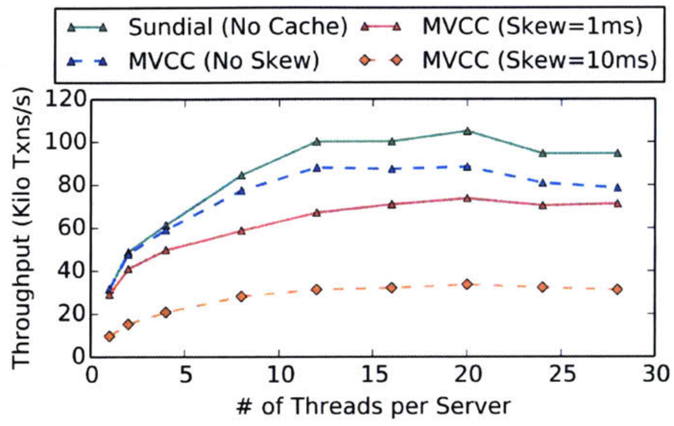


Figure 8-11: **Dynamic vs. Static Timestamp Assignment** – Performance comparison between Sundial and a baseline MVCC protocol that statically assigns timestamps to transactions.

8.5.8 Scalability

For the final two experiments, we deployed DBx1000 on Amazon EC2 to evaluate it in larger and more geographically dispersed clusters. We first study the scalability of the concurrency control protocols as we increase the number of nodes in the cluster. Each node is an `m4.2xlarge` instances type with eight virtual threads and 32 GB main memory. We assign two threads to handle the input and output communications, and the remaining six are used as worker threads. We run the YCSB workload using the workload mixture described in Section 8.5.1.

The first notable result in Figure 8-12 is that the performance of all the protocols drop to the same level when the node count increases from one to two. This is due to the overhead of the DBMS having to coordinate transactions over the network [69]. Beyond two nodes, however, the performance of all of the algorithms increase as the number of nodes increases. We see that the performance advantage of Sundial remains as the node count increases. Again, we attribute this to the reduction in the number of transaction aborts due to the dynamic timestamp management of

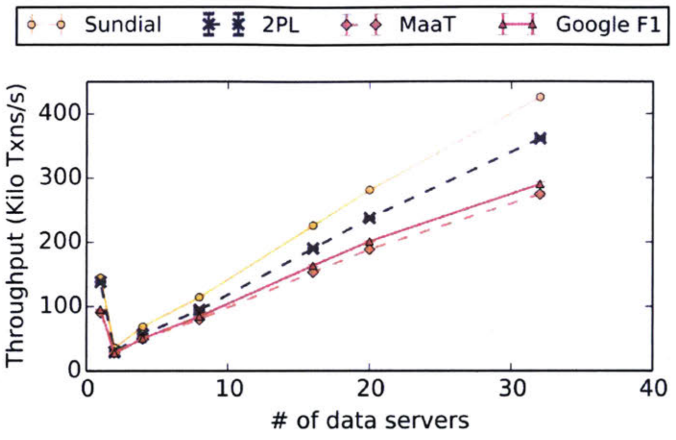
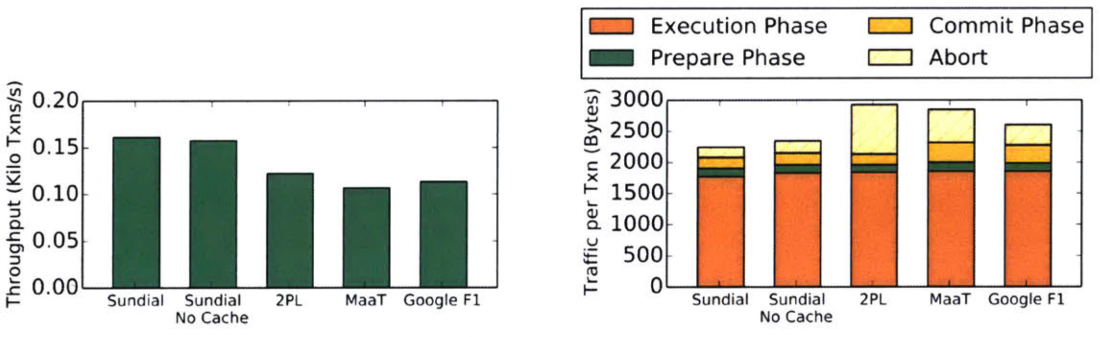


Figure 8-12: **Scalability** – Throughput of concurrency control algorithms for the YCSB workload on Amazon EC2.

Sundial.

8.5.9 Cross-Datacenter Transactions

Lastly, we measure how Sundial performs when transactions have to span geographical regions. In the previous experiments, all of the nodes are located in the same data center and therefore have low communication latency with each other. We deployed DBx1000 on Amazon EC2 in an eight node cluster with each node located in a different datacenter [1]. We ran the YCSB workload again and compared the different concurrency control protocols. The results are shown in Figure 8-13.



(a) Throughput **(b)** Traffic per Transaction

Figure 8-13: **Cross-Datacenter Transactions** – Throughput and Traffic per Transaction when nodes are placed in different data centers.

From the results, we observe that all the concurrency control algorithms suffer from much higher network latencies (on average 100+ ms round trip) due to cross-continent communications, which

[1]North Virginia (us-east-1), North California (us-west-1), Ireland (eu-west-1), Frankfurt (eu-central-1), Singapore (ap-southeast-1), Sydney (ap-southeast-2), Tokyo (ap-northeast-1), and Mumbai (ap-south-1)

leads to three orders of magnitude reduction in performance. As shown in the figure, Sundial still outperforms the other baseline concurrency control algorithms. Caching slightly improves the performance of Sundial due to the reduction of network traffic, which is especially expensive in this operating environment.

Chapter 9

Conclusion and Future Work

This thesis has presented a new technique, logical leases, to improve performance and scalability of shared-memory parallel systems in both hardware and software. In particular, we have made the following contributions.

Scalable cache coherence protocols: We have built Tardis, a scalable cache coherence protocol based on logical leases. Compared to the widely used full-map directory-based coherence protocols, Tardis has less storage and network message overhead and achieves better performance, while being simpler to reason about. We have described the basic Tardis protocol and evaluated its performance compared to other coherence protocols, proposed optimizations and extensions to Tardis, and formally proved the correctness of Tardis.

Scalable concurrency control algorithms: We have studied the scalability of seven traditional concurrency control algorithms in a simulated 1000-core processor, and found that none of the algorithms scale. For each algorithm, we identified both fundamental and artificial bottlenecks. We have proposed TicToc, a new concurrency control algorithm for multicore databases. TicToc eliminates the scalability bottlenecks in traditional concurrency control algorithms through using logical leases. We have built Sundial, a new distributed concurrency control algorithm using logical leases to seamlessly integrate concurrency control and caching into a unified, low-complexity protocol. Both TicToc and Sundial significantly improve performance and reduce abort rate compared to their counterparts.

The ideas in this thesis can motivate the following interesting future research directions:

Multi-chip shared-memory processor: In this thesis, the Tardis cache coherence protocol is designed and evaluated in the environment of a single-chip multicore processor. A computer, however, may contain multiple processor chips. Applying Tardis to multi-chip systems has different design trade-offs than a single-chip environment. There are at least two interesting research questions. First, how should Tardis be implemented across chips where each chip implements a traditional cache coherence protocol (e.g., directory-based)? This enables large scale multi-chip systems using existing commercial processors as building blocks. Second, how should Tardis be deployed over a heterogeneous system, where each chip has its own coherence requirement? Due to the scalability of Tardis, the number of processing units do need to be determined when the coherence protocol is implemented.

Distributed Shared Memory (DSM) Systems: Today, multicore and multi-socket processors have a shared memory programming model and enforce coherence in hardware, while clusters and distributed systems have a message-passing programming model and handle communication in software. As network bandwidth increases and latency reduces, the boundary between multicore and distributed environments blur. It is an interesting research direction to study how to extend the shared programming model to multiple servers. These systems are known as distributed shared memory (DSM) systems. One interesting future research project is to apply Tardis to this environment, and study what hardware and/or software changes are required for maximizing performance. It is also interesting to study how Tardis leverages new hardware features, including hardware transactional memory (HTM) and remote direct memory access (RDMA), in a DSM system.

Transactional Memory: Transactional memory adds transactions to shared-memory systems in order to reduce complexity of synchronization, making it easier for programmers to write parallel codes. Transactional memory has been applied to both hardware and software, known as hardware transactional memory (HTM) and software transactional memory (STM), respectively. Traditional HTM and STM systems are implemented on top of the existing cache coherence protocols (e.g., directory-based protocols). Due to the complexity of the underlying coherence protocols and the difficulty of managing transactions, HTM and STM are challenging to implement. In fact, the first-generation Haswell processors that support HTM had to disable HTM functionality due to a design bug [67]. We believe that since logical leases simplify the design of both cache coherence and concurrency control protocols, it is promising to design HTM/STM systems based on logical leases.

Fault Tolerance of Logical Leases

One benefit of using leases for caching is the simplification of fault tolerance [59]. Compared to directory-based protocols, the amount of metadata maintained in a lease-based protocol is significantly reduced (i.e., from a sharer list to a simple lease). More important, leases are independent of the number of privates caches in the system. In a lease-based protocol, a non-responding private cache does not stall the whole system; other cores make forward progress without waiting for the non-responding core. It is interesting future work to study the implication of logical leases on fault tolerance in all types of shared-memory systems.

Bibliography

[1] CockroachDB. https://www.cockroachlabs.com.

[2] DBx1000. https://github.com/yxymit/DBx1000.

[3] H-Store: A Next Generation OLTP DBMS. http://hstore.cs.brown.edu.

[4] VoltDB. http://voltdb.com.

[5] IEEE standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–300, July 2008.

[6] Intel brings supercomputing horsepower to big data analytics. http://intel.ly/18A03EM, November 2013.

[7] NuoDB architecture. http://go.nuodb.com/rs/nuodb/images/Technical-Whitepaper.pdf, 2017.

[8] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[9] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD Record*, 24(2):23–34, 1995.

[10] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):182–205, 1993.

[11] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 353–362. ACM, 1998.

[12] Niket Agarwal, Li-Shiuan Peh, and Niraj K Jha. In-network coherence filtering: snoopy coherence without broadcasts. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 232–243. ACM, 2009.

[13] Niket Agarwal, Li-Shiuan Peh, and Niraj K Jha. In-network snoop ordering (INSO): Snoopy coherence on unordered interconnects. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 67–78. IEEE, 2009.

[14] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[15] Thomas J Ashby, Pedro Diaz, and Marcelo Cintra. Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters. *IEEE Transactions on Computers*, 60(4):472–483, 2011.

[16] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, and James W. Mehl. System R: Relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[17] Rudolf Bayer, Klaus Elhardt, Johannes Heigert, and Angelika Reiser. Dynamic timestamp allocation for transactions in database systems. In *2nd Int. Symp. on Distributed Databases*, pages 9–20, 1982.

[18] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[19] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983.

[20] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*, chapter 5. 1987.

[21] Phillip A. Bernstein, D.W. Shipman, and W.S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, 5(3):203–216, 1979.

[22] Ritwik Bhattacharya, Steven German, and Ganesh Gopalakrishnan. Symbolic partial order reduction for rule based transition systems. In *Correct Hardware Design and Verification Methods*. Springer Berlin Heidelberg, 2005.

[23] Ritwik Bhattacharya, Steven M. German, and Ganesh Gopalakrishnan. Exploiting symmetry and transactions for partial order reduction of rule based specifications. In *In Antti Valmari, editor, SPIN, volume 3925 of Lecture Notes in Computer Science*, pages 252–270. Springer, 2006.

[24] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[25] Roberto Bisiani, Andreas Nowatzyk, and Mosur Ravishankar. Coherent Shared Memory on a Distributed Memory Machine. In *In Proc. of the 1989 Int'l Conf. on Parallel Processing (ICPP'89)*, pages 133–141, 1989.

[26] Geoffrey M Brown. Asynchronous multicaches. *Distributed Computing*, 4(1):31–36, 1990.

[27] Jason F Cantin, Mikko H Lipasti, and James E Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 246–257. IEEE Computer Society, 2005.

[28] Michael J Carey, David J DeWitt, Michael J Franklin, Nancy E Hall, Mark L McAuliffe, Jeffrey F Naughton, Daniel T Schuh, Marvin H Solomon, CK Tan, Odysseas G Tsatalos, et al. *Shoring up persistent applications*, volume 23. ACM, 1994.

[29] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *Computers, IEEE Transactions on*, 100(12):1112–1118, 1978.

[30] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, 1990.

[31] David Chaiken, John Kubiatowicz, and Anant Agarwal. *LimitLESS directories: A scalable cache coherence scheme*, volume 26. ACM, 1991.

[32] Xiaofang Chen, Yu Yang, Ganesh Gopalakrishnan, and Ching-Tsun Chou. Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. *Form. Methods Syst. Des.*, 36(1):37–64, February 2010.

[33] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V Adve, Vikram S Adve, Nicholas P Carter, and Ching-Tsun Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[34] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *in Formal Methods in Computer Aided Design*, pages 382–398. Springer, 2004.

[35] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC'10*, pages 143–154.

[36] James C Corbett et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3), 2013.

[37] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[38] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Symposium on Operating Systems Principles*, pages 33–48, 2013.

[39] Giorgio Delzanno. Automatic verification of parameterized cache coherence protocols. In E.Allen Emerson and AravindaPrasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer Berlin Heidelberg, 2000.

[40] Christopher Dennl, Daniel Ziener, and Jurgen Teich. On-the-fly composition of fpga-based sql query accelerators using a partially reconfigurable module library. In *FCCM*, pages 45–52, 2012.

[41] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *SIGMOD*, pages 1243–1254, 2013.

[42] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol verification as a hardware design aid. In *Computer Design: VLSI in Computers and Processors, 1992. ICCD '92. Proceedings, IEEE 1992 International Conference on*, pages 522–525, Oct 1992.

[43] Jack Dongarra. Toward a new metric for ranking high performance computing systems.

[44] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *SOSP*, pages 54–70, 2015.

[45] Marco Elver and Vijay Nagarajan. TSO-CC: Consistency directed cache coherence for TSO. In *International Symposium on High Performance Computer Architecture*, pages 165–176, 2014.

[46] E. Allen Emerson and Vineet Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*, pages 247–262, 2003.

[47] 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):624–633, 1976.

[48] Jason Evans. jemalloc. `http://canonware.com/jemalloc`.

[49] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11):1190–1201, 2015.

[50] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005.

[51] Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 1988.

[52] Michael J Franklin, Michael J Carey, and Miron Livny. Transactional client-server cache consistency: Alternatives and performance. *ACM Transactions on Database Systems (TODS)*, 22(3):315–363, 1997.

[53] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, December 1992.

[54] Kourosh Gharachorloo. *Memory consistency models for shared-memory multiprocessors*. PhD thesis, Stanford University, 1996.

[55] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *In Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, 1991.

[56] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. *Memory consistency and event ordering in scalable shared-memory multiprocessors*, volume 18. ACM, 1990.

[57] Sanjay Ghemawat and Paul Menage. TCMalloc: Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[58] James R Goodman. Using cache memory to reduce processor-memory traffic. In *ACM SIGARCH Computer Architecture News*, volume 11, pages 124–131. ACM, 1983.

[59] Cary Gray and David Cheriton. *Leases: An efficient fault-tolerant mechanism for distributed file cache consistency*, volume 23. ACM, 1989.

[60] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Modelling in data base management systems. chapter Granularity of locks and degrees of consistency in a shared data base, pages 365–393. 1976.

[61] Jim Gray. *Concurrency Control and Recovery in Database Systems*, chapter Notes on data base operating systems, pages 393–481. Springer-Verlag, 1978.

[62] Jim Gray. The transaction concept: Virtues and limitations. In *VLDB*, pages 144–154, 1981.

[63] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. SIGMOD, pages 243–252, 1994.

[64] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[65] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[66] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *International Conference on Parallel Processing*. Citeseer, 1990.

[67] Mark Hachman. Intel finds specialized tsx enterprise bug on haswell, broadwell cpus. 2014.

[68] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[69] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, 10(5):553–564, January 2017.

[70] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[71] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[72] M. Heytens, S. Listgarten, M-A. Neimat, and K. Wilkinson. Smallbase: A main-memory dbms for high-performance applications. Technical report, Hewlett-Packard Laboratories, 1995.

[73] Henry Hoffmann, David Wentzlaff, and Anant Agarwal. Remote store programming. In *High Performance Embedded Architectures and Compilers*, pages 3–17. Springer, 2010.

[74] Chung-Wah Norris Ip, David L. Dill, and John C. Mitchell. State reduction methods for automatic formal verification, 1996.

[75] Ranjit Jhala and Kenneth L. McMillan. Microarchitecture verification by compositional model checking. In *CAV*, pages 396–410. Springer-Verlag, 2001.

[76] Ryan Johnson and Ippokratis Pandis. The bionic dbms is coming, but what will it look like? In *CIDR*, 2013.

[77] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: a scalable storage manager for the multicore era. EDBT, pages 24–35, 2009.

[78] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: a scalable approach to logging. *Proc. VLDB Endow.*, 3(1-2):681–692, 2010.

[79] Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark R. Tuttle, and Yuan Yu. Checking cache-coherence protocols with TLA$^+$. *Formal Methods in System Design*, 22(2):125–131, 2003.

[80] Paul F. Reynolds Jr., Craig Williams, and Raymond R. Wagner Jr. Isotach Networks. *IEEE Trans. Parallel Distrib. Syst.*, 8(4):337–348, 1997.

[81] Hyungsoo Jung, Hyuck Han, Alan D. Fekete, Gernot Heiser, and Heon Y. Yeom. A scalable lock manager for multicores. In *SIGMOD*, pages 73–84, 2013.

[82] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[83] Pete Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *USENIX Winter*, volume 1994, 1994.

[84] John H Kelm, Matthew R Johnson, Steven S Lumetta, and Sanjay J Patel. Waypoint: scaling coherence to thousand-core architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 99–110. ACM, 2010.

[85] Won Kim, Jorge F. Garza, Nat Ballou, and Darrell Woelk. Architecture of the orion next-generation database system. *IEEE Transactions on knowledge and Data Engineering*, 2(1):109–124, 1990.

[86] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[87] George Kurian. *Locality-aware Cache Hierarchy Management for Multicore Processors*. PhD thesis, Massachusetts Institute of Technology, 2014.

[88] George Kurian, Omer Khan, and Srinivas Devadas. The locality-aware adaptive cache coherence protocol. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 523–534. ACM, 2013.

[89] George Kurian, Jason Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel Kimerling, and Anant Agarwal. ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network. In *International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[90] Edya Ladan-Mozes and Charles E Leiserson. A consistency architecture for hierarchical shared caches. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 11–22. ACM, 2008.

[91] Kwok-Wa Lam, Kam-Yiu Lam, and Sheung-Lun Hung. Real-time optimistic concurrency control protocol with dynamic adjustment of serialization order. In *Real-Time Technology and Applications Symposium*, pages 174–179. IEEE, 1995.

[92] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[93] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[94] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[95] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *VLDB*, 5(4):298–309, December 2011.

[96] Juhnyoung Lee and Sang H Son. Using dynamic adjustment of serialization order for real-time database systems. In *Real-Time Systems Symposium*, pages 66–75. IEEE, 1993.

[97] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[98] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35. ACM, 2017.

[99] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, and Srinivas Devadas. Memory coherence in the age of multicores. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 1–8. IEEE, 2011.

[100] David Lomet, Alan Fekete, Rui Wang, and Peter Ward. Multi-version concurrency via timestamp range conflict management. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 714–725. IEEE, 2012.

[101] Yeong-Chang Maa, Dhiraj K Pradhan, and Dominique Thiebaut. Two economical directory schemes for large-scale cache coherent multiprocessors. *ACM SIGARCH Computer Architecture News*, 19(5):10, 1991.

[102] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo MK Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for power multiprocessors. In *Computer Aided Verification*, pages 495–512. Springer, 2012.

[103] Hatem A Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. *Proceedings of the VLDB Endowment*, 7(5):329–340, 2014.

[104] Milo MK Martin, Mark D Hill, and David A Wood. Token coherence: Decoupling performance and correctness. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 182–193. IEEE, 2003.

[105] Milo MK Martin, Daniel J Sorin, Anatassia Ailamaki, Alaa R Alameldeen, Ross M Dickson, Carl J Mauer, Kevin E Moore, Manoj Plakal, Mark D Hill, and David A Wood. Timestamp snooping: an approach for extending smps. *ACM SIGOPS Operating Systems Review*, 34(5):25–36, 2000.

[106] Friedemann Mattern et al. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Paral lel and Distributed Algorithms*, 1988.

[107] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *In CHARME 01: IFIP Working Conference on Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science 2144*, pages 179–195. Springer, 2001.

[108] K.L. McMillan. Verification of infinite state systems by compositional model checking. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 219–237. Springer Berlin Heidelberg, 1999.

[109] KL McMillan and James Schwalbe. Formal verification of the Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 111–134, 1992.

[110] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *International Symposium on High-Performance Computer Architecture*, 2010.

[111] David L Mills. Internet time synchronization: the network time protocol. *Communications, IEEE Transactions on*, 39(10):1482–1493, 1991.

[112] Sang Lyul Min and Jean-Loup Baer. A timestamp-based cache coherence scheme. 1989.

[113] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 17(1):94–162, 1992.

[114] Andreas Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 234–245. IEEE Computer Society, 2005.

[115] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, 2014.

[116] SK Nandy and Ranjani Narayan. An incessantly coherent cache scheme for shared memory multithreaded systems. Citeseer, 1994.

[117] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3:928–939, September 2010.

[118] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. PLP: Page Latch-free Shared-everything OLTP. *Proc. VLDB Endow.*, 4(10):610–621, July 2011.

[119] Seungjoon Park and David L. Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296. ACM Press, 1996.

[120] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.

[121] Manoj Plakal, Daniel J Sorin, Anne E Condon, and Mark D Hill. Lamport clocks: verifying a directory cache-coherence protocol. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 67–76. ACM, 1998.

[122] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pinar Tözün, and Anastasia Ailamaki. OLTP on Hardware Islands. *Proc. VLDB Endow.*, 5:1447–1458, July 2012.

[123] Dan RK Ports, Austin T Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. 2010.

[124] Kun Ren, Alexander Thomson, and Daniel J. Abadi. Lightweight locking for main memory database systems. In *VLDB*, pages 145–156, 2013.

[125] Alberto Ros and Stefanos Kaxiras. Complexity-effective multicore coherence. In *Parallel Architectures and Compilation Techniques (PACT), 2012 21st International Conference on*, pages 241–251. IEEE, 2012.

[126] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *SOSP*, pages 285–298, 1995.

[127] Daniel Sanchez and Christos Kozyrakis. Scd: A scalable coherence directory with flexible sharer set encoding. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[128] David Seal. *ARM architecture reference manual.* Pearson Education, 2001.

[129] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[130] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, et al. F1: A distributed sql database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013.

[131] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. Cache coherence for gpu architectures. pages 578–590, 2013.

[132] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[133] Ulrich Stern and David L Dill. Automatic verification of the SCI cache coherence protocol. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 1995.

[134] Tim Stitt. *An introduction to the Partitioned Global Address Space (PGAS) programming model.* Connexions, Rice University, 2009.

[135] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[136] Karin Strauss, Xiaowei Shen, and Josep Torrellas. Uncorq: Unconstrained snoop request delivery in embedded-ring multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 327–342. IEEE Computer Society, 2007.

[137] CK Tang. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 749–753. ACM, 1976.

[138] Dixin Tang, Hao Jiang, and Aaron J Elmore. Adaptive concurrency control: Despite the looking glass, one concurrency control does not fit all. In *CIDR*, 2017.

[139] Shreekant S. Thakkar and Mark Sweiger. Performance of an OLTP application on symmetry multiprocessor system. In *ISCA*, pages 228–238, 1990.

[140] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0), June 2007.

[141] Robert H Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, 1979.

[142] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012.

[143] Pinar Tözün, Brian Gold, and Anastasia Ailamaki. OLTP in wonderland: where do cache misses come from in major OLTP components? In *DaMoN*, 2013.

[144] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[145] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular deductive verification of multiprocessor hardware designs. In *27th International Conference on Computer Aided Verification*, 2015. Accepted paper.

[146] Deborah Anne Wallach. *PHD–a hierarchical cache coherent protocol.* PhD thesis, Massachusetts Institute of Technology, 1992.

[147] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, 2014.

[148] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. The serial safety net: Efficient concurrency control on modern hardware. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, page 8. ACM, 2015.

[149] David L Weaver and Tom Germond. The sparc architecture manual. 1994.

[150] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*, 2015.

[151] Arthur Whitney, Dennis Shasha, and Stevan Apter. High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C++. In *HPTS'97*.

[152] Craig Williams, Paul F. Reynolds, and Bronis R. de Supinski. Delta Coherence Protocols. *IEEE Concurrency*, 8(3):23–29, July 2000.

[153] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[154] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: the architecture and design of a database processing unit. In *ASPLOS*, 2014.

[155] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10:781–792, March 2017.

[156] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32. ACM, 2007.

[157] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. volume 8, pages 209–220. VLDB Endowment, 2014.

[158] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of SIGMOD 2016*, 2016.

[159] Meng Zhang, Jesse D. Bingham, John Erickson, and Daniel J. Sorin. Pvcoherence: Designing flat coherence protocols for scalable verification. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 392–403. IEEE Computer Society, 2014.

[160] Meng Zhang, Alvin R. Lebeck, and Daniel J. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 471–482, Washington, DC, USA, 2010. IEEE Computer Society.

[161] Hongzhou Zhao, Arrvindh Shriraman, and Sandhya Dwarkadas. Space: Sharing pattern-based directory coherence for multicore scalability. In *Parallel Architectures and Compilation Techniques (PACT), 2010 19th International Conference on*, pages 135–146. IEEE, 2010.

[162] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 465–477. USENIX Association, 2014.