SPECULATION-BASED TECHNIQUES FOR TRANSACTIONAL LOCK-FREE EXECUTION OF LOCK-BASED PROGRAMS

by

Ravi Rajwar

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

2002

Abstract

This thesis is motivated by the difficulty in writing correct high-performance programs. Writing shared-memory multithreaded programs imposes a complex trade-off between programming ease and performance, largely due to subtleties in coordinating access to shared data. To ensure correctness programmers often rely on conservative locking at the expense of performance. The resulting serialization of threads is a performance bottleneck. Locks also interact poorly with thread scheduling and faults, resulting in poor system performance.

We seek to improve multithreaded programming trade-offs by providing architectural support for optimistic lock-free execution. In a lock-free execution, shared objects are never locked when accessed by various threads. We propose two hardware techniques: Speculative Lock Elision and Transactional Lock Removal.

Speculative Lock Elision (SLE) is a micro-architectural technique to remove dynamically unnecessary lock-induced serialization and enable highly concurrent multithreaded execution. The key insight is that locks do not always have to be acquired for a correct execution. Synchronization instructions are predicted as being unnecessary and elided. This allows multiple threads to concurrently execute critical sections protected by the same lock. Misspeculation due to inter-thread data conflicts is detected using existing cache mechanisms and rollback is used for recovery. Successful elision is validated and committed without acquiring the lock and non-conflicting critical sections execute and commit concurrently without any serialization on the lock. SLE can be implemented entirely in the microarchitecture without instruction set support and without system-level modifications, is transparent to programmers, and requires only trivial additional hardware support.

Transactional Lock Removal (TLR) uses SLE as an enabling mechanism but in addition provides a successful lock-free execution of lock-based critical sections in the presence of data conflicts if sufficient resources are available for buffering speculative state. TLR elides locks using SLE to construct an optimistic lock-free critical section execution but in addition also uses a timestamp-based conflict resolution scheme to provide lock-free execution even in the presence of data conflicts. By treating the lock-free critical section as a lock-free transaction, TLR provides transactional properties for critical sections and by using timestamps for conflict resolution, TLR provides starvation freedom.

The benefits of SLE and TLR include improved programmability, stability, and performance. Programmers can obtain benefits of lock-free data structures, such as non-blocking behavior and wait freedom, while using lock-protected critical sections for writing programs.

Acknowledgments

I would like to thank my parents, Sushila and G.C.S. Rajwar, and my sister, Dr. Ritu Rajwar, for their constant support, inspiration, and encouragement throughout my education. I would like to especially thank Nathalie Le Coutour for having shared with me in the ups and downs of the graduate program and life in general and I am still amazed she has tolerated my idiosyncrasies. I owe much to her.

My advisor Prof. Jim Goodman taught me about research, thinking broad, and doing the right thing. I thank him for having supported me, both financially and academically, over the past years. I have learnt much from him. I have been lucky to have had the opportunity to interact with Prof. Jim Smith. His approach to research and life has influenced me quite a bit. Prof. Mark Hill and Prof. David Wood played an important role in my graduate career. I thank them for having been supportive of my research and having taken a keen interest in my work. I would like to thank Mark for having given valuable advice on research and non-research issues and for repeatedly emphasizing the importance of good presentation. I thank Prof. Guri Sohi, Prof. Mikko Lipasti, and Prof. Ras Bodik for having provided feedback on the work in this thesis, and valuable advice on various academic and non-academic issues over the past many years. I would also like to thank Mikko for having been there to listen to me while Jim Goodman was away on sabbatical. My thesis committee members David Wood, Guri Sohi, Jim Goodman, Jim Smith, Mark Hill, Mikko Lipasti, and Ras Bodik contributed substantially to improving the quality of this thesis.

Prof. Maurice Herlihy was patient to answer numerous questions and gave valuable feedback on the work in this thesis. I thank him for his support. I would also like to thank Dr. Joel Emer for various discussions on the topics in this thesis. I would especially like to thank Prof. Kewal Saluja for his constant support and sage advice over the years. A summer at Cray Research was spent working with Dr. Steve Scott. I learnt quite a bit about multiprocessors during that summer and I thank Steve for having played an important role in the learning process.

I have been lucky to have known Eric Rotenberg. He has set an example of being an excellent researcher and a great friend and was always ready to discuss ideas. Alain Kägi, Doug Burger, and Stefanos Kaxiras were very supportive during my initial years in graduate school. Scott Breach was an excellent source of advice when it came to writing complex simulators. Timothy Heil contributed extensively to the SimpleMP simulation infrastructure and I learnt a great deal from him about software engineering and Java memory models.

Craig Zilles, Dan Sorin, Manoj Plakal, and Milo Martin have shared in my graduate school experience the entire time I have been at Wisconsin and have served as excellent sources of advice, criticism, and knowledge. Manoj Plakal has been more than a colleague; he has been an excellent friend. Our discussions over wide-ranging topics were a welcome break from work. Dan Sorin was always around to hit a few balls on the tennis court when the going got tough. Brian Fields was ready to give comments on my paper drafts at all odd hours. I learnt much from our extensive discussions on various issues, including computer architecture. Adam Butts, Amir Roth, Andy Glew, Shai Rubin, and Trey Cain have served as great architecture listening boards and I have learnt much from them. Subbu Sastry never ceased to amaze me with his ability to balance research with time for issues of social good. My officemate, Paramjit Oberoi, put up with me for the past four years and he deserves due credit for his patience and tolerance. I have had many engaging discussions with Alaa Alameldeen, Alexey Loginov, Collin McCurdy, Harit Modi, Jason Cantin, Min Xu, Pacia Harper, and Vic Zandy and I am thankful to them for having indulged me.

I have met and interacted with many people during my stay in Madison. I would like to thank Etienne Kuntzel, Marie-Odile Souhaite, and Nita Sahai for being good friends. Robert Zimmerman’s work has been a constant companion during the long hours of coding and contemplation and I thank him. I would like to thank the staff at Muddy Waters for the 175 degree mocha made to perfection.

The CSL provides an excellent quality of computing service and the Condor staff provides an excellent quantity of computing service. I thank them for having effectively and promptly addressed any problems I may have had with the computing environment. Finally, I would like to thank the secretarial staff at the computer sciences department for having taken care of all the necessary administrative procedures required during the graduate program.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Processor systems are increasingly providing explicit support for multithreaded software either in the form of low-cost multiprocessors or hardware multithreaded architectures. Traditionally, processor systems have focused on improving single-thread performance and the cost of adding hardware resources for additional hardware threads was prohibitive. However, increasing transistor densities and transistor counts on chips today and improved semiconductor technology have led to a distinct trend towards increased user-visible hardware parallelism [30, 35, 82, 131, 159, 162]. In addition to improving single-thread performance by extracting implicit instruction-level parallelism, processor systems are providing explicit support for user-visible thread-level parallelism in hardware. Software writers now have, for the first time in computing history, easily available and low-cost hardware threads to exploit for performance and functionality. Programmers can be expected to take advantage of such hardware advances by writing multithreaded software.

While hardware systems have improved dramatically in terms of performance and functionality, the complexity of software systems has also risen owing to their increased functionality. For example, the concept of a web browser as we know it today did not exist a decade ago. Software aspects of reliability, stability, and portability are becoming increasingly important along with performance.

Writing correct, high-performance, and stable code is a complex and difficult task. The cost of developing such programs is increasing rapidly as their complexity and use increases. This cost includes both the one-time development cost and the recurring costs of maintenance, performance tuning, porting, and debugging.

While writing correct and high-performance single-thread code is difficult, writing multithreaded software is more so. Programming complexity is the single most significant problem in writing multithreaded applications [27, 69]. Although the use of threads simplifies the conceptual design of programs, care and expertise is required to coordinate correct interaction among various threads. This expertise is higher than for most single thread programs because coordinating sharing of data objects among various threads requires complex reasoning—actions of a thread can influence the subsequent behavior of other threads. Synchronization mechanisms are used to correctly coordinate thread accesses to shared objects. These mechanisms often enforce some form of serialization while threads access shared objects to ensure a consistent view of the object. Conservative use of such mechanisms aid in writing correct programs but unfortunately, these mechanisms unnecessarily enforce serialization of thread execution and degrade performance.

Writing correct, high-performance, and stable multithreaded programs thus entails a careful trade-off among various aspects of the program. These aspects include the ease of writing a correct program, its performance, and its behavior under unexpected conditions.

Before we discuss the above aspects, we briefly discuss *transactions* and *critical sections* as popular mechanisms for coordinating concurrent access to shared data.

1.1 Transactions and critical sections

Transactions serve as an intuitive model for writing multithreaded programs. A *transaction* [39] comprises a series of read and write operations that provide the following properties: failure-atomicity, consistency, and durability. *Failure-atomicity* states a transaction must either execute to completion, or in the presence of failures, must appear not to have executed at all. Failure-atomicity provides an all-or-nothing property of execution and guarantees a data structure remains in a consistent state, even in the presence of failures. *Consistency* requires the transaction to follow a protocol that provides threads with a consistent view of the data object. Serializability is an intuitive and popular consistency criterion for transactions. *Serializability* requires the result of executions of concurrent transactions to be *as if* there were some global order in which these transactions had executed serially [39]. *Durability* states that once a transaction is committed, its effects cannot be undone.

Serializability is similar to sequential consistency with regard to memory operations. Lamport [98] defined an execution to be *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. Similarly, an execution of transactions is considered *serializable* if it appears as if all transactions were executed in some sequential (serial) order with no interleaving within transaction boundaries.

While the concept of transactions is simple and convenient for programmers to reason with [57], processors today provide only restricted support for such transactions in their instruction sets. Examples are the atomic read-modify-write operations on a single word. The restricted size for these operations and limitations placed on their use render them ineffective in providing functionality of general transactions.

A lack of general transaction support in processors has led to programmers often relying on critical sections to achieve some of the functionality of transactions. *Critical sections* are software constructs used by programmers to enforce mutually exclusive access among threads to shared objects—only one thread is allowed to operate on the object at any given time—and thus trivially satisfy serializability. Providing failure-atomicity with critical sections is difficult since it requires support for logging modifications performed by the transaction, and then making these changes visible instantaneously using an atomic operation to commit the transaction. Critical sections therefore do not provide failure-atomicity. Critical sections are most commonly implemented using a software construct known as a *lock*. A lock is associated with a shared object and determines whether the shared object is currently available. Nearly all architectures support instructions for implementing lock operations. Locks have become the synchronization mechanism of choice for programmers and are extensively used in various software such as operating systems, database servers, and web servers.

Critical sections provide an intuitive interface for reasoning about data sharing because they trivially satisfy serializability. Today, critical sections are arguably the most popular abstraction for reasoning about correctness and coordinating sharing in multithreaded programs.

1.2 Multithreaded program aspects

Three important aspects of multithreaded programs are: 1) ease of writing a correct program, 2) performance of the program, and 3) stability of the program.

1.2.1 Programmability

Programmability, i.e., the ability to write a correct program easily, is perhaps the determining factor for wide-spread use of multithreaded programs [27]. Two common approaches to writing correct multithreaded programs easily are: a) conservative synchronization, and b) coarse-grain locking.

To ensure correct interaction of multiple threads while accessing data objects, programmers can conservatively use a lock to protect the object even though at run time the interactions would not cause an incorrect execution for that particular instance. We call this *conservative synchronization.* When multiple locks are used in the program, they must be managed with care or else deadlock and other related issues arise. A solution to this problem involves minimizing the number of locks in the program. Thus, if a thread potentially accesses multiple data objects, one lock is used to protect these objects. This is *coarse-grain locking*—a lock protects a large set of shared data, even though only a small part of it may be accessed at any time.

While conservative synchronization and coarse-grain locking assist in writing correct and reliable programs quickly, they limit concurrency of the program—execution of threads accessing disjoint data sets protected by the same lock are unnecessarily serialized. Furthermore, lock contention may limit scalability. *Lock contention* occurs when a requested lock is currently held by another thread. Lock contention may not be a problem for a two processor system, but may become a severe bottleneck for an eight processor system. Often, improving scalability requires an expensive process of code restructuring, debugging, and performance tuning. An example is the *linux* kernel [13]. The initial versions used a single global lock protecting all shared data in the kernel. Uniprocessor versions were not affected by this but as linux became more widely used on larger systems, severe scalability issues were encountered.

1.2.2 Performance

To extract high performance, synchronization use must be optimized. To improve the scalability of the linux kernel, fine-grain locking was employed. In contrast to coarse-grain locks where one lock protects a large data set, a fine-grain lock protects a smaller (finer granularity) data set. Efforts in breaking down locks and employing fine-grain synchronization has led to numerous subtle synchronization errors that are hard to find and debug. Another example where fine-grain locking may actually be detrimental is a thread-safe hash-table. Given a good hashing function, bucket conflicts between threads are rare but must be avoided for correct operation. Using a single lock makes programming easy because all access to hash tables is guaranteed to occur consistently. Obviously, this limits concurrency because threads accessing different buckets in the hash table are unnecessarily serialized. However, adding fine-grain locks is not a simple solution. Adding locks to each bucket may increase the footprint of the hash table itself resulting in poor memory system behavior and this can be critical for very large hash tables. Further, if multiple buckets are accessed, managing multiple locks gets tricky resulting in a higher probability of deadlocks and broken code. If the hash table needs to be rehashed, all locks must be acquired with care to avoid deadlocks and doing so is not an easy task.

Programmability and performance thus appear to be an either-or proposition. While conservative locking methodologies help in writing correct and reliable code, they severely limit performance. Extracting performance requires the error prone and difficult task of fine-tuning the locking methodology.

1.2.3 Stability

We define stability as the behavior of the program under unexpected conditions. If a thread is descheduled by the operating system or a thread terminates due to software or hardware errors, the application must be stable enough to allow other non-faulty threads to proceed. These non-faulty threads can take corrective action if necessary and the system does not crash or encounter arbitrary long delays. Using lock-based critical sections makes it exceedingly difficult to provide stability. The difficulty arises from the notion of a programmer-specified wait while some thread is in the critical section. A lock marked held forces other threads to wait for the lock to become free. If a thread is descheduled while holding a lock, other threads waiting for the lock cannot proceed because the lock is not free. This results in convoying. Convoying occurs when a convoy of waiting threads is formed and the lock for which these threads are waiting will not be available for a while.This may result in a critical problem of priority inversion if one is not careful [104]. Further, if a thread terminates due to an error while holding a lock, other threads waiting for the lock never complete as the lock is never free again. The problem is catastrophic in a transaction oriented environment where threads are largely independent except while accessing some critical shared structures. Data modified within the critical section are left in an inconsistent state resulting in application failure. This occurs primarily because critical sections typically do not provide failure-atomicity.

1.2.4 Limitations and solutions

In summary, using locks for coordinating access to shared data structures results in the following key limitations:

1. *Lock acquisitions limit performance*. Memory system behavior for lock variables is often poor because multiple processors read and attempt to write the variable simultaneously but only one processor succeeds in the write operation. The coherence protocol serializes write accesses to lock variables and transferring write permissions among various processors using the cache coherence protocol is expensive and introduces long latencies. Locks also serialize execution of threads even if the threads access disjoint data sets.
2. *Lock acquisitions limit stability.* Threads wait for the lock value to change from a held state to a free state and this “wait” is a key reason for the lack of non-blocking and wait-free behaviors of conventional lock implementations.

Current proposals for addressing the trade-off among various aspects can broadly be divided in two categories: lock-based and lock-free mechanisms.

The lock-based mechanism supporters believe lock-based critical sections are here to stay because they are easy to use and supported widely. Thus, the performance of lock primitives must be improved to provide fast and efficient coordination among threads. The programmer can and will decide how and when to use synchronization and has to reason about program correctness. Often programmers must optimize synchronization by hand to achieve high performance. This results in a complex trade-off between programmability and performance and does not address stability issues. Most work to-date has focused on overlapping computation with communication latency of the lock and many modern processors now support this in a limited form by employing speculative execution.

The lock-free mechanisms use special data structures to address inherent limitations of locks and attempt to address the trade-off between programmability and performance. Lock-free schemes often optimistically provide concurrent data structure implementations without a critical section or a software wait on a lock variable. While these techniques help address the stability aspect of programs, they often require more complex operations than critical sections. Programmers still have to reason about correctness in the presence of complex data structures and formal proof techniques are often required for correct implementations in software. These techniques often also suffer from high software overhead and thus unfortunately aggravate the complexity/ performance trade-off. Lack of portability and generality, and often poor performance with respect to lock-based schemes have resulted in lock-free approaches not being popular.

1.3 Problem statement

In summary, while lock-based critical sections are easy to use for writing correct and portable code, lock-free approaches overcome the inherent limitations of locks by avoiding using locks. Lock-based critical sections often outperform lock-free implementations but require careful programming methodologies for high performance. The three aspects—programmability, performance, and stability—appear to be at odds; with only one, or at-most two, being satisfied by current methodologies.

The natural question then is this: *is it possible to maintain a lock-based critical section as the programming model while transparently obtaining the benefits of lock-free approaches and achieving high-performance while doing so?*

This dissertation answers the above question in the affirmative and for the first time shows how one can maintain a lock-based critical section as the programming model of choice and use modest hardware support for automatically achieving behavior of lock-free data structures with high performance. The dissertation demonstrates how to address the multithreaded program trade-offs dynamically and transparently.

1.4 Contributions

This dissertation provides the first solution that bridges the long-standing gap between writing correct and stable multithreaded code and writing high-performance multithreaded code. The underlying philosophy behind the solution lies in maintaining the current programming model while transparently transforming the model in hardware to a concurrent one.

1.4.1 Primary contributions

The thesis makes two primary contributions:

1. **Speculative Lock Elision**. Speculative Lock Elision (SLE) [139] for the first time demonstrates that it is possible to execute and commit concurrently, critical sections protected by the same lock without acquiring (or requiring exclusive permissions on) the lock if the critical section executions do not experience any data conflict. SLE is a microarchitectural technique to remove dynamically unnecessary lock-induced serialization of threads. Synchronization instructions are dynamically predicted as being unnecessary for a particular dynamic execution and elided. This allows multiple threads to concurrently execute critical sections protected by the same lock and without any dependence on the lock. Misspeculation due to inter-thread data conflicts is detected using existing cache coherence mechanisms and a rollback mechanism is used for recovery. Successful elision is validated and committed without acquiring the lock. SLE can be implemented entirely in the microarchitecture without instruction set support and without system-level modifications. It is transparent to programmers and requires modest additional hardware support. SLE thus provides the mechanism to extract a lock-free execution from a lock-based execution. However, SLE provides a lock-free execution only if data conflicts do not occur. In the presence of data conflicts, SLE may require a lock acquisition. Since lock-based critical sections do not provide failure-atomicity, SLE cannot provide full transactional semantics (serializability and failure atomicity) in the presence of data conflicts.

2. **Transactional Lock Removal**. Transactional Lock Removal [140] uses SLE as an enabling mechanism but in addition provides a successful lock-free execution even in the presence of data conflicts if sufficient resources are available for buffering speculative state. TLR treats critical sections as optimistic lock-free transactions. TLR elides locks using SLE to construct an optimistic lock-free transaction but in addition also uses a timestamp-based conflict resolution scheme to provide lock-free execution even in the presence of data conflicts. A single, globally unique, timestamp is assigned to all memory requests generated for data within the optimistic lock-free transaction. Existing cache coherence protocols are used to detect data conflicts. On a conflict, some threads may restart (employing hardware misspeculation recovery mechanisms) but the same timestamp determined at the beginning of the optimistic lock-free transaction is used for subsequent re-executions until the transaction is successfully executed. A timestamp

update occurs only after a successful execution. Doing so guarantees each thread will eventually win any conflict by virtue of having the earliest timestamp in the system and thus will succeed in executing its optimistic lock-free transaction. If the speculative data can be locally buffered, all non-conflicting transactions proceed and complete concurrently without serialization or dependence on the lock. Transactions experiencing data conflicts are ordered without interfering with non-conflicting transactions and without lock acquisitions.

Both SLE and TLR avoid writing to the lock variables. By not writing to the lock variables, both, the overhead of lock acquires and the wait for lock variables to change value, are eliminated. We show in this thesis that TLR performs better than common lock-based algorithms even in the presence of high lock contention and data conflicts.

Since SLE and TLR use hardware resources, they are applicable only if the data set accessed within the critical section can be locally cached. Other implementation specific constraints may exist and are discussed in later chapters. Importantly, if any situation arises where SLE and TLR cannot be applied, the lock can always be acquired normally and a correct execution is guaranteed. In these cases, complete transactional semantics and starvation freedom cannot be provided because the execution falls back on the conventional lock acquire sequence.

Our techniques are different from earlier approaches in two significant ways.

1. Rather than change the programming model to obtain transactional semantics, we change the hardware implementation to transparently provide such semantics. A transaction is semantically stronger than a critical section since it also provides failure atomicity. A critical section is treated as a transaction and optimistically executed without lock operations. The intuition lies in treating locks as defining the scope of a transaction, using a conflict resolution scheme to order conflicting transactions correctly, and using a technique to give the appearance of an atomic commit of the transaction, such as is provided by SLE.

2. TLR uses a conflict resolution scheme to provide starvation freedom and thus can provide wait-free execution of critical sections subject only to potential resource limitations.

SLE and TLR maintain the programming interface of a familiar lock-protected critical section and thus programmers do not have to learn new methods to write programs. Additionally, existing legacy code using critical sections can directly benefit from our proposals. By treating critical sections as lock-free optimistic transactions, inherent concurrency in the transactions is exposed independent of lock granularity. By using a fair conflict resolution scheme and providing sufficient

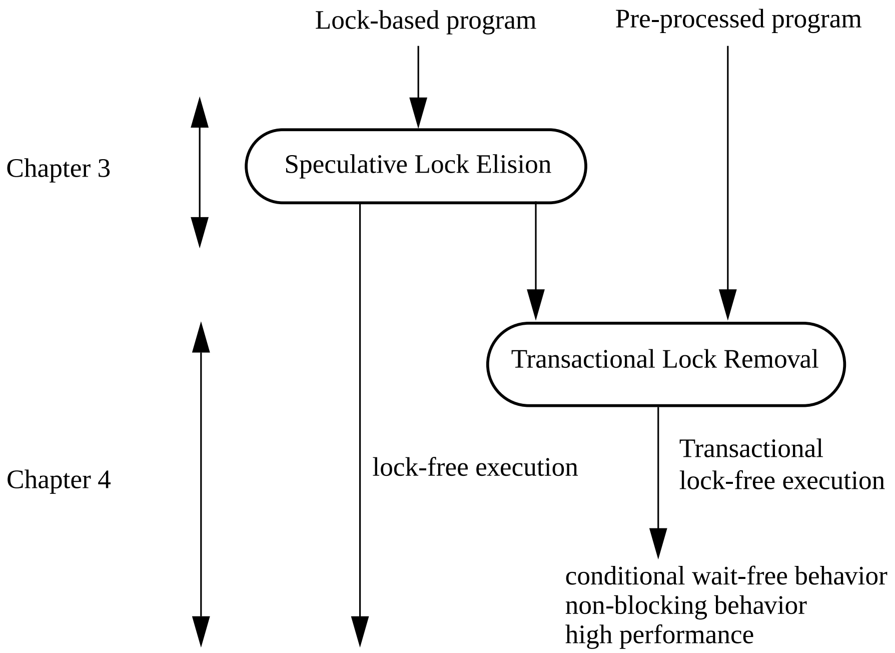


Figure 1-1: Solution overview. *Speculative Lock Elision (Chapter 3) and Transactional Lock Removal (Chapter 4) form the core contributions of the thesis. While we target lock-based programs, TLR can be used without SLE if the input program to TLR is already lock-free and the transactions are identified.*

resources for buffering speculative updates, we guarantee high-performance lock-free and wait-free executions of lock-based critical section.

Figure 1-1 shows how the two techniques fit together. Conceptually, SLE accepts as input a program that uses locks as a synchronization mechanism and transforms the execution dynamically and transparently into a lock-free execution. TLR takes the lock-free execution output of SLE, or may take a pre-processed output from some other source with transactions already identified, and provides a high-performance transactional lock-free execution.

1.4.2 Other contributions

Some of the other contributions are:

1. First technique to achieve transparent lock-free execution of lock-based programs. This shows how concurrency can be naturally exposed and exploited for improving performance.
2. First proposal for achieving wait-free execution of critical sections subject only to potential resource limitations. This can allow programmers to write high-performance wait-free algorithms using the basic mechanisms developed in this thesis as building blocks for wait-free synchronization.
3. A distributed lightweight deadlock avoidance protocol for concurrency control using a general cache coherence protocol.
4. Demonstrate a lock-free algorithm can outperform lock-based techniques using modest hardware support.

1.5 Evaluation

Two aspects form the evaluation.

1. Performance. We use detailed cycle-accurate architectural simulations and a set of microbenchmarks and benchmarks to study the performance of our proposals. We identify the conditions under which high performance is achieved and the conditions under which performance loss may occur.
2. Implementation overhead. We estimate the additional hardware required to implement our proposals. We do not provide a detailed implementation as it is highly dependent upon the underlying microarchitecture and system architecture. We however outline and discuss the key mechanisms to help estimate the implementation overhead in a system independent manner.

1.6 Organization

Chapter 2 provides background information and surveys relevant previous work. The chapter attempts to provide a common ground for understanding the thesis. It provides sufficient background for multiprocessor system issues such as memory consistency and cache coherence and discusses speculative execution techniques in modern processors. We also discuss some formal aspects of concurrency control, namely their safety and liveness properties. This discussion motivates our correctness arguments and the design of our techniques. Apart from background, related work is discussed. The discussion includes lock-based, lock-free, and wait-free synchronization techniques and some work in database concurrency control.

Chapter 3 describes Speculative Lock Elision. We provide the key insight for eliding locks and present the complete algorithm. We also present various implementation strategies for SLE.

Chapter 4 presents Transactional Lock Removal. TLR is analogous to database concurrency control. We identify certain implementation-independent invariants that allow TLR to be implemented on systems. We also discuss the limitations of TLR and the stability and programmability aspects of TLR, specifically non-blocking behavior, wait-free behavior, and operating systems interactions.

Chapter 5 and Chapter 6 focus on performance evaluation. Chapter 5 outlines the performance evaluation methodology, discusses the simulation infrastructure and presents the various microbenchmarks and benchmarks. Chapter 6 presents performance evaluation results for both SLE and TLR using microbenchmarks and benchmarks.

Chapter 7 summarizes the contributions of the dissertation and suggest future topics for research.

Chapter 2

Background

This chapter provides the background for understanding the thesis and presents related work. The chapter is divided into four main sections. Section 2.1 provides a background into shared-memory multiprocessing. Shared-memory systems are the target architecture for the work in this thesis. We focus on two important and relevant features of shared-memory systems: the memory consistency model and mechanisms for maintaining cache coherence.

Section 2.2 focuses on concurrency control mechanisms and synchronization techniques. We discuss techniques such as lock-based, lock-free, wait-free, and non-blocking synchronization. We do not provide a comparison between prior work and the techniques proposed in this thesis—the specific comparisons are left to the related work sections of each individual chapter.

Section 2.3 lays the foundations for reasoning about correctness of our techniques by discussing safety and liveness properties.

The techniques proposed in this thesis use speculative execution as an enabling mechanism and Section 2.4 presents a background on common speculative execution techniques in processors.

2.1 Shared-memory multiprocessing

Multiprocessor architectures are increasingly becoming a viable and cost-effective technology even for small processor counts. Most multiprocessor systems are differentiated based on the communication mechanism among different processors. Two popular classifications are Message-Passing Architectures and Shared-Memory Architectures. In message-passing systems, each processor has a local memory accessible only by that processor, and communication among various processors occurs through explicit messages. On the other hand, shared-memory systems make at least part of the memory accessible to all processors and thus allow processors to communicate directly through read and write operations to memory. This thesis is concerned with shared-memory multiprocessor architectures. A typical shared-memory system is shown in Figure 2-1.

Shared-memory architectures have emerged as the dominant class of systems today largely due to the relative ease of writing shared-memory parallel programs [26, 47, 72, 105, 115, 148] as compared to writing message-passing parallel programs.

Two important aspects of shared-memory systems related to the thesis are *memory consistency* and *cache coherence*. A memory consistency model is a conceptual model for semantics of memory operations that allow programmers to use shared memory correctly. Such a model specifies how memory behaves with respect to read and write operations from multiple processors. Cache coherence is one of the mechanisms required to implement a memory consistency model on systems that support caching of shared data at the processors. In the remainder of this section, we discuss memory consistency and cache coherence.

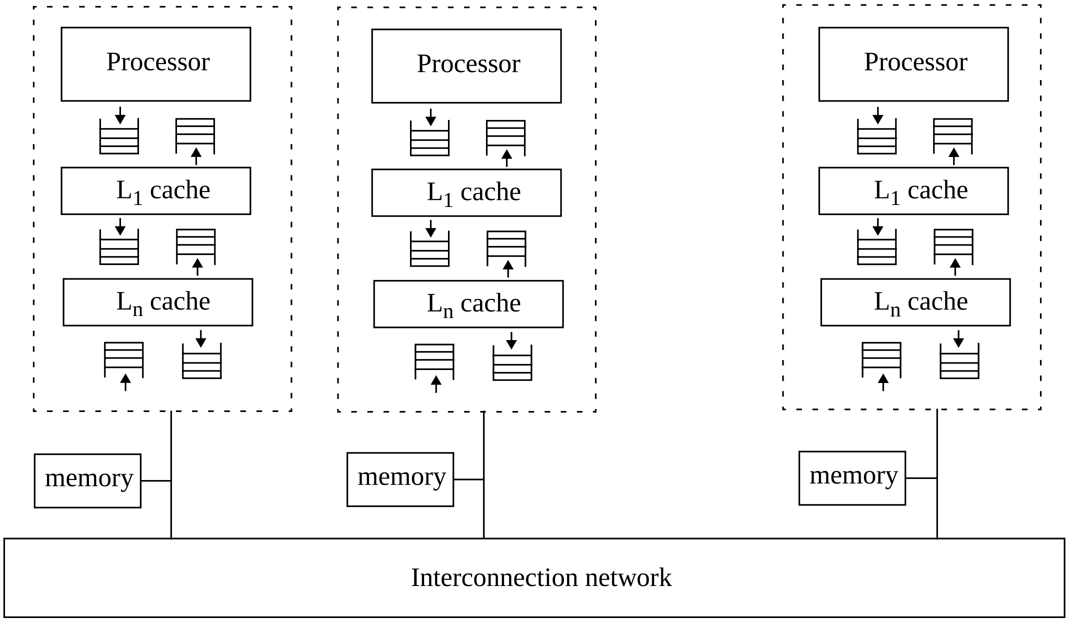


Figure 2-1: A typical shared-memory multiprocessor. *Each node consists of a processor, multiple levels of caches, L_1 through L_n, and inter-cache buffers. The interconnection network may be an ordered broadcast network or an unordered network. The cache coherence protocol implemented may be snoop-based or directory-based.*

2.1.1 Memory consistency

From the programmer's perspective, a memory model allows for correct reasoning about memory operations in a program, and from a system designer's perspective, the model specifies acceptable memory behavior for the system.

The program for each processor imposes a conceptual total order on the operations issued by the processor in a given execution. The *program order* is defined as a partial order on all memory operations that is consistent with the per-processor total order on memory operations defined for each processor. An operation is considered *atomic* if the operations appear to occur instantaneously with respect to all processors.

An intuitive memory model is based on the sequential semantics of memory operations in uniprocessors and viewing a multiprocessor as a multiprogrammed uniprocessor. Lamport formally defined such a model as *sequential consistency* [98]:

> "...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

Sequential consistency provides an interface that most people expect [69]. Figure 2-2 shows a conceptual representation of sequential consistency. Memory is shared among multiple processors. Each processor issues its memory operation in program order. Operations are serviced by memory one-at-a-time; thus they appear to occur atomically with respect to other memory operations. The order of servicing of operations from different processors may be arbitrary thus leading to an arbitrary ordering of memory operations from different processors into a single sequential order. An execution of a program is sequentially consistent if there exists at least one execution on a sequentially consistent system that provides the same result.

Architectural and compiler innovations, such as write buffers and caching, have introduced complexity in supporting sequential consistency as a model of choice on shared-memory multiprocessors. These have led to extensive work in specifying, defining, and implementing various memory consistency models on modern shared-memory multiprocessors [2, 3, 4, 44, 45, 46, 133, 147].

Relaxed memory models have been proposed to enable the use of more optimizations by relaxing the limitations on the ordering of memory operations as imposed by strict memory mod-

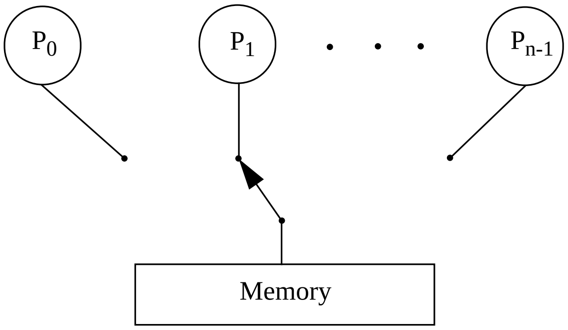


Figure 2-2: Conceptual view of sequential consistency. *Each processor interacts with shared memory through a single switch in a one-at-a-time fashion.*

els such as sequential consistency. While sequential consistency requires the illusion of program order and atomicity to be maintained for all operations, relaxed models allow certain memory operations to execute out of program order and/or non-atomically.

Categorization of various relaxed memory consistency models is based on two characteristics [44]:

1. *How the program order requirement is relaxed*. This may involve the relaxation of the order of a write to a following read, a read to a following read or write, and between two writes. Here, the relaxation only applies to operation pairs to different addresses.

2. *How the write atomicity requirement is relaxed*. This is based on whether a read is allowed to return the value of *another* processor's write before the write is made visible to all other processors. Here, the relaxation applies to operation pairs to the same address.

Since relaxed models allow reordering of memory operations, programmers are provided with explicit mechanisms to prevent such reordering from occurring if so desired. Such mechanisms are generically referred to as *safety nets* for a model [3]. Examples of such safety nets include read-modify-write operations in TSO and PC [70, 166], MEMBARs in the ALPHA architecture [28] and SYNC in the PowerPC [31]. In all cases, there is a single point (at least one) where all preceding operations have been completed and no succeeding operations have been executed.

A tutorial on shared-memory models by Adve and Gharachorloo [3] and their respective theses [2, 44] provide detailed background into various memory consistency models.

2.1.2 Cache coherence protocols

Caching is a popular technique to reduce long latency memory operations and to reduce memory bandwidth requirements—processors access the local cache for a fast path to commonly accessed data. Caching shared data in shared-memory multiprocessor systems results in multiple copies in the system for a given memory location. Cache coherence is the mechanism to keep all such copies up-to-date with respect to one another. Nearly all shared-memory systems today support some form of cache coherence. While memory consistency models issues exist even in the absence of caches, cache coherence is a central component for correctly implementing memory consistency models on multiprocessor systems that cache shared data. This section provides a background into cache coherence protocols.

2.1.2.1 Aspects of cache coherence protocols

Two mechanisms of any cache coherence protocol are:

1. Mechanism for locating all cached copies of a memory location

2. Mechanism for keeping all cached copies of a memory location up-to-date

Two common schemes for locating all copies of a memory location are: *snoop-based* and *directory-based* schemes. In a snoop-based scheme, the address of the memory location is broadcast to all caches. In a directory-based scheme, a directory per memory location is maintained that identifies the list of copies. Snoop-based schemes are more popular than directory-based schemes in commercial implementations. Two popular approaches distribute directories either with memory or with caches. Memory-based schemes store directory information for a cache block at the home node of the block. Examples of memory-based directory protocol systems include the SGI Origin 2000 [105]. In cache-based schemes, most of the sharing information is distributed among the various copies (rather than at the home node). Each cache block contains a pointer to the node that has the next cached copy of the block in a distributed linked-list organization. The home node still needs to know if the memory block is cached and where one copy is. The IEEE 1596-1992 Scalable Coherent Interface (SCI) standard indicates a full specification and C code for a standardized cache-based directory organization and protocol [72]. Commercial implementations of the SCI include the Sequent NUMAQ [115] and the Convex Exemplar X [21].

A write operation must ensure all cached copies are kept up-to-date. This can be accomplished by either invalidating any stale copies or updating the cached copies to the newly written value. Such an action is often accompanied by an acknowledgement response to signal completion of the action. Cache coherence protocols serialize the effect of simultaneous write operations to a given memory location. If two processors simultaneously issue a write to the same location with different values, cache coherence ensures the two writes are observed in the same order by all processors with the same value persisting at all copies.

In most invalidation-based cache coherence protocols, the cache hierarchy with the dirty copy (modified with respect to memory) of the cache block is responsible for servicing read requests from other processors for either shared copies or exclusive copies of the cache block since the data in memory is stale. A cache hierarchy that does not have the cache block in dirty state does not need to respond. For such caches, an incoming read request is simply ignored and an incoming read-for-exclusive-ownership request is treated as an invalidate request.

2.1.2.2 Coherence granularity and false sharing

Most systems today support cache coherence mechanisms in hardware [26, 29, 47, 59, 72, 82, 105, 115, 132] and typically maintain coherence at the granularity of a cache block, specifically, at the coherence granularity [49]. This thesis deals with hardware shared-memory cache coherence protocols and we only discuss hardware cache coherence here. Cache block fetches and invalidations are performed at the granularity of a cache block. While a larger granularity helps when good spatial locality in data accesses is present, poor spatial locality may result in a performance degradation due to *false sharing*. Goodman and Woest [51] were the first to define false sharing as a situation when two processors alternately read or write different parts of the same coherency block, resulting in the block's being moved repeatedly between the two processors as if the data were shared when in fact no sharing is occurring.

2.1.2.3 Correctness issues for cache coherence protocols

Gharachorloo [44] formalized three correctness conditions for implementing coherence protocols and we restate them here. The three conditions that must be satisfied by the cache coherence protocol for a correct implementation are:

1. *Termination condition for writes.* Every write issued by a processor is eventually completed with respect to all processors.

2. *Value condition for reads.* For read and write operations to the same address, a read operation by a processor P$_i$ returns a value that satisfies the following conditions: a) If there is a write operation issued by P$_i$ that has not yet completed with respect to P$_i$ before the read completes, then the value returned by the read must be from the last such write that has been issued by P$_i$; b) Otherwise, the value returned by the read must be from the latest write (from any processor) that has completed with respect to P$_i$ before the read completes; c) If there are no writes that satisfy either of the above two categories, then the read must return the initial value of the location.

3. *Coherence requirement for writes.* Writes to the same address complete in the same order with respect to every processor.

In later chapters we show how our proposals do not change these underlying mechanisms for cache coherence and thus maintain the correctness conditions for cache coherence protocols.

2.1.2.4 Cache coherence protocol mechanisms

Sweazey and Smith [160] proposed the *Modified*, *Owned*, *Exclusive*, *Shared*, and *Invalid* (MOESI) classification of cache coherence protocols based on the stable states of a cache block. A cache block in stable state has valid data and the block is not waiting for any state transition to occur. Many cache coherence protocols can be represented as a subset of the five-state MOESI protocols. The five states are defined as follows: Modified—the block is dirty (memory is stale) and exclusively owned by the cache; Owned—the block is dirty (memory is stale), the block is possibly shared among multiple caches, and this cache is responsible for ensuring memory is kept up-to-date; Exclusive—the block is clean (memory is up-to-date) and exclusively owned by the cache; Shared—the block is clean (memory is up-to-date) and the block is possibly shared among multiple caches; Invalid—the block is not present in the cache.

While cache coherence protocols are referred to by their stable states, implementing high-performance cache coherence protocols often requires additional states, also known as transient or pending states. This is because a non-zero time may exist between the request initiation and request completion phases of a memory operation during which other operations may be performed. A cache block makes a transition out of the stable state (one of the MOESI states) at the request initiation phase and makes another transition into a stable state at the end of the request completion phase (which may involve the completion of a data transfer). The cache block remains in a pending state between the two phases and may transition to multiple subsequent pending states depending upon the coherence events occurring. Hennessy and Patterson [62] provide an example demonstrating the complications introduced by the addition of pending states to a conventional cache coherence protocol.

We briefly discuss some common cache coherence protocol mechanisms. We focus on two common classes of multiprocessors: snoop-based and directory-based.

Snoop-based systems typically rely on a *logical bus* to propagate address requests to all processors simultaneously. A common design is a *split-transaction* bus where address requests that require a response (typically a data response) are split into two independent sub transactions—a request transaction and a response transaction. Buffering is used to allow multiple transactions to be outstanding on the bus waiting for responses from the controllers. While this adds complexity to the design, the bus is more effectively utilized.

The design space for such split-transaction logical buses is large. While conventional logical buses have been implemented as actual “physical buses”, modern systems adopt an aggressive approach. The logical bus is often organized as a complex interconnect and a logical ordering is emulated on the interconnect. Examples include commercial systems such as the Sun Gigaplane-XB [26].

Next we discuss two protocols: the Sun Gigaplane and the SGI Origin 2000. Since we employ the cache coherence protocol in this thesis to track shared data and track write operations to shared data, we focus on how write operations are serialized and how requests to exclusively-owned cache blocks are handled.

Sun Gigaplane. The Sun Gigaplane uses a split-transaction, pipelined address bus with support for a large number of outstanding transactions and out-of-order responses (the data responses return in any order irrespective of the order in which the address requests were generated and hardware is used to match up the requests appropriately). The bus implements an invalidation-based three-state (Owned[1], Shared, Invalid) snooping cache coherence protocol and the coherent caches implement the five-state MOESI protocol. The cache with the requested block in Owned state (as seen by the bus) will respond to the next request for that block.

An interesting aspect of the protocol is the way *interventions* are handled. An intervention occurs when a processor issues a request for a cache block that is currently in another processor's cache. Consider the case where processors issue read-for-exclusive-ownership (`rd_X`) requests in sequence. An `rd_X` request gets the cache block in an exclusively-owned state. Assume address block *A* resides in memory and is not cached by any processor. Assume processors P_0,..., P_{n-1}. P_0 issues a `rd_X` request for the block *A*. The request misses in the local cache, a pending buffer is allocated to record the request, and the request is sent to the memory system. When this request is serialized by the coherence protocol, i.e., all processors are *assumed* to have seen the request in a given order, P_0 gains exclusive ownership of the block. The memory, on observing the request on the coherency network, initiates a data transfer. Now, P_1 issues a `rd_X` request for the same block *A* and this request is serialized by the coherency network after P_0's request but before the data transfer for block *A* from memory to P_0's cache completes. Since P_0 owns the block *A*, P_0 receives P_1's request. However, P_0 does not have the corresponding data yet. P_0 buffers P_1's request locally. Now, since P_1's `rd_X` has been serialized, P_1 owns the block. However, P_0 itself does not have the data yet. Now, if P_2 also issues an `rd_X` for block *A*, P_1 will respond to P_2. Thus, a chain of requests is automatically formed according to the order of request-serialization in the coherency network. At some time later, P_0 receives the data block from memory, performs its operations on the block, and then services P_1's buffered request. P_1 does the same and so on—the actual write operations on the cache block lag the time at which the request is serialized at the coherency network and the requests are serviced in the form of a queue. This results in an efficient and fast cache-to-cache transfer when multiple writes appear on the coherency network for the same address.

We shall see in the next protocol discussion an alternative approach where rather than buffering requests for cache blocks in a valid state but currently without valid data (i.e., the block is in a

[1] The Owned state here is different from the Owned state of the MOESI classification discussed earlier. Here, the Owned state determines which processor responds with the data and subsumes the MOE states of the MOESI classification.

pending state), a negative acknowledgement is sent and the requestor is asked to retry at a later time.

SGI Origin 2000. The SGI Origin 2000 protocol supports the MESI (Modified, Exclusive, Shared, Invalid) states and is non-blocking: memory does not buffer requests while waiting for other messages to arrive. The protocol supports request forwarding for interventions. The protocol does not rely on an ordered network. Only two virtual channels are provided and deadlock in the request network (due to request forwarding) is broken by the use of backoff messages. Details of the SGI Origin 2000 protocol can be found elsewhere [34, 105].

In the protocol, memory is the owner for all clean cache blocks in the system: thus any request for clean data is immediately serviced by memory. In addition, for read-for-exclusive-ownership requests, ownership is transferred to the requestor and invalidates are sent to other cached copies. The cached copies subsequently send invalidate acknowledgements to the requestor.

Requests to a cache block not owned by memory are forwarded to the owner and in the case of a read-for-exclusive-ownership request, the requestor becomes the owner. The directory goes to a pending busy state for that cache block until a revision message is received from the previous owner. All requests received by the directory while in busy state are NACKed (a negative acknowledgement is sent to the requestor) and asked to retry at a later time. On receipt of a revision message, a transition to a stable state occurs. A processor receives only one intervention request for a given block at any time.

The SCI protocol [72] and newer directory protocols such as the Alpha server GS320 [47] support the non-nack-based approach similar to the Sun Gigaplane protocol discussed above and form chains of requestors.

2.2 Synchronization techniques and concurrency control

Concurrency control [16] is the activity of coordinating concurrent access to a shared object; i.e., of controlling the relative order of conflicting operations from different threads. *Synchronization technique* [16] is the algorithm to perform such concurrency control.

We start by discussing the classic mutual exclusion problem in Section 2.2.1. In Section 2.2.2 and Section 2.2.3 we discuss relevant research in the area of lock-based, lock-free, and wait-free synchronization. Much of this discussion focuses on shared-memory multiprocessors. Then in

Section 2.2.4 we discuss database concurrency control as we borrow some of its concepts in our work.

2.2.1 Mutual exclusion problem

The mutual exclusion problem is as follows. There is a collection of asynchronous processes, each alternately executing a critical and a non-critical section. These processes must be synchronized so that no two processes ever execute their critical sections concurrently. The mutual exclusion problem was first described and solved by Dijsktra [36].

Since mutual exclusion is an intuitive model for reasoning about concurrency, it is the most popular way to coordinate correct access to shared data and has been extensively studied over the years.

Almost all formal models of concurrent processing are based on the underlying assumption of mutually exclusive atomic operations. Lamport however demonstrated that atomic reads and writes can be implemented from non-atomic reads and writes without mutual exclusion [94, 100, 101].

Mutual exclusion has similarities to concurrency control techniques. Discussion regarding differences between mutual exclusion and concurrency control can be found elsewhere [25].

Lamport et al. recently introduced the notion of virtual mutual exclusion [103]. With virtual mutual exclusion, operations are executed in a way that makes it appear as if one critical section precedes another. If only memory accesses are performed during critical sections, then virtual mutual exclusion is sufficient to achieve mutually exclusive access. However, if I/O operations are also performed during the critical section, then it can be shown that true mutual exclusion is needed. Thus, in the absence of any I/O operations, critical sections may concurrently execute as long as they appear to execute as if one critical section precedes another.

2.2.2 Lock-based synchronization

Lock-based synchronization techniques rely on the use of a software variable, called a lock, to guard entry into a critical region of code, known as a critical section. The lock ensures only one thread is in the critical section at any time and access to shared data occurs within the critical section. Lock-based techniques typically imply mutual exclusion—no two processes ever execute their critical sections concurrently.[2]

A lock is a software construct associated with a shared object and determines whether the shared object is currently available. Once a process acquires a lock, no other process will be able to do so until the current lock holder releases the lock.[3] The process wishing to perform the critical section must acquire the corresponding lock first, which, once granted, guarantees its current owner that no other process will access the locations accessed in the critical section. When the lock owner completes its critical section, it releases the lock, allowing other processes to observe the updates performed by the now committed critical section.

2.2.2.1 Locking primitives

Lock-based synchronization has been extensively studied. The first synchronization primitive was test&set supported in the IBM System/360 series [7]. Test&set performs an atomic swap on a location in memory. Test&set performs well in the absence of contention but is quite inefficient under heavy load. Test&set can generate a large amount of traffic on the interconnection network. Rudolph and Segal proposed test&test&set [145] which reduces load on the network by having waiting processors spin on a local copy of the lock. This mechanism increases traffic for uncontended locks in exchange for reducing it when the locks are contended. The traffic can still be substantial in the presence of lock contention.

Queue-based locking primitives attempt to minimize the number of network transactions required to acquire and release a lock to a constant factor. These primitives maintain a queue of waiting processors in which each node typically maintains pointers to adjacent processors in the queue. Network traffic is minimized by performing arbitration for access to a critical section at the time of the lock request, by allowing processors to locally spin waiting for a lock, and by limiting the number of nodes involved in the actual lock transfer.

[2] Variations such as read locks have also been proposed and these are useful when one has multiple readers concurrently accessing the shared object.

[3] Possibly, a process other than the lock holder may also release the lock by simply writing the variable. Lock acquire and release operations are merely software conventions and thus must be used with care because they rely on the programmer's involvement for providing an execution free of data races.

Goodman et al. proposed the first queue-based locking primitive, known as Queue-On-Lock-Bit (QOLB) [50].[4] QOLB maintains the queue of waiting processors in hardware, storing pointers to adjacent queue entries in the fields associated with each cache block. When a processor requests a lock, it first allocates a cache block and then sends a request to join the queue. The processor waits for the lock by spinning locally until the cache block contains valid data sent from the previous lock owner. When the holder releases the lock, it sends the corresponding cache block directly to the next processor, thus transferring the lock in exactly one network message.

Following the QOLB proposal, the Stanford DASH prototype implemented a variant of the queue-based synchronization primitive [110]. Unlike QOLB, their proposal stored the queue at the directory rather than the caches. Doing so introduced an indirection in transferring locks—the lock can no longer be transferred directly from the releaser to the next waiter; instead the lock must go through the directory. Lee and Ramachandran proposed an extension to QOLB with support for read locks [107]. Recently, Rajwar et al. proposed Implicit QOLB (IQOLB) [141]. IQOLB is a hardware technique that transparently converts the test&set-based locking primitive to a hardware queue-based lock using modest extensions to the cache coherence protocol. Unlike QOLB, IQOLB does not require any instruction set support nor does it require any software changes.

Software queue-based locking schemes were proposed by Anderson [9, 10] and Graunke and Thakkar [55]. Mellor-Crummey and Scott proposed MCS, an improvement to Anderson’s algorithm. The MCS scheme [120, 121] is a software-based queued lock scheme. MCS adds requesters for a held lock into a software queue at the time of the request, using atomic operations such as swap and compare&swap to update the list. Arbitration for the eventual recipient of the lock is therefore performed in advance, first-come, first-served. Extensions to the above techniques have been proposed [33, 116].

Maintaining the requester queue in software has large overhead, especially in the absence of contention. When a lock is released, however, communication occurs only between the releaser and the requester at the head of the queue. Network traffic is thus reduced to a constant number of network traversals per synchronization access. In addition, each processor waiting for the lock spins locally on distinct memory addresses (instead of a single address as with test&test&set),

[4] QOLB was initially called QOSB (Queue-On-Synch-Bit).

which further reduces the load on the network. Each processor in the queue maintains a pointer to the address on which the next processor in the queue spins. When the current lock holder leaves the critical section, it simply clears the value pointed to by the address it maintains.

Lim and Agarwal proposed reactive synchronization, a technique that attempts to select the software primitive best suited for a given level of lock contention [113].

Woest and Goodman [168] present a quantitative and qualitative comparison of test&set, MCS, and QOLB. Kägi et al. [81] were the first to perform a comprehensive performance comparison of various popular synchronization algorithms. The study concluded that for the set of benchmarks used, QOLB consistently performed the best among known synchronization primitives.

In addition to the synchronization primitives, additional mechanisms have been proposed to reduce the overhead of synchronization operations. Software techniques in the form of collocation [19, 50] and fuzzy acquires [133] and hardware techniques in the form of speculative execution [45] have been proposed to overlap the transfer of lock and data.

2.2.2.2 Limitations of locking primitives

We discussed the limitations of locking primitives earlier in Section 1.2 and we summarize them again here. Lock-based synchronization techniques suffer from a lack of stability due to an inherent limitation of their conventional implementation. The limitation of the locking construct stems from the notion of the programmer-specified wait while some thread is in the critical section. A lock marked as held forces other threads to wait for the lock value to be free. This limitation manifests itself in two potentially catastrophic ways:

1. *Poor system wide interactions with thread scheduling.* If a thread holding a lock is descheduled by the operating system, other threads waiting for the lock cannot proceed because the lock is not free. In a high concurrency environment, all threads may wait until the descheduled thread runs again. This results in convoying (a convoy of waiting threads is formed) and may result in a deadlier problem of priority inversion (no thread may ever proceed). This is known as the blocking problem where one thread blocks other threads from running.

2. *Fault-tolerance limitations*. If a thread holding a lock terminates due to a fault, other threads waiting for the lock never complete as the lock is never free again. This problem is catastrophic in a transaction oriented environment where threads are largely independent except while

accessing some critical shared structures. Data modified within the critical section is left in an inconsistent state resulting in application failure.

Locking algorithms can be modified to address these limitations but require significant effort and creativity from the programmer. Software proposals have been made to make lock-based critical sections non-blocking [163] and thread scheduling that is aware of blocking locks [87, 123]. Bohannon et al. present a recovery strategy to allow a system to recover from a lock held by a process believed to have failed [20].

Locks also restrict parallelism: two operations on the same object cannot execute in parallel, even if they access disjoint parts of the object. In some cases, this problem can be addressed by using finer granularity locks. By using several locks per object, operations that access disjoint parts of the object can execute in parallel. Unfortunately, such an approach requires dynamic information about the operations at the time of writing the program and is error-prone because of complex reasoning required on the part of the programmer.

2.2.3 Lock-free and wait-free synchronization

Lock-free synchronization was proposed as an alternative to lock-based synchronization to overcome the limitations of lock-based techniques. Lock-free synchronization techniques coordinate correct access to shared resources without relying on mutual exclusion and thus access shared resources without employing a critical section. However, they rely on mechanisms other than locks to guarantee a correct execution.

Lock-free synchronization is a loosely used term referring to the absence of locking semantics. Two formally defined terms are non-blocking and wait-free synchronization and are most commonly used when referring to concurrent object operations.

A synchronization technique is *non-blocking* if some process will complete an operation in a finite number of steps, regardless of the relative execution speeds of the processes [64]. The non-blocking condition guarantees the system as a whole makes progress despite individual halting failures or delays.

A synchronization technique is *wait-free* if any process can complete any operation in a finite number of steps, regardless of the execution speeds of other processes [64]. Wait freedom adds starvation freedom, even in the presence of failures, to the non-blocking condition. The term *wait* *freedom* implies “not waiting forever.” It does not imply the processes never wait—processes may have to wait for a finite time—but they never wait for a process that has failed and aborted.

We briefly discuss key work in lock-free and wait-free techniques. As we shall see, these techniques suffer from difficulty of use and often poor performance.

2.2.3.1 Lock-free and wait-free techniques

Lamport introduced lock-free synchronization to allow multiple threads to work on a data structure concurrently without a lock [95]. In Lamport’s lock-free read/write buffer, the buffer consists of a sequence of digits which are read and written atomically. Only one writer is assumed, so there is no need to consider concurrent write operations. The writer may interfere with concurrent read operations. If a read operation is interfered with, then it must be retried. Repeated retries may be needed before a read can successfully complete. Two version numbers *V1* and *V2* are associated with the buffer. These version numbers allow readers to detect when interference has occurred. To perform a write operation, the writer increments *V1*, writes the buffer, and then increments *V2*. To perform a read operation, a reader reads *V2*, reads the buffer, and then reads *V1*. If the values read from *V1* and *V2* are identical, then a consistent value was read from the buffer. The order in which the sequence numbers and the buffer are read is the opposite of the order in which they were written. The version number themselves are multi-digit numbers with each digit written and read in opposite directions.

The above algorithm is optimistic in nature; in other words, a failed read operation retries until successful. As mentioned earlier, Lamport demonstrated that, in a sequentially consistent memory, atomic reads and writes can be implemented from non atomic reads and writes without mutual exclusion [94, 100, 101]. Since then, extensive research has been conducted in lock-free and wait-free synchronization [8, 11, 17, 58, 63, 64, 65, 66, 75, 78, 119, 126, 127, 128, 138, 149, 158, 164].

Lock-free and wait-free operation implementations consist of code that typically executes multiple atomic statements and does not involve mutual exclusion. The correctness conditions for lock-free and wait-free implementations are necessarily more complicated than for mutual-exclusion-based implementations. To reason about correctness of concurrent objects, *Linearizability* was proposed as a correctness condition [68]. Each operation is “invoked” in an “interval” of time. Since concurrent invocation of operations is possible by multiple processes, such intervals may overlap. Thus, for a correct execution of a series of invocations, each invocation on an object must appear to the invoking process to be executed instantaneously at some distinct point during the invocation's interval.

Relevant here is the notion of a universal synchronization primitive. An object is *universal* if it can be used as a building block to provide a wait-free implementation of any other object. Herlihy drew the link between universality and the consensus problem[5] [136] and proved that any object with a consensus number *n* is universal in a system of at most *n* processors [63]. He further showed that compare&swap and load-linked/store-conditional are both universal if one assumed unbounded memory (these primitives are discussed later). Plotkin proposed a sticky bit and showed it to be universal even with bounded memory [138].

Until the late 1980s, most architectures did not have support for universal synchronization primitives. To implement such primitives, Bershad proposed an efficient software-only mechanism [17] and restartable atomic sequences [18]. Both techniques assume that the operating system is aware of any long delay a process may encounter. The operating system can thus restart an atomic action that experiences the delay before the operation completes. A lock can be implemented without a universal synchronization primitive and protects the sequence. Waiting processors spin on the lock before performing the operation. The time the lock is held is bounded because the operating system releases the lock if the process is delayed. The operation is atomic with respect to processes on the same processor since the operating system will restart the sequence if preempted. The operation is atomic with respect to processes on other processors because it is protected by a lock.

While substantial research has been conducted in lock-free and wait-free synchronization, such techniques are quite difficult to design and verify as correct [64, 126]. To allow the easy development of correct concurrent objects, Herlihy proposed *universal constructions* [63]. A universal construction takes as input a sequential implementation of an object and produces a lock-free or wait-free implementation of the given object.

While practical universal constructions have been proposed, these implementations still suffer from significant time and space overhead and complexity of reasoning about correctness. Herlihy's constructions [64, 65] required copying of the entire shared object, sometimes multiple times. The

[5] The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value.

space and time overhead for performing and maintaining the copies is quite large for large objects. While optimizations have been proposed for reducing this overhead, the process of doing so is quite complex. Further, concurrent access to the object is not allowed. Barnes presented a mechanism allowing concurrent access to the object [11]. The proposal required the object be protected by a number of locks. Operations on the object acquire locks associated with relevant parts of the object in such a way that processes can “help” each other to perform operations and release locks. Barnes’ technique is not wait-free. Since locks are used, the programmer must still deal with concurrency. Another drawback is the presence of “useless helping”. If a process *p* that is helping another process *q* encounters a lock that is held by a third process *r*, then *p* must help *r* before helping *q*. This gives rise to long chains of useless helping.

Moir [126] extended Herlihy’s universal constructions and developed more efficient universal constructions. Recently, Moir proposed the use of a lock-free multi-word compare-and-swap (MWCAS) operation for efficient support for software wait-free transactions [127]. Israeli and Rappoport proposed lock-free constructions for multi-word synchronization primitives [76].

The compare&swap (CAS) was introduced in the IBM System/370 architecture [24] and is supported in some current architectures [166]. A CAS operation atomically swaps the value of a memory location with that of a register subject to a comparison.

The load-linked/store-conditional instructions (LL/SC), originally proposed by Jensen et al. [78] have emerged as popular primitives for lock-free read-modify-write operations on a single word. These instructions expose the steps involved in performing an atomic read-modify-write operation to the programmer and normally rely on the cache coherence mechanism to ensure correctness. The load-linked (LL) instruction loads a memory location into a processor register. This is followed by an arbitrary sequence of operations involving the register. The second special instruction in the pair, store-conditional (SC), attempts to write to the same memory location as the previous LL instruction. The store-conditional will succeed only if the hardware can guarantee that no other processor has successfully written to the memory location since the previous LL instruction was executed. The success of an SC instruction implies that a read-modify-write sequence has occurred atomically, completing at the time of the SC. In the case of a failure, the entire sequence may be retried.

The LL/SC paradigm has been adapted for several architectures, including Alpha [28], MIPS [54], and IBM PowerPC [31] and is also known as load-locked/store-conditional and load-with-reservation/store-conditional in various architectures. In various implementations, a link flag and registers are used to store the load-link information. A register typically stores the physical address to which the LL was issued. The link flag is set when the LL is issued. The success of an SC can only be determined at the point of coherency, that is, at the time a write operation can be performed on the designated memory location. If, at that point, the link flag was still set, and no incoming invalidate to the address in the locked physical address register is encountered, the store-conditional can successfully complete. Implementations differ among architectures, and implementation details vary considerably depending on the type of coherence mechanism provided. In either snoop-based or directory-based implementations, a store-conditional is treated differently from a traditional write operation, because the write of the store-conditional may or may not be completed on a given execution, and because the success of the operation must be reported back for testing by the processor.

While the basic concept is elegant and simple, in theory permitting the implementation of arbitrarily complex synchronization primitives, in practice it is difficult to design a system that can reliably guarantee success of the sequence. Two obvious problems that must be accommodated demonstrate the difficulty: (1) A memory conflict that forces the cache block containing the variable to be evicted. This problem could be dealt with in a variety of ways, but the simplest is to prohibit memory operations that might cause such an eviction; (2) An intervening page fault or other interrupt that may result in a large delay between the LL and SC. It is difficult to account for all possible actions that might occur before the following SC is executed, and a simple implementation will simply reset the link flag upon encountering most or all types of traps. Because of such considerations, each architecture provides a set of guidelines or requirements to increase the likelihood of success, but even with such constraints, guarantees of success are very carefully worded. For example, according to the Alpha Architecture Handbook [28], a write of a different word on the same *block* (the size of a block being an implementation-dependent constant of some power of two, being no smaller than a cache block, and no larger than a page) as the target address may cause the SC to fail. In addition, numerous system calls or traps will cause the link flag to be reset. Other restrictions apply as well, with the handbook indicating that “no useful program should do this” because a sequence may always fail for some implementations. Such restrictions include (1) that there be no instructions that access memory between the LL and SC instructions, (2) that there be no taken branches between the immediately preceding LL and SC instructions (the processor may execute multiple LL instructions before it attempts the SC), and (3) that a “large number” of instructions not be executed between the LL and SC. The term “large number” is not defined — though the specification does require a minimum number of instructions every implementation must execute between timer resets.

The requirements are similar in other architectures and implementations. A requirement in all implementations we have studied is that an LL instruction on one processor must not affect any architecturally visible state on another processor, and in particular cannot cause a SC on another processor to fail. This restriction is necessary to make possible forward progress claims, though great care is necessary in the design of the inter-processor communication mechanism to prevent starvation of processor nodes relying on LL/SC for fairness.

The load-linked/store-conditional primitives were later extended by Herlihy and Moss [66] and Stone et al. [158].

Herlihy and Moss proposed Transactional Memory [66]—a hardware mechanism that allows programmers to write transactions that execute atomically or fail without updating memory and thus implement lock-free data structures. Transactional Memory requires six new instructions for programmers to use and uses an extra cache called the transactional cache to buffer optimistic updates. Transactional Memory supports arbitrary read-modify-write operations and the size of the operations is limited only by the processor’s transactional cache. The basic insight behind Transactional Memory is that invalidation-based cache coherence protocols can be used to detect transaction conflicts. By using the existing cache coherence protocol, atomic transactions can be supported cheaply. Transactional Memory still requires programmers to reason about correctness of lock-free algorithms. The evaluation results showed that Transactional Memory outperforms locking implementations for all their benchmarks. Transactional memory relies on exponential backoff to provide forward progress and thus is strictly not non-blocking.

Stone et al. proposed Oklahoma Update [158], which exploits the existing cache coherence protocol in the same fashion as Transactional Memory. The Oklahoma Update adds a set of special reservation registers to each processor and uses the concepts of transactional loads and stores. Transactional loads are like normal loads except they also update the reservation registers. Transactional stores do not update memory and only update the reservation registers. At commit, the Oklahoma Update resorts to a two-phase commit protocol. In the first phase, the precommit phase, addresses are sorted in ascending order, are checked for validity, and exclusive ownership for the appropriate addresses is requested. If any reservation is invalid, or the exclusive ownership request fails, the transaction aborts and restarts. During the precommit phase, all external ownership requests for addresses lower than the currently active address are deferred until after commit. If the precommit phase is successful, the commit phase atomically updates all modified shared variables by writing the data in the registers to the cache. During the commit phase, all external requests are deferred. They do not provide any simulation numbers. While the Oklahoma Update is non-blocking, it is not wait-free.

Shavit and Touitou proposed Software Transactional Memory [149], a purely software implementation of Transactional Memory. Their scheme as presented does not support dynamic transactions (i.e., if the set of locations accessed by the transaction are not known at the start of the transaction). Software transactional memory is lock-free but not wait-free.

Object specific lock-free implementations have also been proposed. Such implementations take advantage of the semantics of the object under consideration to improve performance. These techniques also suffer from the complexity of reasoning about correctness. Some techniques use instructions stronger than normal reads and writes. Various concurrent queue implementations not relying on mutual exclusion fall in this category [67, 75, 99, 122, 167]. Valois proposed lock free implementations of common data structures such as queues, trees, and lists [164, 165]. Massalin and Pu implemented an operating system using only lock-free synchronization techniques [119].

The terms non-blocking synchronization and lock-free synchronization have been used interchangeably. A lock-based synchronization primitive can be made non-blocking with some effort. However, locking algorithms, implemented conventionally, are blocking. On the other hand, all lock-free and wait-free techniques are, by definition, non-blocking. The most effective use of non-blocking synchronization has been in the area of data-structure-specific algorithms. Greenwald provides an extensive discussion of non-blocking synchronization techniques in his thesis [58].

2.2.3.2 Limitations of lock-free and wait-free techniques

Lock-free and wait-free techniques often require more complex operations than critical sections and rely on programmers to write appropriate code. Programmers have to reason about correctness in the presence of complex data structures. These alternatives commonly suffer from difficulty of use, complex programming methodologies, and often high software overheads, thus aggravating the trade-off between complexity and performance. These techniques have been shown to perform poorly with respect to lock-based schemes in the absence of failures and delays [5, 17] primarily due to excessive data copying to allow rollback.

While substantial theoretical and practical research has been conducted in making lock-free and wait-free techniques more efficient, a performance gap nevertheless remains and nearly all proposals have required programmers to reason about correctness of the algorithms. Such techniques are quite difficult to design and verify as correct. Most software, such as database servers, web servers, and virtual machines, still rely on the intuitive parallel correctness reasoning model of mutual exclusion and lock-based synchronization.

2.2.4 Database concurrency control

Extensive research has been conducted in databases on concurrency control and Thomasian [161] provides a good summary and further references. Bernstein and Goodman present an extensive discussion of concurrency control mechanisms in distributed database systems [16]. Books by Papadimitriou [135] and Bernstein, Hadzilacos, and Goodman [15] cover the topic of concurrency control in detail.

Optimistic Concurrency Control (OCC) was proposed by Kung and Robinson [90] as an alternative to locking in database systems. OCC involves a read phase where objects are accessed (with possible updates to a private copy of these objects) followed by a serialized validation phase to check for data conflicts (read/write conflicts with other transactions). This is followed by the write phase if the validation is successful.

In spite of extensive research, there are no commercially successful database systems that use OCC as a concurrency control mechanism.[6] Haerder [60] was the first to point out potential problems with OCC schemes. An excellent discussion regarding the issues involved with OCC approaches and their shortcomings which make OCC unattractive for high-performance database systems is provided by Mohan [124]. Special requirements of and guarantees required by database systems, specifically for storage management, access path maintenance, recovery models, fine-granularity conflict checking, fine-grain locking and semantically-rich lock modes [125],

[6] Some objected-oriented database systems (such as Ontos) used to provide options for specifying optimistic concurrency control or locking as the concurrency control mechanism but these systems have not been successful [130]. OCC in relational vendors is summarized elsewhere [142].

make OCC hard to use for high performance. To provide these guarantees, substantial state information must be stored in software resulting in large overheads in executing transactions. In addition, with OCC, the validation phase is often serialized, thus limiting performance.

2.3 Safety and liveness in concurrency control algorithms

Safety and liveness were first described by Lamport [96]. We consider two properties under safety: serializability and deadlock freedom. In later chapters, we will construct arguments to show why TLR provides a correct execution. We also discuss liveness and show how to provide a sense of fairness.

2.3.1 Safety

Lamport informally defined safety as "bad things do not happen" [96]. A safety property constrains permitted actions, and therefore the allowed state changes of a program—actions an algorithm may do. In general, a safety specification may be any safety property which is one that holds for an execution if and only if it holds for all finite initial segments of the execution. Mutual exclusion, deadlock freedom, serializability, FIFO processing, and partial correctness are all safety properties. Two safety properties we are interested in are: serializability and freedom from deadlock.

2.3.1.1 Serializability

Serializability is a correctness condition commonly assumed by database and distributed systems. *Serializability* requires the result of executions of concurrent transactions to be *as if* there were some global order in which these transactions had executed serially [39]. While similarities exist between serializability and sequential consistency, the two correctness conditions target different problem domains. By treating critical sections as transactions and thus constraining them to satisfy serializability conditions, we can transparently apply much of the theoretical work in transactions while allowing programmers to use critical sections as their model of choice for reasoning about sharing. For our discussion, sequential consistency is orthogonal to serializability considerations mainly because sequential consistency deals with ordering among individual memory locations while serializability deals with ordering among multiple memory locations.

Our solution, while maintaining the appropriate underlying memory consistency model as specified by the system, also meets serializability as a correctness condition, thus achieving the semantics of critical sections without requiring explicit lock acquires. One can envisage critical section executions that are not serializable (i.e., programs that have data race constructs in them) that may nevertheless be correct in a given execution, but all serializable executions will provide the functionality and semantics of critical sections.

Since we are treating critical sections as optimistic lock-free transactions, we are interested in serializability as a safety property. Let E denote an execution of transactions T_{1},..., T_{n}. E is a serial execution if no transactions execute concurrently in E; i.e., each transaction is executed to completion before the next one begins. Every serial execution is defined to be correct because the properties of transactions imply that a serial execution terminates properly and preserves memory consistency. An execution is serializable if it is computationally equivalent to a serial execution, that is, if it produces the same output and has the same effect on the memory image as some serial execution. Since serial executions are correct and every serializable execution is equivalent to a serial one, every serializable execution is also correct.

We reproduce two theorems provided by Bernstein [14] and Papadimitriou [134] to characterize serializable executions precisely. Much of the discussion in this section is taken from Bernstein and Goodman [16] and provides a context for the discussion of correctness of our proposal.

Theorem 1

Let T = {T_1,..., T_m} be a set of transactions and let E be an execution of these transactions modeled by logs {L_1,..., L_m}. E is serializable if there exists a total ordering of T such that each pair of conflicting operations O_i and O_j from distinct transactions T_i and T_j (respectively), O_i precedes O_j in any log L_1,...,L_m if and only if T_i precedes T_j in the total ordering.

The order hypothesized by theorem 1 is called a serialization order. To attain serializability, all executions must satisfy the condition of theorem 1, namely, that conflicting reads and writes be processed in a certain relative order.

While theorem 1 treats *rw* (read-write) and *ww* (write-write) conflicts together under the general notion of conflicts, these two types of conflicts can be further distinguished. For each pair of transactions, T_i and T_j,

1. $T_i \rightarrow_{rw} T_j$ if in some log of E, T_i reads some data item into which T_j subsequently writes
2. $T_i \rightarrow_{wr} T_j$ if in some log of E, T_i writes to some data item that T_j subsequently reads
3. $T_i \rightarrow_{ww} T_j$ if in some log of E, T_i writes into some data item into which T_j subsequently writes
4. $T_i \rightarrow_{rwr} T_j$ if $T_i \rightarrow_{rw} T_j$ or $T_i \rightarrow_{wr} T_j$
5. $T_i \rightarrow T_j$ if $T_i \rightarrow_{rwr} T_j$ or $T_i \rightarrow_{ww} T_j$

Every conflict between operations in E is represented by an → relationship. Therefore, theorem 1 can be restated in terms of →. E is serializable if there is a total order of transactions that is consistent with →, and this is possible only if → is acyclic. Theorem 1 is restated below using the → relationship.

Theorem 2

Let $\rightarrow_{rwr}$ and $\rightarrow_{ww}$ be associated with execution E. E is serializable if (a) $\rightarrow_{rwr}$ and $\rightarrow_{ww}$ are acyclic, and (b) there is a total ordering of the transactions consistent with all $\rightarrow_{rwr}$ and all $\rightarrow_{ww}$ relationships.

While different techniques can be used to guarantee the acyclic nature of $\rightarrow_{ww}$ and $\rightarrow_{rwr}$, there must be one serial order consistent with all → relations. Thus, to show any execution serializable, we only need to show the graph to be acyclic. We will use this property later when we describe our mechanisms in detail.

Serializability vs. atomicity. Lamport discusses the relationship between serializability and atomicity [102]. In the absence of failures that result in transaction aborts, atomicity and serializability can be considered equivalent. For example, while a semaphore operation is atomic, a database transaction appears to be atomic and the atomicity of database operations is achieved using a serializable execution order.

Examples of serializable and non-serializable schedules. An example of a serializable schedule is shown in the left part of Figure 2-3. Two transactions T1 and T2 are shown accessing locations *A*, *B*, and *C*. The two transactions do not conflict on any data access and thus can execute concurrently. A non-serializable schedule occurs when two transactions access the same data item,

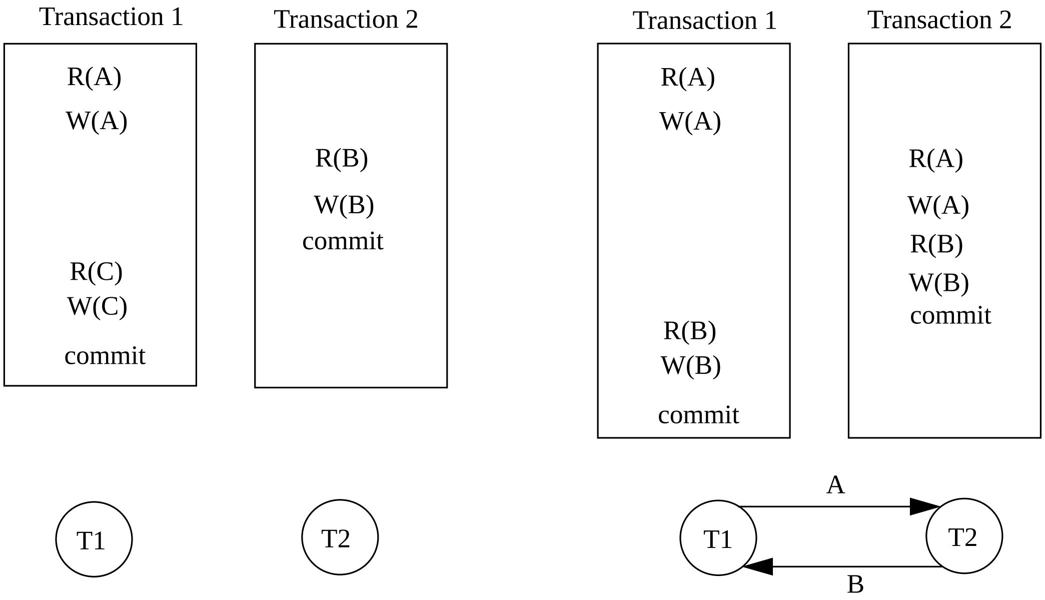


Figure 2-3: Serializable and non-serializable examples. *"A", "B", and "C" are locations accessed by the transactions. On the left, T1 and T2 do not depend on each other for the execution and can concurrently execute. On the right, T1's output depends on T2 and T2's output depends on T1. A cyclic dependence exists and these concurrent executions cannot be serialized.*

one of the transactions is modifying the data item, and the execution does not correspond to any serial execution. A serial execution requires T1 and T2 to appear to occur in some serial order. However, in the right part of Figure 2-3, T1 writes *A* and reads *B*. In between the two actions, T2 reads *A* and writes *B*. Thus, there is a cycle in the dependence graph. Three general conflict situations are:

1. *Write-read conflict.* T2 reads something T1 wrote

2. *Read-write conflict.* T2 overwrites what T1 read and T1 reads it again

3. *Write-write conflict.* T2 overwrites what T1 wrote

All these conflict situations must be avoided or the execution must be ordered so that these transactions do not occur concurrently.

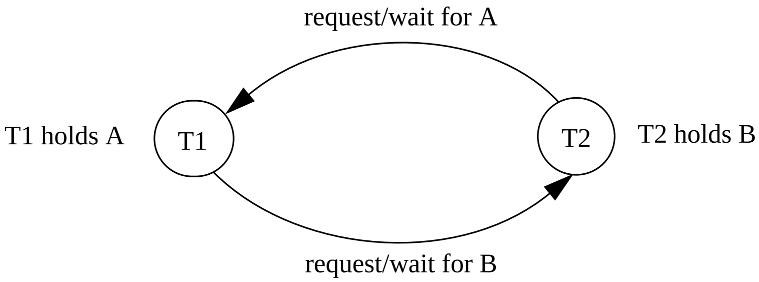


Figure 2-4: Deadlock with two transactions. *Transactions T1 and T2 are part of a cycle in the waits-for graph. T1 waits for T2 for B while holding A and T2 waits for T1 for A while holding B.*

2.3.1.2 Freedom from deadlock

A transaction typically achieves serializability by obtaining ownership of various data items the transaction is accessing, and then executing to completion. If ownership cannot be acquired, the transaction must wait until it can be acquired. In database systems, these ownerships are acquired via the use of locks. In the remainder of this section, we discuss common techniques by which database systems avoid deadlock while acquiring locks.

If a lock is held by a thread, another thread may have to wait for the lock. Such waiting for unavailable locks may be uncontrolled, thus leading to deadlock. Thus, the second safety property we are interested in is deadlock freedom. Deadlock situations can be characterized by *waits-for* graphs [71]—directed graphs that indicate which transactions are waiting for which other transactions. Nodes of the graph represent transactions, and edges represent the “waiting for” relationship: an edge is drawn from transaction T_i to transaction T_j if T_i is waiting for a lock currently owned by T_j. There is deadlock in the system *if and only if* the waits-for graph contains a cycle. This is illustrated in Figure 2-4.

Deadlock detection and deadlock prevention are two common techniques available for deadlock resolution. Deadlock detection allows deadlock to occur and the deadlocks are detected periodically by explicitly constructing the waits-for graph and testing for cycles. If a cycle is found, one transaction in the cycle is aborted thereby breaking the deadlock. Deadlock prevention is a conservative scheme in which a transaction is restarted when a system thinks a deadlock may occur. A watchdog timer is also commonly employed that detects deadlock with some false positives.

Deadlock detection. The difficulty in implementing deadlock detection in a distributed system is in the efficient construction of the waits-for graph at a global level. This technique requires periodic transmission of local waits-for graph information to the deadlock detector sites in order to construct a global picture of the waits-for graph and detect cycles.

Deadlock prevention. Consider two transactions T_i and T_j, and T_i requests a lock currently owned by T_j. If T_i is restarted, the deadlock prevention algorithm is considered non-preemptive. If T_j is restarted, the algorithm is considered preemptive.

A common approach to deadlock prevention is to assign priorities to transactions. Again, consider two transactions T_i and T_j. T_i could wait for T_j if T_i has a lower priority than T_j. This prevents deadlock because for every edge (T_i, T_j) in the waits-for graph, T_i has a lower priority than T_j. Since T_i cannot have a lower priority than itself, no cycle can exist.

Assigning static priorities can lead to cyclic restarts—a transaction may continually restart without ever completing. Rosenkrantz et al. [144] proposed using timestamps as priorities. Each transaction is assigned a unique timestamp. The timestamp consists of the local clock time appended with a unique identifier to the lower order bits. A new timestamp is not assigned until the next clock tick. Timestamps are unique across the system and the clocks at different sites do not have to be precisely synchronized. A restarting transaction is not assigned a new timestamp until it successfully completes.

Two timestamp-based deadlock prevention schemes proposed are Wait-Die and Wound-Wait [144].

1. *Wait-Die*. This is a nonpreemptive technique in which the requesting transaction either waits or dies (hence the name). Suppose T_i tries to wait for T_j. If T_i has lower priority than T_j, then T_i is permitted to wait; else it is aborted and forced to restart (“dies”).

2. *Wound-Wait*. This is a preemptive technique in which the requesting process in any conflict either waits or wounds the other process or processes (hence the name). Suppose T_i tries to wait for T_j. If T_j has higher priority than T_i, then T_i is permitted to wait; else T_j is aborted (“wounded”) and forced to restart.

Rosenkrantz et al. [144] present several distributed concurrency control algorithms and show them correct. We use key concepts (such as use of timestamps for deadlock-free concurrency control and starvation freedom) developed in that work and adapt them to our proposals.

Another common approach is preordering requests to avoid restarts altogether. This approach requires predeclaration of locks and each transaction acquires its locks before starting. Each lock is assigned a number and the priority of a transaction is the highest numbered lock it owns. The transaction requests locks serially (and one-at-a-time) in numeric order. No deadlock can occur because a transaction only waits for transactions with higher priority. The disadvantages of such a technique include the requirement for pre-declaration and the sequential acquisition of locks thus leading to increased response times.

2.3.2 Liveness

Informally, liveness dictates that something "good" must eventually happen during execution [96]. Examples of liveness properties include starvation freedom, termination, and guaranteed service. Liveness does not restrict what a "good thing" can be. Liveness properties cannot stipulate that some "good thing" *always* happens, only that it *eventually* (at an unspecified time later) happens [6].

Two liveness properties we are interested in are: freedom from livelock, and freedom from starvation.

2.3.2.1 Freedom from livelock

Informally, freedom from livelock may be paraphrased as: "If some process wants to execute a transaction, *some* process will eventually execute the transaction."

2.3.2.2 Freedom from starvation

Informally, freedom from starvation may be paraphrased as: "If some process wants to execute a transaction, then *that* process will eventually execute the transaction." To ensure starvation freedom, all threads must eventually succeed. This is achieved by using an appropriate conflict resolution scheme guaranteeing all conflict losers eventually become winners.

2.4 Speculative execution

Speculative execution has emerged as a key enabling technique for numerous processor innovations and is supported on nearly all modern processors [35, 70, 82, 108, 170]. Speculative techniques often require support for buffering speculative state and for recovering from misspeculation conditions. While most implementations today allow speculative updates to registers in the processor, they do not allow speculative updates to be propagated to the memory system. These updates may still be buffered in the store buffers of the processor core. Recently, proposals allowing for speculative values to update memory have also been made and we discuss them below. We discuss some applications of speculative execution and schemes for buffering speculative state.

2.4.1 Speculative execution proposals

In this section, we discuss some of the speculative execution proposals made. Three categories we discuss are: uniprocessor optimizations, aggressive memory consistency implementations, and speculative parallelization of sequential programs.

2.4.1.1 Uniprocessor program optimizations

While speculation on branch direction through the use of branch prediction is common, recent proposals have included speculative execution based on the predicted values of memory locations [114].

2.4.1.2 Aggressive implementation of memory consistency

Speculative execution for aggressive implementation of memory consistency models was proposed by Gharachorloo et al. [45]. As we discussed earlier in Section 2.1.1, memory models enforce an ordering on the execution of certain memory operations. Gharachorloo et al. proposed a technique where the ordering restriction for loads was relaxed. These synchronization loads [46] were speculatively executed using the processor reorder buffer and any violations were detected using the cache coherence protocol. This was later extended by Ranganathan et al. [143]. They used speculative retirement for tolerating longer latencies and used a history buffer to recover from any misspeculation. Gniady et al. [48] further extended the techniques by using additional speculative hardware support. These techniques showed ways to reduce the performance gap between sequential consistency and relaxed memory models.

2.4.1.3 Speculative parallelization of sequential programs

To the best of our knowledge, the first proposal for speculative parallelization of sequential programs was by Knight in the context of functional languages [86]. Knight described an architecture to automatically extract and execute parallel portions of a sequential program while giving the appearance to the programmer that the program is being executed sequentially. Hardware was used to dynamically check the correctness of the execution and fully associative caches were used as a means of maintaining and enforcing dependencies between portions of the program.

The Multiscalar proposals [41, 154] have popularized and driven current research into speculative parallelization. Multiscalar divided a single program into a collection of tasks by a combination of software and hardware and the tasks were distributed to numerous parallel processing units on the processor with one task always being the non-speculative task. This research led to other more general applications of speculation such as memory dependence speculation where memory accesses may occur speculatively without knowledge of preceding loads or stores [129]. Other thread-level speculations proposals followed the Multiscalar work [61, 156].

2.4.2 Handling speculative state

Buffering speculative register state is well studied and supported in modern processors either in the form of checkpoints, history buffers, or future files [153]. Nearly all proposals for speculative execution discussed earlier use local buffers to store speculative updates to memory. Knight [86] used the “confirm cache” local to each processor to store uncommitted data. Herlihy and Moss [66] used a “transactional cache” to track and buffer speculative updates of the transactions. The multiscalar work proposed the “address resolution buffer” [42] and the “speculative versioning cache” [52] to perform memory disambiguation and store speculative memory updates.

Gharachorloo et al. [45] used the processor reorder buffer to track speculatively issued loads and the coherence protocol to check for violations. Ranganathan et al. [143] used a history buffer [153] to store speculatively retired instructions. These two schemes do not update memory speculatively. Gniady et al. [48] use a special buffer, the Store History Queue, to buffer speculative updates to memory.

2.4.3 Detecting violations

Nearly all techniques discussed above that speculate on memory ordering and speculate across speculative threads use the cache coherence mechanisms to detect violations. Knight proposed using cache coherence protocols in the context of speculatively parallelizing sequential code [86]. Subsequently the Herlihy and Moss [66] used the same mechanism for implementing Transactional Memory. Gharachorloo et al. [45] used cache coherence protocols for detecting violations to memory ordering. Franklin proposed the use of the address resolution buffer for detecting data races in shared-memory multiprocessors [40].

2.5 Chapter summary

We have presented concepts key for understanding the thesis and have provided a background into related work in the area of synchronization, concurrency control, and speculative execution. We use concepts developed in database concurrency control and we use much of the hardware support proposed for speculative execution in our work.

We do not study barriers as a synchronization method for coordinating activities in a parallel program. Barriers are well studied [53, 85, 109, 137, 148] and recent work has proposed speculating past barriers in parallel programs [146]. This thesis is concerned with programs that use lock-based synchronization.

Chapter 3

Speculative Lock Elision

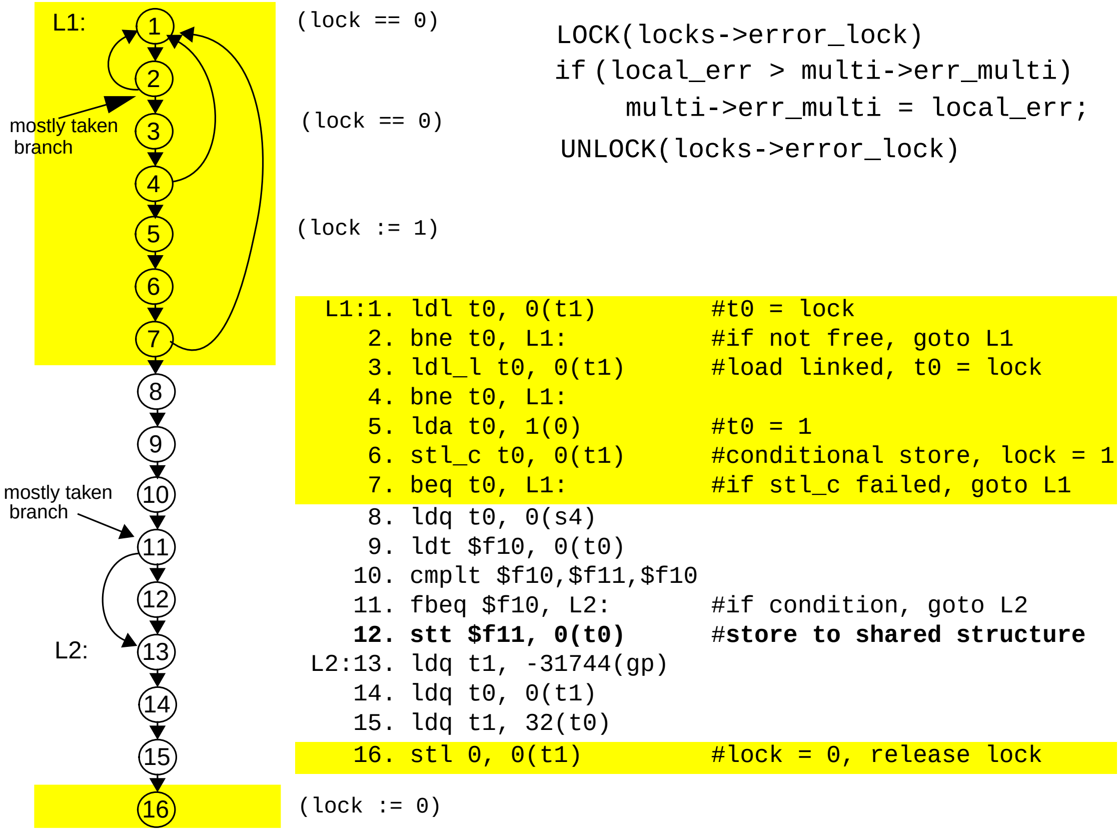
This chapter presents Speculative Lock Elision[1] (SLE). SLE is a hardware technique to elide lock operations from a dynamic execution if the locks were not required for correctness. We first discuss two examples to demonstrate that a lock acquisition may not be necessary for executing a critical section if data conflicts did not occur among various threads concurrently executing threads in a particular dynamic execution. Figure 3-1 shows an example from a multithreaded application `ocean-cont` [169]. The C code and the corresponding ALPHA instructions [28] are shown. Since a store instruction (inst. i12) is present, the lock is required to coordinate access to the shared structure. Branch instruction (inst. i11) is a mostly taken branch because of conditional error code calculations and most dynamic executions follow the control path "7,8,9,10,11,13,14,15" within the critical section where the store instruction (inst. i12) is not executed. These executions do not require a lock.

Another example where lock acquisitions may not be necessary is when multiple threads update disjoint fields of a shared object while holding the shared object lock. A thread-safe hash table is such an example and is shown in Figure 3-2. This example is similar to an implementation from SHORE, a database object repository [23]. Although hash lookups and updates can usually proceed concurrently, the lock prevents such parallelism from being exposed to the microarchitecture, thus serializing execution and limiting performance.

Critical sections provide a mechanism to obtain atomic access (all reads and writes in the critical section appear to occur atomically and instantaneously) to shared objects (Section 2.2).[2] Here,

[1] Elision: the act or an instance of dropping out or omitting something.

[2] Since critical sections are a software convention, atomic access (including atomic updates) is possible only if such a convention is followed. In the presence of unprotected accesses—cases where protected data is accessed without a lock—critical sections may not provide atomicity of updates.

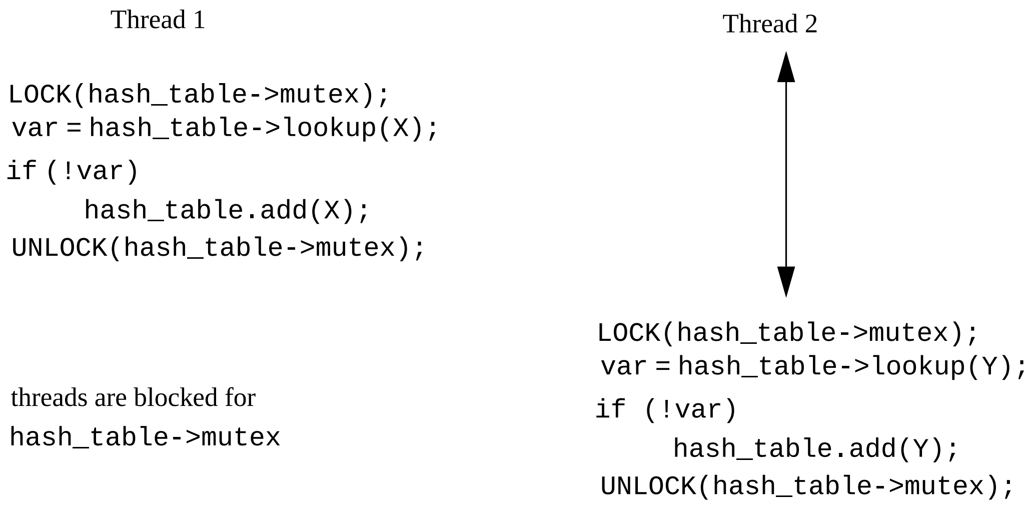


```
LOCK(locks->error_lock)
if (local_err > multi->err_multi)
    multi->err_multi = local_err;
UNLOCK(locks->error_lock)
```

```
 L1:1. ldl t0, 0(t1)          #t0 = lock
    2. bne t0, L1:            #if not free, goto L1
    3. ldl_l t0, 0(t1)        #load linked, t0 = lock
    4. bne t0, L1:
    5. lda t0, 1(0)           #t0 = 1
    6. stl_c t0, 0(t1)        #conditional store, lock = 1
    7. beq t0, L1:            #if stl_c failed, goto L1
    8. ldq t0, 0(s4)
    9. ldt $f10, 0(t0)
   10. cmplt $f10,$f11,$f10
   11. fbeq $f10, L2:         #if condition, goto L2
   12. stt $f11, 0(t0)        #store to shared structure
L2:13. ldq t1, -31744(gp)
   14. ldq t0, 0(t1)
   15. ldq t1, 32(t0)
   16. stl 0, 0(t1)           #lock = 0, release lock
```

Figure 3-1: Control-flow induced unnecessary serialization. *The lock acquire and release sequence is shown shaded. In most executions, threads skip store inst. 12 and thus the lock is not required. However, this cannot be determined at compile time because in other execution instances, the store is performed. The lock often unnecessarily serializes execution of multiple threads.*

the appearance of instantaneous change is key. By acquiring a lock, a thread can prevent other threads from observing any memory updates that are in progress until the lock is released. While this conventional approach trivially guarantees atomicity of all access (including updates) in the critical section, it is only one way to achieve atomicity. In this chapter, we present another way to achieve such atomicity—*Speculative Lock Elision (SLE)*.



```
LOCK(hash_table->mutex);
var = hash_table->lookup(X);
if (!var)
     hash_table.add(X);
UNLOCK(hash_table->mutex);
```

```
LOCK(hash_table->mutex);
var = hash_table->lookup(Y);
if (!var)
     hash_table.add(Y);
UNLOCK(hash_table->mutex);
```

Figure 3-2: Locking-granularity induced unnecessary serialization. *Example of a thread-safe hash table is shown. With good hash functions, conflicts are not common and thus the operations to the hash table would occur without conflicts. However, access is unnecessarily serialized. A similar thread-safe hash table (the table is protected by a single lock) is used in [23], a database repository manager.*

3.1 Chapter roadmap

The chapter is divided into four parts. The first part (Section 3.2 through Section 3.7) discusses the concepts behind Speculative Lock Elision. The algorithm of SLE is presented in Section 3.4. We introduce the concept of silent store-pairs in Section 3.5 and show how it can be employed to elide lock operations.

The second part, Section 3.9, discusses implementation details of SLE. Included in this section is a discussion of identifying regions of speculation, buffering speculative state, conditions for misspeculation, recovering from misspeculation, and committing speculative state.

The third part comprises of Section 3.10, Section 3.11, and Section 3.12. These sections discuss SLE’s handling of nested critical sections, and SLE’s interactions with software and hardware.

The fourth part is the rest of the chapter. Related work is contrasted in Section 3.13 and a chapter summary is presented in Section 3.14.

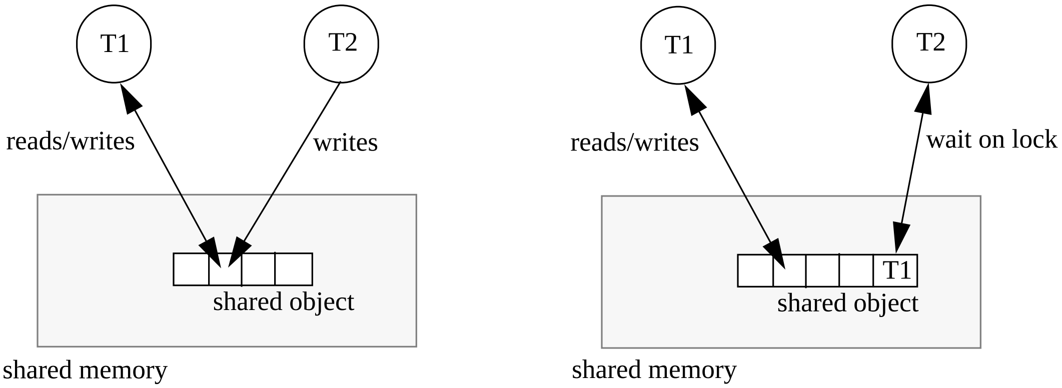


Figure 3-3: Data conflict and lock contention.

3.2 Data conflict and lock contention

We distinguish between *data conflict* and *lock contention*. Consider Figure 3-3. Two threads, T1 and T2, operate on a shared object (represented as an array of memory locations) in shared memory. The left half of the figure shows the threads accessing a common data memory location unprotected by a lock. A *data conflict* occurs if at least one thread is writing to a data memory location simultaneously while another thread is accessing the same data memory location. In such situations, atomicity of operations (read and write operations to the data location) cannot be guaranteed. The right half of the figure shows the same example but now the array locations are protected by a lock. Thread T1 owns the lock and performs memory accesses to the memory location while thread T2 waits for the lock to become free. We call this wait *lock contention*. Since the lock only allows one thread to access the object at any time, no data conflicts are experienced and the threads observe a consistent view of memory.

If the two threads access distinct memory locations, the lock is not required because a data conflict does not exist. However, this may not be known a priori and the lock is used to serialize access to the shared array. The lock unnecessarily inhibits concurrent access to the data structure even though correctness is guaranteed because data conflicts are not present. Thus, only data conflicts limit concurrency; lock contention by itself does not. Unfortunately, processors today do not differentiate between data conflicts and lock contention.

In shared-memory programming models, memory locations corresponding to lock variables are treated no different than the data variables they protect—locks and data share the same address space. The conceptual difference between locks and the data they protect is however fundamental—locks are the only locations associated with a critical section that may be touched from outside the critical section (this is true of programs that are free of data races). Lock contention is an example of a conflict because multiple threads race for the lock variable.[3] However, we do not consider this to be a *data conflict* because locks are not considered part of the critical section data.

3.3 Enabling concurrency by eliding locks

SLE enables concurrency in multithreaded program execution by removing unnecessary execution serialization on control variables such as locks.

As we show next, with SLE, lock acquires can be elided and critical sections can be concurrently executed and committed if serializability is maintained for all memory operations within the critical section. Serializability can be maintained by providing the “appearance” of atomicity of all operations in the critical section. For example, in a distributed system, two events may be initiated at the same physical time but the propagation delays inherent in the system may make them appear to have occurred at different instances. While locks enforce ordering of critical sections in physical time to achieve serializability, critical section executions can also be made to appear serializable without using locks to enforce ordering.

A processor can provide the appearance of atomicity for memory operations (and thus trivially provide serializability) within the critical section without acquiring locks by ensuring the partial updates performed by a thread within a critical section are not observed by other threads until the critical section completes. The desired effect is illustrated in Figure 3-4. The entire critical section appears to have executed atomically and program semantics are maintained.

The appropriate memory consistency model must be maintained for ordering the critical section operations with operations prior to the critical section and operations after the critical section. The specific ordering constraints are dependent upon the underlying memory model implemented and are discussed in Section 3.12.2.

[3] The test&set operation for implementing lock acquires constitute a race where multiple threads read the location while one (or more) threads write the location.

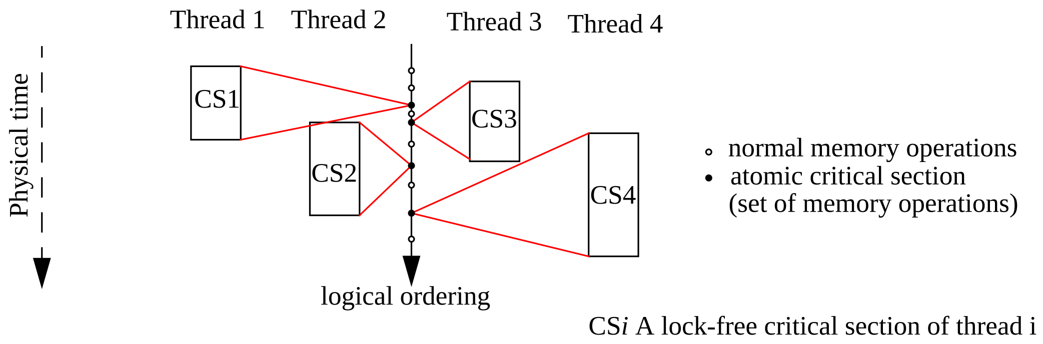

Thread 1
Thread 2
Thread 3
Thread 4
Physical time
CS1
CS2
CS3
CS4
logical ordering
∘ normal memory operations
• atomic critical section
(set of memory operations)
CS*i* A lock-free critical section of thread i

Figure 3-4: SLE and global memory ordering. *While critical section executions (without lock acquires) overlap in physical time (with or without data conflicts), each critical section logically appears to be inserted atomically and instantly in a logical ordering of memory operations with respect to other atomically inserted critical sections and individual memory operations.*

For serializability, the following conditions must be true for an execution of a speculative lock-free critical section:

1. Data read within a speculatively executing critical section does not appear to be modified by another thread before the speculating critical section completes.

2. Data written within a speculatively executing critical section does not appear to be accessed (read or written) by another thread before the speculative critical section execution completes.

If a data conflict occurs, i.e., two threads access the same data simultaneously other than for reading, serializability may not be guaranteed (because atomicity of updates cannot be guaranteed) and the lock needs to be acquired. Any execution not meeting the above two conditions is not retired architecturally. The tracking of data access by various threads and detection of data conflicts among threads rely on hardware mechanisms such as processor caches and cache coherence protocols. We discuss the details of these mechanisms in Section 3.9.

3.4 An initial algorithm for SLE

The algorithm for SLE is shown in Figure 3-5. A sequence of instructions is identified for atomic execution—we call this sequence the *atomic region*.[4] SLE aims at providing atomic execu-

[4] An atomic region is the same as a critical section except we are allowing multiple threads concurrently within the critical section. The term critical section typically implies mutual exclusion.

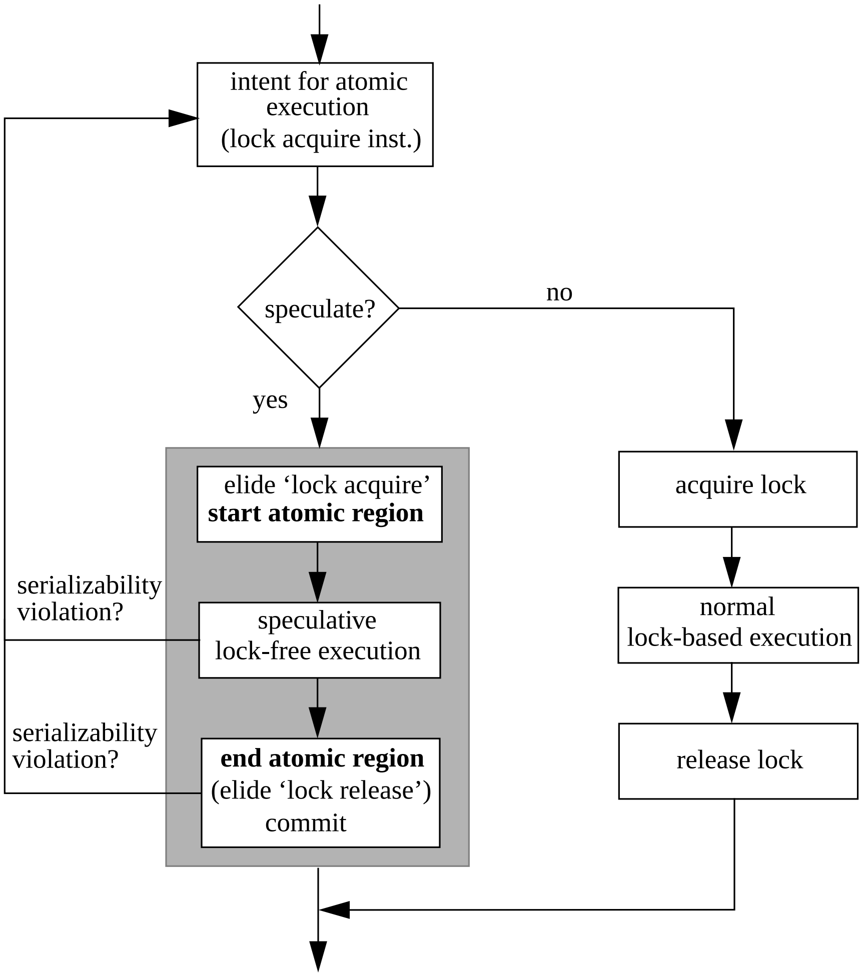

intent for atomic execution (lock acquire inst.)
speculate?
no
yes
elide 'lock acquire'
start atomic region
speculative lock-free execution
end atomic region
(elide 'lock release')
commit
serializability violation?
serializability violation?
acquire lock
normal lock-based execution
release lock

Figure 3-5: Initial algorithm for SLE. *The SLE algorithm is shown shaded. Updates of all writes performed within the speculative lock-free execution are buffered until commit. The serializability violation check is performed as the execution proceeds.*

tion without requiring lock acquires. When an intent for an atomic execution is observed or predicted (say, by the processor decoding a synchronization instruction), a prediction regarding whether the execution can be serializable without locks is performed. If the prediction is made in favor of speculative lock-free execution (i.e., the probability of having a successful conflict-free lock-free execution of the critical sections is high and correctness can be guaranteed without lock acquisitions), SLE is invoked. The lock acquire operation is elided thus leaving the lock variable in a free state. Doing so removes serialization on the lock variable and other threads can apply the same algorithm for higher concurrency. The critical section is speculatively executed in a lock-free manner. During the speculative execution if any condition precludes serializability, a misspeculation is triggered and execution restarts. Since execution is speculative, any update performed to memory or registers is speculative and thus must be buffered. End of the speculative execution region is determined by observing a lock release operation. At such point, the lock release operation is also elided and the updates (to memory) are committed in an atomic manner.

Even though the lock was not modified, either at the time of the acquire or the release, critical section semantics are maintained because all operations within the critical section appear to execute atomically with respect to other memory operations in the system and thus trivially guarantee serializability.[5] In the event of a misspeculation, a processor may repeat the speculation a bounded number of times in the event that the execution may still succeed without lock acquisitions. The number of times the processor restarts before explicitly acquiring the lock is the *restart threshold*. Forward progress is always guaranteed because the speculation can be forced to fail and the normal lock-based execution sequence is followed where locks are explicitly acquired.

Semantically, the lock is a control variable employed to provide the illusion of atomicity (by actually enforcing mutual exclusion) and thus removing the lock variable is acceptable if the illusion of atomicity of memory operations is provided by other means. However, to show the transformation of a lock-based execution to a lock-free execution to be correct, we need to study the instructions executed, independent of the semantics implied by the program. Further, modern instruction set architectures do not have a special instruction for acquiring and releasing locks. Instead, they provide atomic read-modify-write primitives that may be used to implement various locking algorithms and may also be used for various other operations unrelated to locks. Thus, often hardware does not have sufficient information to decide whether an instruction is accessing a lock. The processor only observes a sequence of loads, stores and synchronization primitives but cannot associate semantic information.

In the next section we introduce the notion of silent store-pairs and propose a technique that allows the transformation shown in Figure 3-5 without requiring precise semantic information from the software.

[5] This is discussed in detail in the appendix at the end of the dissertation. In short, the execution achieved by SLE corresponds to a legal execution of the critical sections if the locks had indeed been acquired.

```
if (lock == UNHELD)    // these two operations
    lock = HELD        // are executed atomically
      .
      .
      .
lock = UNHELD
```

Program Semantic	Instruction Stream	Value of _lock_	
		as seen by self	as seen by other threads
TEST _lock_	`L1:i1 ldl t0, 0(t1)`	FREE	FREE
	`i2 bne t0, L1:`		
TEST _lock_	`i3 ldl_l t0, 0(t1)`	FREE	FREE
&	`i4 bne t0, L1:`		
SET _lock_	`i5 lda t0, 1(0)`		
	`i6 stl_c t0, 0(t1)`	HELD	FREE
	`i7 beq t0, l1:`		
critical section	*i8-i15*		
RELEASE _lock_	`i16 stl 0, 0(t1)`	FREE	FREE

Figure 3-6: Silent store-pair elision. *The locking algorithm shown is test&test&set. Inst. i6 and i16 can be elided if i16 restores the value of _lock_ to its value prior to i6 (i.e., value returned by i3), and i8 through i15 appear to execute atomically with respect to other threads. Although the speculating thread elides i6, it still observes the held value itself (because of program order requirements within a single thread) but others observe a free value. The test corresponding to instruction i1 is shown but is not necessary for the elision. The elision relies on two store instructions; in this case the instructions are i6 and i16.*

3.5 Silent store-pair elision

We would like to elide lock acquire and release operations without knowledge of the semantics of the operations but must provide a correct execution. We make a key observation about lock variables. Lock acquire and release operations comprise store operations that "undo" each other. The lock acquire operation reads the lock, then *writes* the lock changing its value from `free` to `held`. A lock release operation also *writes* the lock changing its value from `held` to `free`. Figure 3-6 shows memory references of a lock acquire and release sequence in three columns. We assume the lock acquire algorithm is based on the popular test&set.[6] Instructions are numbered in program order. The first column shows the programmer's view, the second column shows the operations performed by the processor, and the third column shows the value of location _lock_ as seen by different threads.

If i3 returns `free`, i6 writes `held` to location _lock_. i16 releases the lock by marking it `free`. After the lock release (i16), the value of _lock_ is the same as it was at the start of the lock acquire (i.e., before i6)—i16 restores the value of _lock_ to its value prior to i6. We exploit this property of store operations to elide lock acquires and releases. If critical section memory operations appear to occur atomically, then stores i6 and i16 form a *silent pair.* The architectural changes performed by i6 are undone by i16. When executed as a pair, the stores are *silent* because the second store undoes the effects of the first thereby not affecting the architectural state due to the stores; individually, they are not. These two write operations (i6 and i16) delimit the set of instructions that are executed atomically. *Location _lock_ must not be modified by another thread, else i6 and i16 cannot form a silent pair.* Other threads can read memory location _lock_.

The above observation means the SLE algorithm need not depend on accurate program semantic information, specifically whether an operation is a lock acquire or lock release. SLE merely guesses such information and does not require a validation of the guess. The lock elision can be done by simply observing load and store sequences and the values read and to be written. If the location _lock_ is not modified by another thread, and the memory operations in the critical section appear to execute atomically, the two stores corresponding to i6 and i16 are elided. The location _lock_ is never modified, and other threads can proceed without being serialized on the value of _lock_.

What does store elision imply? Eliding a store means not exposing the new value outside the processor context and not requesting write permissions for the address. The store instructions are still fetched, dispatched, and committed by the processor core. In a multithreaded environment, if a thread elides a store, the value that the store operation would have written to the shared memory

[6] This includes the test&test&set algorithm and variants thereof. Discussion regarding other algorithms such as ticket locks, MCS software queued locks, etc. are found later in the chapter. Note the difference between a test&set instruction as proposed by the IBM System/360 and the test&set synchronization algorithm. In the example above, the test&set algorithm is implemented using the load-linked/store-conditional instructions. In the IBM System/360, this would correspond to a single test&set instruction.

space is not made visible to the other threads. The assumption is that a subsequent store will follow that will undo the changes made by the first store and thus the two stores would be silent. By eliding stores associated with lock operations, the lock is not written to and exclusive coherence permissions are *not* required. As a result, memory traffic and latency associated with lock operations—namely obtaining exclusive permissions on them—can be eliminated.

By maintaining the illusion of atomicity of operations between the two stores that are part of the silent store-pairs, the entire sequence of operations starting from the first store of the silent store-pair to the second store of the silent store-pair can be considered atomic because the architectural value of memory locations at the end of the sequence of operations would be identical even if the elided stores had actually been performed.

Additionally, since the execution of a critical section appears atomic, it can arbitrarily be reordered with respect to the memory operations of other threads (we discuss this later in Section 3.12.2). However for correctness, one must maintain program order. Thus, the value written by the elided store must be visible to the thread that elided the store (even though the value is not visible to other threads). This is required for maintaining program order and the implications of doing so are discussed in Section 3.11.2. The key point is that coherence ownership of the lock variable is not required to successfully execute and commit non-conflicting critical sections.

3.6 SLE algorithm using silent store-pair elision

To use SLE when the processor core cannot precisely identify lock acquire and release operations, we augment the SLE algorithm with the concept of silent store-pairs. An additional prediction is thus added to the algorithm of Figure 3-5. On a store operation, the processor predicts that the changes performed by the store will be undone shortly by another store, and no other thread will modify the location in question. If this occurs, the two stores can be elided because the entire sequence is globally observed to occur atomically and the value of the memory location corresponding to the two store operations remains the same at the start and at the end of the sequence. The first store can be seen as an approximation of the lock acquire operation and the second store can be viewed as an approximation of the lock release operation. The new SLE algorithm relying on silent store-pair elision is shown in Figure 3-7.

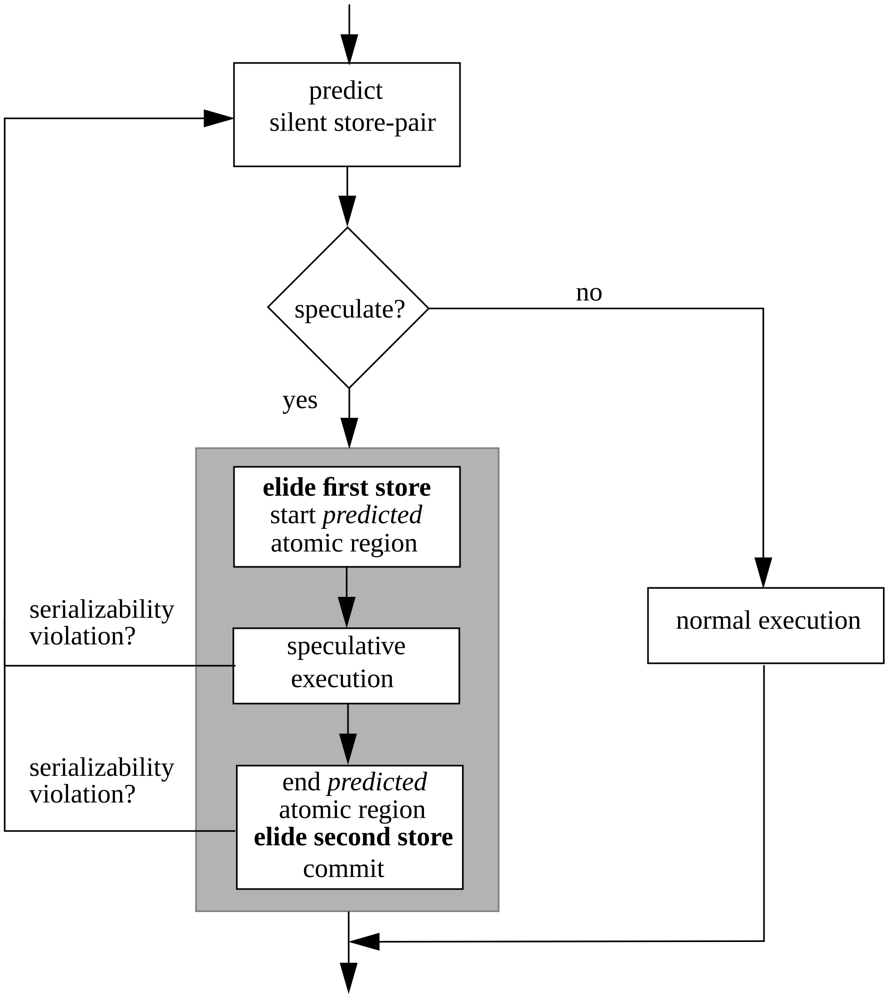


Figure 3-7: Algorithm for SLE using silent store-pair elision. *The SLE algorithm is shown shaded. The second elided store completes the silent store-pair. Unlike Figure 3-5, the algorithm shown above has no notion of lock acquire or release semantics. The algorithm identifies regions for atomic execution as delimited by the two stores that form a silent pair.*

The key difference between the new algorithm of Figure 3-7 and the earlier algorithm of Figure 3-5 is that the new algorithm does not rely on *semantic* information from the program. Unlike Figure 3-5 where the notion of lock acquires and releases was present in the algorithm, the new algorithm has no notion of lock acquires and releases. Instead, the speculation is based on the presence of silent store-pairs.

The critical section is deduced by the presence of silent store-pairs. When the first such store of the store-pair is identified, a determination regarding speculative execution is made. If the prediction is made to elide the store, the store elision is performed and speculative execution mode entered. The speculation typically ends when the second store of the silent store-pair is encountered and the speculation is committed. The two stores elided conveniently match the pattern we are interested in eliding—the lock acquire and release sequence. If the second store is encountered but it does not undo the effect of the first store, in other words the stores do not form a silent pair, the second store is performed and a commit is attempted and the processor makes a transition into a non-speculative mode.

The two stores may not actually correspond to the lock acquire and release operations since the hardware does not have semantic information. However, a correct execution will nevertheless be guaranteed—the instructions between the two elided stores will simply appear to be executed atomically.

3.6.1 Predictions and their resolution in SLE

SLE involves two key predictions that make no assumptions about the program semantics:

1. On a store, predict that another store will shortly follow and undo the changes by this store. The prediction is resolved without stores being performed (with respect to other threads) but it requires the memory location (of the stores) to be monitored. If the prediction is validated, the two stores are elided.

2. Predict that all memory operations within the region bounded by the two elided stores can be made to appear to execute atomically.

The above predictions do not rely on semantics of the program. In addition, no partial updates are made visible to other threads until the end of the critical section thus maintaining serializability of the execution and maintaining critical section semantics. Store elision works because the architectural state at the end of the second elided store is the same, with or without SLE.

3.6.2 In search of silent store-pairs

A naïve implementation of the above algorithm would apply the elision to every store operation assuming it forms part of a silent store-pair. However, most store-pairs would not form a silent store-pair. Further, it may not always be possible to exploit such situations.

Instruction pattern	Location *addr* value pattern
`load  X, addr`	FREE
`...`	
`store Y, addr`	HELD
`...`	
`store X, addr`	FREE

Figure 3-8: Detecting silent store-pairs patterns. *We are interested in identifying pairs of store instructions that match the store address and value pattern shown above. X and Y are values written to location addr.*

We have discussed how common lock operations match the patterns of silent-pairs. Thus, to detect opportunities for speculation, we only need to detect a silent-pair pattern. Silent store-pairs can be detected using various techniques. In this dissertation, we use a simple hardware-based predictor. Alternatively, software annotations can be employed by the compiler to reduce hardware requirements.

3.6.2.1 Simple hardware predictors

Detecting silent store-pairs involves detecting a sequence of instructions that match the store pattern of Figure 3-8.

This pattern may match non-lock operations also, but correctness is nevertheless guaranteed. Whenever such a pattern is detected, an atomic execution of all operations between the two store operations is attempted and this is always a correct execution. The precise implementation of the silent-pair predictor depends on the underlying architectural specification.

We must emphasize that while we are interested in eliding locks, our hardware detection mechanism is not specifically trying to detect locks. The hardware is *only* looking for silent store-pair patterns. Spin-locks often match such a simple silent store-pair pattern and thus we are able to apply it quite effectively.

3.6.2.2 Software annotations

As an alternative to hardware-based silent store-pair predictors, compiler hints can also be used for reducing the hardware required to detect the pattern in Figure 3-8. Importantly, these hints can be ignored and they need not be correct because program semantics are always guaranteed because SLE does not rely on this semantic information for correctness. Using software hints however requires either architectural support (in the form of new instructions) or a convention to pass information from the software to the hardware (e.g., by using some specific instruction sequences or `nops`).

3.6.2.3 Silent store-pairs and non-lock operations

The algorithm detects patterns independent of whether the instructions correspond to a lock-unlock pair; it does not rely on any semantic information. As such, if any silent store pair instructions are detected, they will be detected and potentially elided. While it is unclear whether doing this is useful for non-lock patterns, the technique will automatically detect and exploit such situations and provide a correct execution. Importantly, SLE will always provide a correct execution even if it detects non-lock silent store-pairs.

3.7 SLE algorithm example

Figure 3-9 shows the application of SLE to our earlier example Figure 3-1. The modified control flow is shown on the right with instructions 6 and 16 elided. All threads proceed without serialization. Instructions 1 and 3 bring the _lock_ into the cache in a shared state. Instruction 6 is elided and the modified control flow is speculatively executed. The location _lock_ is monitored for writes by other threads. All loads executed within the critical section are recorded. All stores executed within the critical section are temporarily buffered.[7] If instruction 16 is reached without any serializability violations, SLE is successful.

[7] Requests for exclusive ownership for the corresponding cache blocks are issued to the memory system but the data updates are not made visible until SLE commits.

```
LOCK(locks->error_lock)
if (local_err > multi->err_multi)
     multi->err_multi = local_err;
UNLOCK(locks->error_lock)
```

```
 L1:1. ldl t0, 0(t1)        #t0 = lock
    2. bne t0, L1:          #if not free, goto L1
    3. ldl_l t0, 0(t1)      #load linked, t0 = lock
    4. bne t0, L1:
    5. lda t0, 1(0)         #t0 = 1
    6. stl_c t0, 0(t1)      #conditional store, lock = 1
    7. beq t0, L1:          #if stl_c failed, goto L1
    8. ldq t0, 0(s4)
    9. ldt $f10, 0(t0)
   10. cmplt $f10,$f11,$f10
   11. fbeq $f10, L2:       #if condition, goto L2
   12. stt $f11, 0(t0)      #store to shared structure
L2:13. ldq t1, -31744(gp)
   14. ldq t0, 0(t1)
   15. ldq t1, 32(t0)
   16. stl 0, 0(t1)         #lock = 0, release lock
```

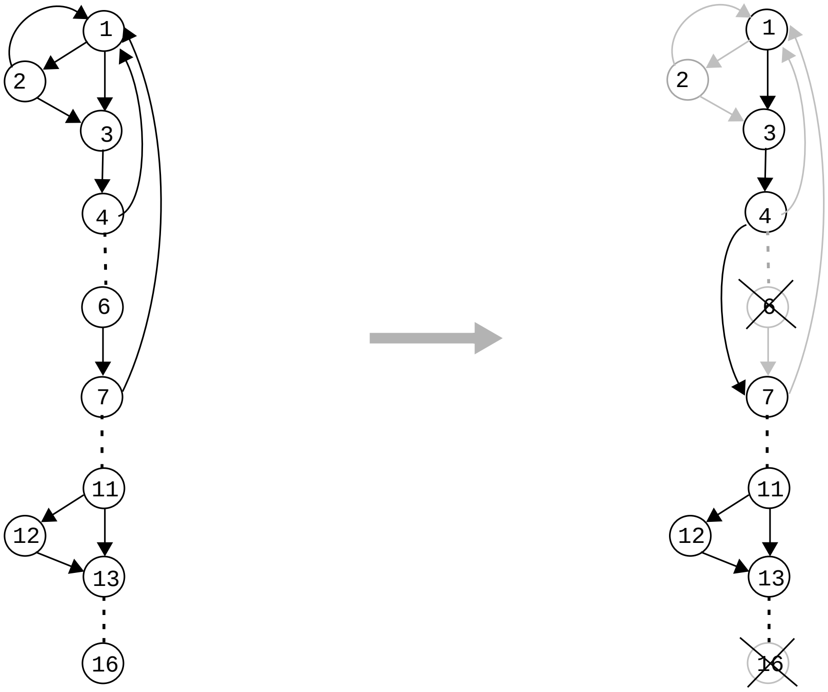


Figure 3-9: Speculative Lock Elision algorithm example. *Often, branch 11 is taken thus skipping the store inst. 12. The greyed portion on the right graph is not executed. Inst. 6 and 16 are elided and the code sequence executes with no taken branches between i1 and i8.*

If the thread cannot record accesses between the two stores, or the hardware cannot maintain serializability, a misspeculation is triggered, and execution restarts from instruction 6. On a restart, if the restart threshold has been reached, the lock is acquired and the execution occurs conventionally.

3.8 SLE key enablers

In this section, we discuss two key mechanisms that enable SLE implementations easily. The first is the concept of speculative execution and the second is the emergence of invalidation-based cache coherence protocols that make detection of data conflicts straightforward.

3.8.1 Speculative execution

The concept of speculative execution is well understood and widely used in modern processors. Keys to the success of speculative execution include a high probability of success, ability to buffer speculative state, fast validation of successful speculation, and a low overhead recovery from misspeculation. SLE handles speculative state (buffering state and recovering from misspeculation) in a way similar to that employed by other common speculative execution techniques and we discuss this later in Section 3.9.

SLE is unique in the way it validates a successful speculation. As discussed in Section 3.6.1, predictions involved in SLE are resolved locally without any global information, other than the cache coherence information, being exchanged across multiple processors (even though SLE is an optimization that affects executions on multiple processors). While SLE elides locks and thus does not perform lock acquisitions, *validating a successful lock-free execution of a critical section does not require a lock acquisition*. The success is determined by simply observing the local memory interface and the absence of any misspeculation events (such as data conflicts) by the end of the critical section is sufficient validation of success. Thus, the validation latency for SLE in the event of successful speculation is essentially nonexistent.

3.8.2 Cache coherence protocols

We revisit the discussion about ownership-based cache coherence protocols (Section 2.1.2). When a data block not present in the cache is accessed, the cache coherence protocol is triggered and the block is brought into the cache. Writing the data block requires the cache to have exclusive write permissions for the block. This is done by invalidating all shared copies of the block in other caches. Once other caches have been invalidated, the local copy can be modified if necessary. Exclusive permissions force other processors to request the latest architecturally correct data copy from the exclusive owner of the block.

The above protocol functionality provides us with two capabilities. First, accessed data is easily tracked by local caching. Second, data conflicts are detected trivially—writes to shared data trigger invalidation messages to sharers, and requests to exclusively owned blocks are automatically forwarded to the exclusive owner. The coherence protocol typically tracks cache blocks rather than individual words of the cache block. As a result, the information tracked is conservative because of the potential for false sharing. While the information may not be precise it will be correct because it is conservative. This is discussed in detail later in Section 3.12.3 where we discuss SLE's interactions with false sharing.

3.9 SLE implementation

We now show how SLE can be implemented using well understood and commonly used techniques. Assume the locking algorithm discussed in Section 3.5 for the discussion in this section. Other locking algorithms are discussed in Section 3.11.4. The architecture provides the load-linked/store-conditional synchronization primitives for implementing locks.

Four aspects of implementing SLE are:

1. Identifying speculation regions[8] and initiating speculation
2. Speculative execution and buffering speculative state
3. Committing speculative state
4. Detecting and handling misspeculation conditions

[8] Ideally this corresponds to a critical section. However, because the processor core does not know whether a region is indeed a critical section, we refer to the region as a speculation region.

3.9.1 Identifying speculation regions and initiating speculation

Once the start point of a speculation region is identified, speculative execution mode is entered. The mode is exited when the region end is detected. In this section we discuss an implementation for identifying these regions, identifying the memory operations in these regions, and actions performed when speculation is initiated.

3.9.1.1 Identifying start and end points

As discussed in Section 3.6, SLE looks for silent store-pairs for identifying regions for speculation. Detailed discussions about various ways to identify such regions can be found in Section 3.6.2 and here we discuss one implementation we use in our experiments.

Region start. The start of the speculation region is identified by looking for a candidate store instruction predicted to be the first store of the silent store-pair to be elided. Once the candidate store instruction is decoded, a confidence prediction table is consulted to determine whether the store elision should be performed. The confidence table records among other information, a history of success in correctly identifying this store to belong to a silent store-pair, and a successful elision. This confidence prediction is mainly to identify conditions where store-pair elision is not successful and may hurt performance due to frequent conflicts or due to resource and other constraints. If the prediction is made to apply the store-pair elision, then the processor enters speculation mode when this elided store is speculatively retired. Speculative retirement here means instruction retirement in SLE mode. All uniprocessor program order retirement rules are maintained.

Since only the store of the lock acquire is elided, the instructions in the critical section execute normally. The processor core is unaware of a spin loop of the lock acquire algorithm. For example, if the lock is already held by another thread, the lock acquire algorithm spins (by performing load operations) and waits for the lock to be released. Under SLE, the spin would also occur because SLE is only concerned with the store operations associated with the lock acquire and release and treats load operations normally. Thus SLE will automatically not enter speculation mode. Similarly, if the load operation of the test returns a held lock, the algorithm may do something else (such as yield or execute other code sequences) and under SLE the same sequence would also occur. Remember this will only happen if SLE could not be applied otherwise due to lack of exploitable concurrency and some other thread then acquired the lock.

The older and newer value of the location to which the store has been elided, is recorded in a hardware structure along with the address of the location. This is done to detect and handle silent store-pairs. We discuss handling multiple silent store-pairs later in Section 3.10 where we discuss nested critical sections.

While every store instruction can be considered while determining the starting point of a speculation region, to keep the confidence table size small one implementation may use only the store-conditional instruction as a candidate for the first store of the predicted silent store-pair. This is because lock acquires are often implemented using the load-linked/store-conditional instructions or some similar atomic read-modify-write primitive.

Region end. The end of the speculation region is identified by the second store of the silent store-pair. This is done by observing the retirement stream and identifying a store operation that completes the silence prediction of the two stores of the silent-store pair to be elided. Every store instruction is analyzed because a lock release operation may be implemented using a simple store operation (i.e., not a store-conditional or any such special primitive).

The identified region may not actually correspond to a critical section since the hardware does not have semantic information. However, a correct execution will nevertheless be guaranteed—the instructions between the two elided stores will simply appear to be executed atomically.

3.9.1.2 Identifying speculation region memory operations

Memory operations that are part of the predicted speculation region (critical section) must be identified. These operations correspond to all memory operations speculatively retired between the start and end regions of atomic execution. By identifying these operations, data accesses within the lock-free transaction are tracked and any data conflicts detected. Again, compiler support can be used for identifying these operations but we focus on hardware-only techniques in this discussion.

Consider Figure 3-10. The processor reorder buffer is conceptually shown with the head of the reorder buffer to the right. For ease of explanation, assume instructions in the reorder buffer are also the instructions in the instruction window of the processor core. Assume the critical section is identified by *region_start* and *region_end* and is somewhere in the instruction window. In

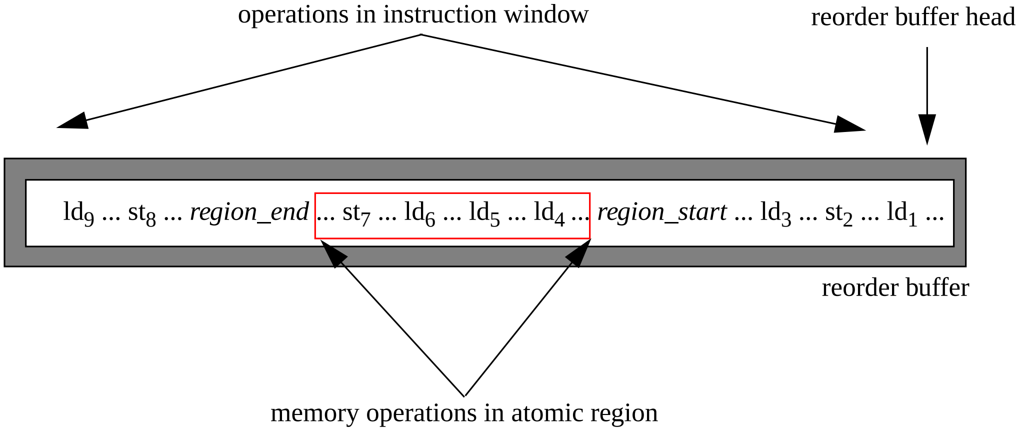

operations in instruction window
reorder buffer head
ld9 ... st8 ... region_end ... st7 ... ld6 ... ld5 ... ld4 ... region_start ... ld3 ... st2 ... ld1 ...
reorder buffer
memory operations in atomic region

Figure 3-10: Identifying memory operations within a critical section. *In a modern out-of-order processor, memory operations may issue from anywhere in the window. The processor has no clear way of identifying the operations that indeed belong to the critical section. For example, the processor cannot distinguish between operations* ld_3 *and* ld_6*.*

out-of-order processors, loads from anywhere in the window may issue in any order. Thus, the core has no clear way of differentiating between say ld_3 and ld_6.

A simple and conservative way to identify the operations within the critical section (i.e., ld_4, ld_5, and ld_6) is by assuming all memory operations issued while a critical section is predicted to be in the instruction window. Thus, when *region_start* is decoded, any subsequent memory load operations are assumed to be part of the critical section. For example, assume the current instruction window is as shown and *region_start* has not retired (i.e., SLE mode has not been entered), and ld_3 operation is issued to the memory system. The processor conservatively assumes ld_3 as part of the critical section even though clearly it is not. Thus, the data set is dilated due to the out-of-order issue characteristics of modern processors.

Alternatively, a bit can be associated with every load operation in the core. Since instructions are decoded and dispatched in program order, the bit for the load operation is set only if a *region_start* has been decoded and dispatched. This helps in differentiating precisely between operation ld_6 and say operation ld_3. Since the operation will only retire (speculatively) when SLE mode has been entered, at that time the bit is checked. If the bit is not set and the instruction is retiring in SLE mode, the instruction must be re-executed. This check is required to handle situations where arbitrary goto code sequences may be present jumping into critical sections without the lock acquire sequence. A disadvantage of this approach is that the bit must be carried along the processor core pipeline with the instruction. An advantage is that the load data set is precisely identified. Store operations do not have the issue of dilation.

3.9.1.3 Actions in initiating speculation

Once a candidate store is ready for speculative elision, the processor enters SLE speculation mode. Before entering this mode, the processor register state must be stored for recovering in the event of a misspeculation. Only a single register checkpoint is typically sufficient. Store-pair elisions simply eliminate write requests to the memory system and prevent the values of the stores from being observed by the other threads whereas the checkpoint serves as a recovery point in the event that the atomic execution of the speculation region (which may consist of multiple nestling in the form of multiple silent store-pairs) could not be successful. At this point, we know the regions for speculation, and the memory operations predicted to be part of the region. For ease of discussion, we consider single nesting even though multiple store-pair elisions may be performed. We discuss handling nested critical sections later in Section 3.10. In short, multiple silent store-pairs may be elided automatically using little additional hardware.

3.9.2 Speculative execution and buffering of speculative state

During SLE, all instructions (included the first elided store discussed in the above section) are speculatively retired—they are not committed to architectural state until after SLE successfully completes. Uniprocessor retirement rules are maintained; i.e., an instruction is speculatively retired only if it is determined to be correct according to the microarchitecture specification. Two aspects to be handled during speculative execution are: buffering speculative register state and speculative memory state. We discuss these two aspects in the following sections.

3.9.2.1 Buffering processor register state

Most modern microprocessors support speculative execution where processor register state is speculatively modified. In the event of a misspeculation, the architecturally correct processor register state is restored and any speculative updates are discarded. Examples of such techniques are branch-prediction-based speculative execution where control flow prediction is made and execution is based on the prediction. If a branch mispredict is detected, the architected register state prior to the incorrect prediction is restored.

In the simplest SLE implementation, only a single restoration point is required. This point corresponds to the architected state just prior to entering SLE mode independent of the nesting of critical sections. A single checkpoint is sufficient and optimizations for finer checkpoints can be considered if necessary. Multiple checkpoint optimizations are however not necessary for handling and eliding nested critical section locks.

Smith and Pleszkun [153] discuss various schemes for buffering speculative updates to processor register state such as history buffers, future files, reorder buffers, etc. Techniques discussed in this section are well understood and well studied in literature and have been proposed by researchers for optimizations other than SLE. Two techniques applicable to SLE we discuss are: using the reorder buffer, and using a register checkpoint approach.

Using the reorder buffer (ROB). The ROB can be used to buffer all speculative updates to registers in a manner similar to branch-prediction-based speculative execution techniques. The same mechanisms for branch prediction are employed (except in this case, the speculative execution is based on a silent store-pair prediction rather than a branch prediction) and the recovery mechanisms are identical. Instructions (including loads and stores) are speculatively retired but not removed from the ROB until after SLE is validated.

The disadvantage of using the ROB is the limitation on the size of the critical section (the dynamic instruction count of the critical section) that can be speculatively executed. Additionally, the commit rate of the critical section at the end of SLE validation is limited by the commit bandwidth of the core: for N_{dyn_inst} dynamic instructions in the critical section speculatively executed and N_{commit_bw} being the number of instructions that the core can retire in a single cycle (assuming all instructions are treated equal at retirement time), the core takes at least (N_{dyn_inst}/ N_{commit_bw}) cycles to retire the speculative execution to architectural state.

Using a register checkpoint. The checkpoint may either be of the register state itself or of the register dependence maps. The latter case may place restrictions on how physical registers are freed. Prior to entry into SLE mode, an appropriate checkpoint is created. On a misspeculation, the checkpoint is restored and the instructions are re-executed similar to a branch misprediction recovery event. Instructions (including loads and stores) on being speculatively retired can be removed from the ROB.

Using a checkpoint approach frees the limitation of the critical section size (dynamic instruction counts) on the reorder buffer size because in the event of a misspeculation, an architecturally correct register state is available for recovery in the form of a single checkpoint.

3.9.2.2 Buffering processor memory state

Although most modern processors support speculative execution of load instructions, they do not retire store instructions speculatively; store instructions are only removed from the reorder buffer once their program order and memory consistency requirements have been maintained and their values are written to the memory system (including the cache) once the stores are known to be non-speculative. SLE, like other proposals for speculation [48, 61, 154, 156], uses speculative store retirement.

Use the processor write buffer. In the proposed implementation, the processor write buffer, lying between the processor and the level-one cache, is augmented to buffer speculative store values. While the write buffer is used to store speculative memory updates, the speculative values are not committed to the cache and the lower memory hierarchy until after SLE is successfully validated. On a misspeculation, the speculative memory updates in the write buffer are discarded.

An advantage of the write buffer approach is that an architecturally correct value for an address (to which a speculative store has been retired) is always available in the processor cache in the event of a misspeculation and the cache does not require support for speculative store buffering.

As an additional benefit, under SLE, speculative writes can now be merged in the write buffer, *independent* of the memory consistency model. For example, while sequential consistency and processor consistency prevent some write operations from being merged in the write buffer[9], under SLE the merging is legal. This is possible because, for successful speculation, all memory accesses are guaranteed to appear to complete atomically. The write buffer size limits the number of unique

[9] This is true under conventional implementations. One can construct complex implementations where such support may be possible.

cache blocks modified in the critical section and does not restrict the dynamic number of store instructions executed in the critical section.

Use the processor cache. Alternatively, the speculative memory state can be exposed to the processor caches. Other proposals have been made for allowing caches to buffer speculative state [42, 52, 61, 155] and these proposals can be adapted for use in SLE. The requirement, as is the case for any speculative technique that allows stores to retire speculatively, is that an architecturally correct value of the speculatively modified cache block must be available in the event of a misspeculation. This can be achieved by using a special buffer below the level-one cache to store the architecturally correct values or using the level-two cache for doing so [155].

3.9.3 Committing speculative state

The discussion in this section focuses on committing register state and committing speculative memory state when speculation is successful.

3.9.3.1 Committing processor register state

If the reorder buffer approach is used to implement SLE, the processor core retires instructions at its peak commit bandwidth and the architected registers are written.

If the register checkpoint approach is used, the processor simply discards the register checkpoint created at the start of speculation. Since under speculative retirement, the registers are already being written to, the current register state is marked as being non-speculative (with respect to SLE). Note, the processor may perform speculative execution (e.g., branch prediction and data value prediction driven speculative execution) with and without SLE.

3.9.3.2 Committing processor memory state

Committing memory state requires ensuring that buffered speculative stores are committed and made visible *instantaneously* to the memory system—they must appear to execute atomically. We exploit cache coherence protocols for doing so. Processor caches have two aspects: 1) data, and 2) state. The cache coherence protocol determines state transitions of cache block state. Importantly, these state transitions can occur speculatively as long as the data is not changed speculatively. This is how speculative loads and exclusive prefetches (operations that bring data in exclusive state into caches) are issued in modern processors. We use these two aspects in performing atomic memory commit without making any change to the cache coherence protocol.

Using non-binding exclusive prefetches. When a speculative store is added to the write buffer, a non-binding request for exclusive ownership of the cache block is sent to the memory system. The request is non-binding because the block, once brought into the local cache, remains exposed to the coherence protocol. The request initiates pre-existing state transitions in the coherence protocol and brings the cache block into the local cache in the exclusive state. Note the *cache block data is not speculative*—speculative data is buffered in the write buffer. When the critical section ends (i.e., the second elided store is encountered), all speculative entries in the write buffer will have a corresponding block in exclusive state in the cache, otherwise a misspeculation would have been triggered earlier. At this point, the write buffer is marked as having the latest architectural state.

Draining the speculative write buffer. The instantaneous commit is possible because the process of marking the write buffer as having the latest state involves setting one bit—exclusive permissions have already been obtained for all speculatively updated and buffered blocks. One approach is to add functionality to the write buffer of being able to source data for requests from other threads. Alternatively, the write buffer can be lazily drained into the cache as needed or temporarily stalling the processing of incoming requests from the lower memory hierarchy while the write buffer is drained. Note, however, that a read to such a location from another processor must be serviced perhaps with a small delay—with the recently modified value. During the draining of the write buffer entries into the cache, no deadlock possibilities exist because all required exclusive permissions have been obtained in the cache for the appropriate cache blocks. The draining process must be atomic and during the process, any interrupts must be delayed until after the draining is complete. All external requests to the cache must also be delayed. This delay is however bounded because all blocks are available in the cache in appropriate state (no miss will occur during the draining process) and the delay is a function of the number of speculative entries in the write buffer and the latency of writing a block into the level-one cache.

If all appropriate blocks are not yet in the local cache in appropriate state (exclusive or shared), then speculative execution can proceed until the blocks corresponding to the write buffer are available in appropriate state.

3.9.4 Detecting and handling misspeculation conditions

We now discuss conditions under which SLE may trigger a misspeculation and mechanisms to handle such situations.

3.9.4.1 Misspeculation conditions

A misspeculation is triggered only if misspeculation conditions occur while the processor is in SLE mode. The misspeculation conditions under SLE are 1) atomicity violations, and 2) violations due to limited resources.

An atomicity violation potentially occurs when at least two threads perform competing accesses to a common memory location and one thread is in its speculative lock-free critical section execution mode. This may prevent an atomic commit of memory operations and thus prevent serializability from being maintained.

Resource-limitation-induced violations occur when the processor in SLE mode cannot buffer speculative state. This includes, buffering speculative memory updates, or an inability to maintain book-keeping information (e.g., tracking data locations accessed by the speculative transaction) to detect atomicity violations.

3.9.4.2 Atomicity-violation induced misspeculation

Atomicity violations are detected using the cache coherence protocol. As discussed earlier (Section 2.1.2), cache coherence is a mechanism to propagate memory updates to other caches and make memory operations visible to other processors. Invalidation-based coherence protocols guarantee an exclusive copy of the memory block in the local cache when a store is performed. Since most modern processor systems already implement some form of invalidation-based coherence control as part of the cache hierarchy, the mechanism for detecting conflicts (i.e., among simultaneous operations to a given memory location, at least one is a write operation) already exists on most processors. Using the coherence protocol to detect data conflicts and using the coherence granularity as the granularity for sharing results in false positives due to false sharing. These situations are treated as true sharing by SLE and we discuss this later in Section 3.12.3.

A mechanism to track data locations accessed within the speculative lock-free critical section, including the lock variable itself, is required. We discuss two such mechanisms: 1) using the processor load/store queue, and 2) augmenting the local cache to record blocks accessed.

Using the load/store queue. Some modern processors such as the MIPS R10K [170] and the Intel Pentium 4 [70] allow aggressive implementations of memory models by allowing loads to issue speculatively similar to the proposal by Gharachorloo et al. [45]. The speculatively issued loads are tracked using the load/store queue (LSQ). If an invalidation is received from the coherency mechanism and the invalidation reaches the core, the LSQ is snooped to determine whether a memory consistency violation may have occurred. If SLE is implemented using the ROB approach discussed above, then the LSQ itself could be used to track speculatively issued memory operations (both loads and stores). The additional functionality would be to snoop the LSQ if external requests are received for data blocks that have speculatively been modified during SLE.

If the register checkpoint approach is adopted for implementing SLE, then the LSQ alone cannot be used to track atomicity violations because memory operations may speculatively retire and be speculatively removed from the ROB (and thus the LSQ). Note, this is a greater concern for loads because store values are buffered in the write buffer and hence they are implicitly tracked (the write buffer must be snooped in such cases if the LSQ is used for tracking data sets). For the checkpoint approach, a table at the cache could be employed that records all addresses accessed and is matched in parallel with a cache lookup to detect conflicts. Alternatively, the cache tags themselves could be extended to track blocks accessed. We discuss the latter approach next.

Augmenting the local cache. Each cache block is augmented with a *speculative access bit.* Every memory access issued by the processor core and predicted to belong to the critical section sets the speculative access bit for the corresponding cache block. The identification of memory accesses predicted to belong to the critical section was discussed in Section 3.9.1.2. When an external request is received by the coherency controller, the local cache is already snooped (to maintain cache coherence). The additional bit is also tested in parallel with the cache tag lookup. A data conflict occurs if any of the following occurs:

1. An external invalidate request to a cache block with a set speculative access bit.

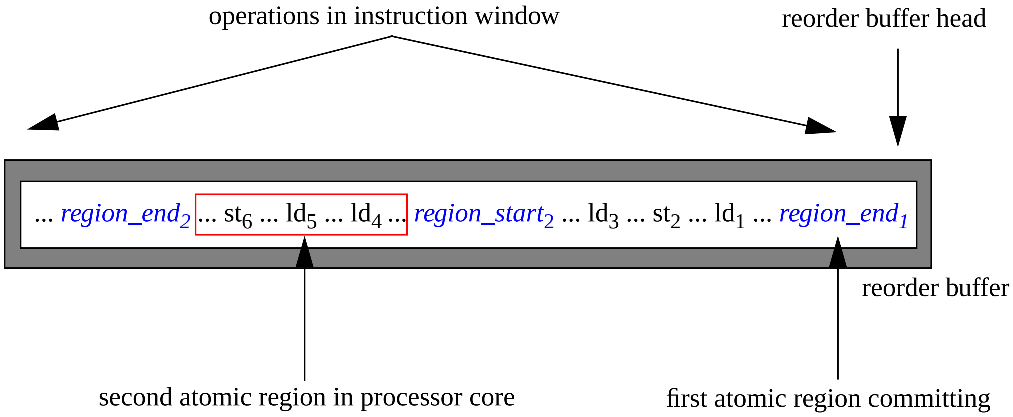

operations in instruction window
reorder buffer head
... region_end2 ... st6 ... ld5 ... ld4 ... region_start2 ... ld3 ... st2 ... ld1 ... region_end1
reorder buffer
second atomic region in processor core
first atomic region committing

Figure 3-11: Handling multiple critical sections in instruction window. *Out-of-order processors allow memory operations to be simultaneously issued from multiple critical sections if these critical sections are present in the instruction window at the same time. In such situations, the speculative access bits may not be reset at the end of the first critical section because these accesses may also correspond to the subsequent uncommitted critical sections in the instruction window.*

2. An external read request to a cache block to which the processor has speculatively retired a store (i.e., a write has been performed to that location by the processor) and the block's speculative access bit is set.

In condition 2, the speculative value is buffered in the write buffer and the cache is only used to track whether a write has been performed to the location. Alternatively, the write buffer may also be snooped to determine such conflicts. In a MOESI cache coherence protocol (Section 2.1.2), condition 2 corresponds to a cache block in the M (modified) state.[10]

On misspeculation and processor commit events, the speculative access bit may be reset for all cache blocks using a technique such as flash invalidation [106]. The speculative access bit may be reset only if the processor determines it has not issued any subsequent memory operations belonging to a later uncommitted critical section. Consider Figure 3-11. The first critical section retiring is identified by *region_end$_1$* at the head of the ROB. However, at this moment, another

[10] This is not quite accurate. The discussion here also includes the case where the cache block might have been in the M state before the processor entered speculative execution mode. To differentiate between such cases, a second bit (speculative dirty bit) can be used to track whether the block has been written to within the critical section.

atomic region is in the processor core identified by *region_begin*$_2$ and *region_end*$_2$. If this is an out-of-order core implementation, memory operations from the second atomic region may also have issued and may have set the speculative access bit for the corresponding cache block. Thus, the speculative access bits must not be reset if the first atomic region commits and another is predicted to be in the processor core. This is mainly because the processor core has no simple way to differentiate between memory operations within a critical section from those issued outside critical sections. Further optimizations are possible but we do not discuss them.

The above scheme is independent of the number of cache levels in the local hierarchy because all caches maintain coherence and any requests that require coherence state transitions (these also correspond to the requests that trigger conflicts) are propagated to all coherent caches automatically by the existing cache coherence protocols.

3.9.4.3 Resource-constraint induced misspeculation

Limited resources *may* force a misspeculation. Since we use the cache coherence protocol to track data accesses and detect conflicts, situations that make it impossible to track data accessed and detect conflicts may result in a misspeculation. Note that while resource constraints are a fundamental limitation, they can be made small (arbitrarily unimportant) by providing additional resources.

Common conditions for possible misspeculation are:

1. *Finite cache size or associativity.* If the cache is used to track the lock and data accesses within a critical section, the finite size of the cache restricts the data set size that can be tracked speculatively. The associativity of the cache also places a limit because conflict misses force evictions of cache blocks. Well known and well understood techniques for handling such situations exist, such as victim caches [79]. Victim caches are small, fast, fully associative structures that buffer cache blocks evicted from the main cache due to conflict and capacity misses. The victim cache can be extended with a speculative access bit per entry to track the cache blocks. The issue of sufficient buffering resources is an engineering decision involving a trade-off. While more resources can be provided, caches today are sufficiently large to buffer most critical sections.

2. *Finite write buffer size*. Since the write buffer is used to buffer speculative memory updates, its size restricts the number of static addresses that can be written to within a critical section. Note, since stores are retired (speculatively) in program order the write buffer has the precise stores within the critical section. This is because once in SLE mode (when the candidate silent store-pair is elided), the processor sends all store updates to the write buffer. Since writes are merged in the write buffer and memory locations can be rewritten within the write buffer (because atomicity is guaranteed), the number of unique cache blocks written to within the critical section is limited by the size of the write buffer.

3. *Finite reorder buffer size*. The ROB size restriction exists only if the reorder buffer approach for implementing SLE is adopted. This works well for small critical sections but may not work for larger critical sections. The size of the critical section here is determined by the number dynamic instructions executed and retired within the critical section.

4. *Uncached accesses and other such events.* These are events that occur within a critical section where the processor cannot use the cache coherence protocol for tracking accessed memory locations. Examples of such events are uncached memory accesses or certain operating system events, such as I/O, that may prevent tracking data accesses.

5. *Instructions that cannot be undone.* Instruction set architectures may have certain instructions where their effects cannot be undone. These instructions cannot be executed speculatively. Such instructions may involve operations that force caches to be flushed. Included here are memory mapped I/O operations that may require immediate external visibility.

6. *Expiration of operating system scheduling quantum.* If the time to execute a critical section is longer than the operating system scheduling quantum assigned for that thread, the thread will be descheduled by the operating system. An operating-system-induced descheduling event results in a misspeculation being triggered.

Misspeculation conditions need not always trigger misspeculation. We have listed common misspeculation conditions above. If any situation arises that cannot be handled easily, the lock can always be acquired and a correct execution guaranteed. A misspeculation need not be triggered for all of the conditions listed. For example, for cache-size-limitation-induced and write-buffer-size-limitation-induced misspeculation conditions, one approach is to stall the processor and allow the elided store (which was buffered at the start of SLE mode) to go to the memory system. This essentially emulates the acquiring of the lock (without having to misspeculate, and re-execute the lock acquire code sequence). If the write operation succeeds, the processor state can be assumed committed. This is because writes to a given location are serialized by the cache coherence protocol (Section 2.1.2) and only one such write succeeds while the invalidation request generated by this write would trigger a misspeculation in all other processors forcing them to restart. The current processor can then find itself within its critical section (by virtue of having successfully acquired the lock) and all speculative work performed is committed.

3.9.4.4 Handling other misspeculation conditions

An advantage of SLE is that in the event of unexpected conditions, the lock can always be acquired and a correct execution guaranteed along with forward progress. If silent store-pair predictors are employed, then no guarantee can be made regarding whether the second store of the silent-store pair will be encountered. In such an event, execution may proceed for a while before the processor runs out of buffer space. However, the problem may be more acute where one can consider a malicious critical section. In such a critical section, the thread acquires the lock, and goes into an infinite loop. In other words, the program is broken. However, SLE must reproduce the error faithfully. To handle such situations a *time-out* mechanism is employed. The time-out mechanism tracks the number of dynamic instructions executed within a critical section (on the order of 10s of thousands of instructions and will typically be less than the instructions that typically execute in a given operating systems scheduling quantum).[11] Once a time-out threshold is achieved, a misspeculation is triggered and the execution is restarted. After a preset number of restarts (the restart threshold), the lock is acquired, thus guaranteeing a correct execution with forward progress even under unexpected and unknown conditions.

[11] The number chosen here is somewhat arbitrary but is one way to guarantee that the processor does not speculate forever.

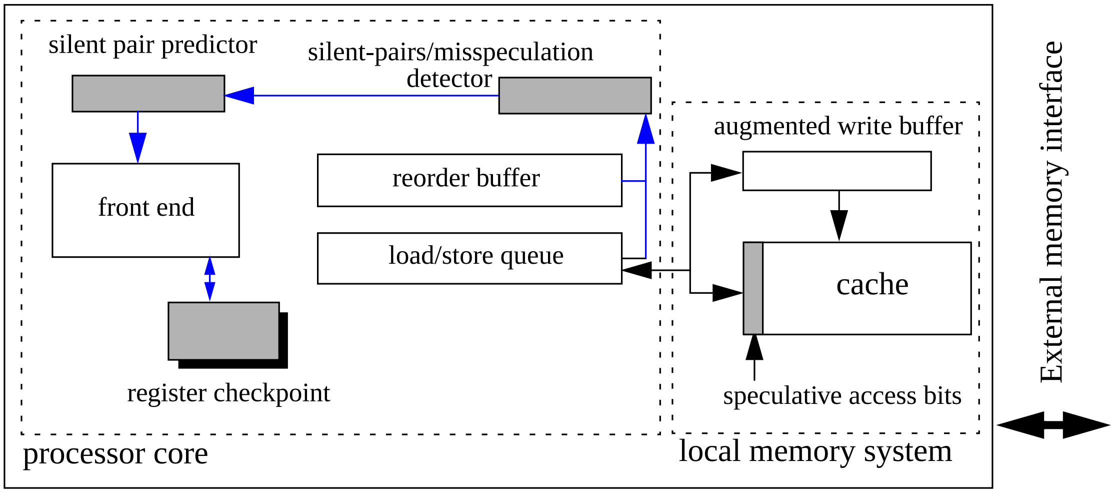


Figure 3-12: A microarchitectural implementation of SLE. *The additional hardware is shown shaded. All changes are made within the processor core and additional bits for the level-one cache.*

3.9.4.5 Recovering from misspeculation

On a misspeculation, the execution restarts. The architected register state is restored and the speculative write buffer entries are discarded. All speculative access bits in the cache are reset if necessary.[12]

SLE may be reapplied. However, the processor core may decide to acquire the lock once the restart threshold is reached or certain conditions (as discussed earlier) make it necessary to acquire the lock. If a thread acquires the lock on a restart, the lock variable (cached in shared state) is written to. This automatically triggers the coherence protocol and invalidate messages are sent to all sharers of the cache block containing the lock variable. This is already handled by the conventional cache coherence protocol. Thus, all other speculating threads are automatically informed if the lock is acquired, forcing them to misspeculate if in speculation mode.

Figure 3-12 shows a design point for SLE. The SLE additions are shown shaded. All modifications are within the processor complex.

[12] See Section 3.9.2 and Figure 3-11 for the situations involving multiple critical sections simultaneously in the reorder buffer where the speculative access bits must not be reset.

3.10 SLE and nested critical sections

We now discuss how nested critical sections are handled under SLE. In Section 3.10.1, we first discuss the simplest approach of handling nesting by essentially not eliding nested locks. Then we show how modest hardware can be used to apply SLE to properly nested critical sections in Section 3.10.2 and then discuss how SLE interacts with improperly nested critical sections in Section 3.10.3.

3.10.1 Trivially handling nested critical sections

The simplest SLE implementation would apply elision to a single store-pair. Thus, it would handle only one lock at a time for elision. If the processors enters SLE mode and subsequently encounters nested locks (at least conceptually because the processor does not quite have a notion of locks), these nested locks are treated as normal memory operations. In other words, the write operations corresponding to the lock acquire and releases are performed to these nested locks. If another thread reads the value of the nested lock, it will trigger a misspeculation because the nested locks are treated as normal data operations. If the outermost nesting cannot be elided because of resource limitations, SLE will acquire the outermost lock and then apply SLE to the next inner level—applying SLE one-at-a-time.

3.10.2 Handling properly nested critical sections

We now discuss how SLE can be easily extended to handle nested critical sections. Consider Figure 3-13. The solid black circles correspond to lock acquire, the crossed solid black circles correspond to lock releases, and the white circles correspond to accesses to data protected by the locks. Further assume the value 1 corresponds to a held lock and the value 0 corresponds to a free lock. The program execution and commit stream as seen by the processor is shown in the top part of the figure with time progressing to the right. Multiple locks are acquired and released in the sequence. The same execution is expanded in detail in the lower part of the figure. For convenience, assume the number of nesting levels, *i*, is 4. Four horizontal lines are shown corresponding to the execution of each nesting level. The white circles on each line correspond to the data

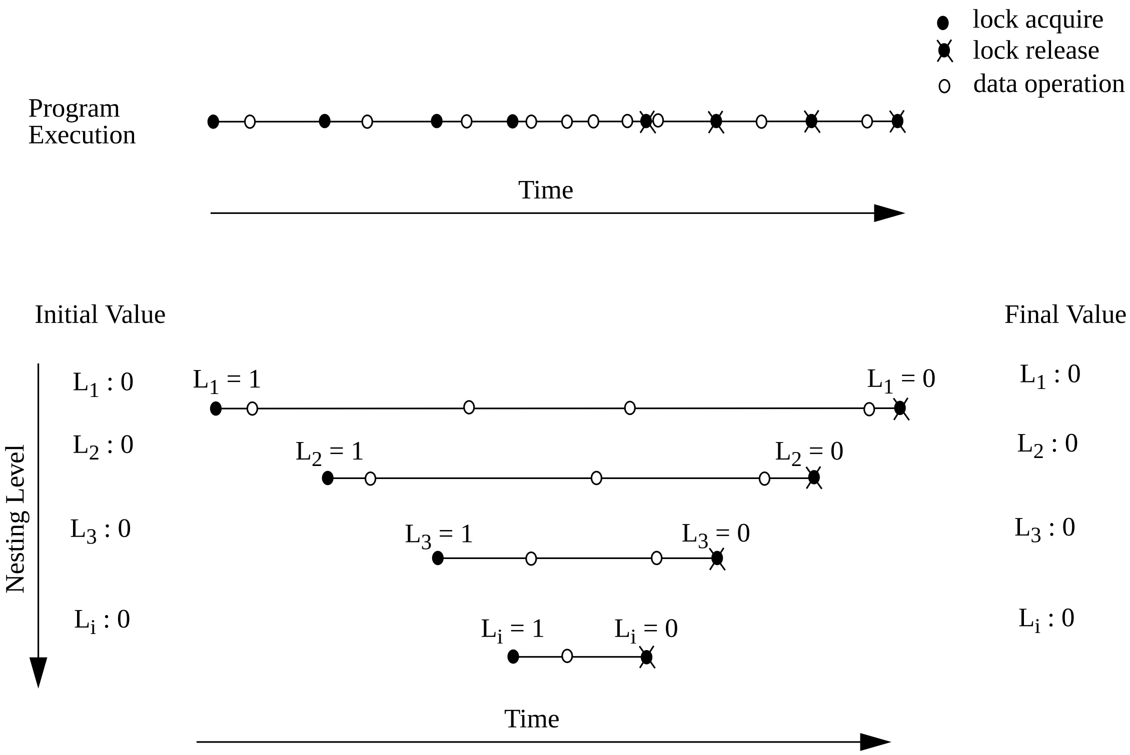


Figure 3-13: Handling properly nested critical sections. *The program execution is shown above and the execution is expanded in the lower part of the figure. The various levels of nesting are shown and the initial and final values of the nested locks are also shown. A simple hardware stack mechanism is employed to apply store-pair elision to every level of nesting.*

accesses protected by the appropriate lock. While the programmer may be able to determine which data is protected by which nested lock, this is not possible to determine dynamically in the interleaved execution stream as seen by the processor. To be conservative, the outermost lock L_1 protects all data accesses below itself; i.e., level L_i protects all data accessed by level L_i and below.

Assume L_i corresponds to the lock at each level. The initial and final values of the variable L_i are shown. The solid black circle and the crossed solid black circle on each line make a silent store-pair. Multiple silent store-pair elisions can be performed if sufficient hardware is provided to track the multiple silent store-pairs. A simple stack mechanism is used to track silence if the critical sections are properly nested. Even if multiple silent store-pairs are elided, only a single checkpoint of register state is necessary.

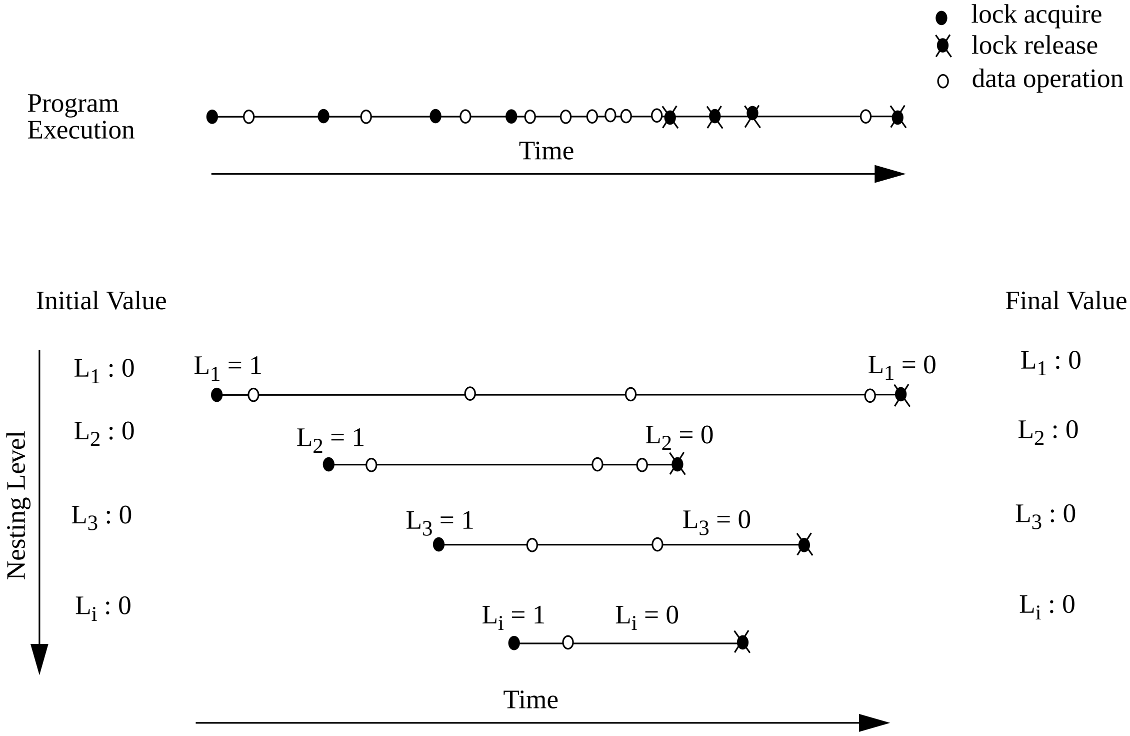


Figure 3-14: Handling improperly nested critical sections. *Although the inner locks are not properly nested, the overall execution shows proper nesting because silence of lock variables is maintained.*

3.10.3 Handling improperly nested critical sections

Figure 3-14 shows an example of improperly nested inner critical sections. These can also be handled as long as silence of all elided stores is maintained.

Note, store-pair elision only eliminates the write requests to the memory system and is unaware of the notion of nesting levels. If a silent store-pair region cannot be identified, store elision cannot be performed. Further, if the data accessed by level L_1 cannot be buffered in the local cache hierarchy, its lock is acquired but the inner levels are still candidates for elision.

A complex critical section nesting is shown in Figure 3-15. Here, a new lock is acquired before the earlier lock is released and thus improper nesting is present throughout the execution. This is similar to a B-Tree walking algorithm [12, 43]. If only a single level of silent store-pair is exploited, then every second lock is acquired. Thus the outermost is not acquired but the next level

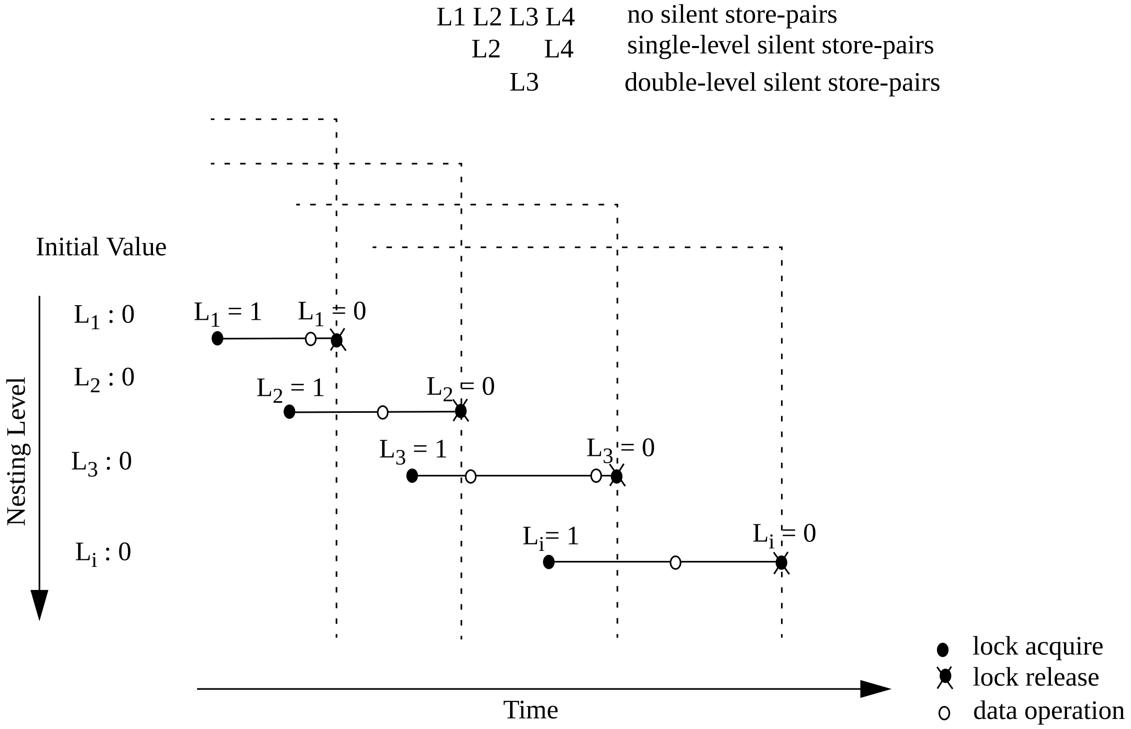


Figure 3-15: Handling complex improperly nested critical sections. *In this example, every level is improperly nested.*

lock is acquired and so on. As the number of levels to be exploited increases, so does the pressure on local caching resources. In the case of B-Tree locking, the root is not locked in our example.

Note, additional hardware is required to track the nesting. However, if the nesting level for the silent store-pairs crosses the hardware provided limit, elision is not performed and the stores (corresponding to the predicted lock operations) are executed if necessary, and thus a correct execution is always guaranteed. Again, a correct execution results because semantically the critical section is executed in an atomic manner.

3.10.4 SLE and recursive critical sections

In recursive critical sections, a thread may repeatedly acquire a lock but increments a counter field associated with the lock. Here also there is silence but across a range rather than a pair. The silent store-pair is separated by a series of increments in between. This may be handled if the silence of store-pairs is maintained but hardware resources are required to track the level of recursion. We leave adapting SLE for recursive critical sections as future work.

3.11 SLE interactions with software

In this section, we discuss interactions between SLE and software. We first discuss the forward progress guarantee that SLE maintains (Section 3.11.1). We also discuss how SLE does not change program semantics (Section 3.11.2), SLE interactions with programs that make timing assumptions (Section 3.11.3), and SLE interactions with various synchronization algorithms (Section 3.11.4). Finally we discuss SLE interactions with thread scheduling (Section 3.11.5).

3.11.1 SLE and forward progress

SLE always guarantees forward progress (similar to the implementation of the locking algorithm itself) because the lock can always be acquired if necessary. SLE does not change this guarantee (or lack thereof) or make it any worse.

3.11.2 Interactions with program semantics

In this section, we discuss how SLE does not rely on program semantics for a correct execution and how SLE does not change any program semantics.

SLE does not rely on program semantics. We now discuss why SLE guarantees a correct execution even in the absence of precise semantic information from the software and independent of nesting levels and memory ordering. As mentioned earlier, SLE involves two predictions that are resolved locally:

1. On a store, predict that another store will shortly follow and undo the changes by this store. The prediction is resolved without stores being performed but it requires the memory location (of the stores) to be monitored. If the prediction is validated, the two stores are elided.
2. Predict that all memory operations within the window bounded by the two elided stores can occur atomically. This prediction is resolved by using preexisting cache coherence mechanisms.

```
       Thread 1                                 Thread 2
                                          .
                                          assert ( LOCK_IS_FREE(lock) );
                                          .
LOCK_ACQUIRE(lock);                       LOCK_ACQUIRE(lock);
.                                         .
.                                         .
assert ( LOCK_IS_HELD(lock) );            assert ( LOCK_IS_HELD(lock) );
.                                         .
.                                         .
LOCK_RELEASE(lock);                       LOCK_RELEASE(lock);
```

Figure 3-16: SLE does not change program semantics. *Two non-conflicting thread executions are shown. Both assertions in the threads evaluate true because program order is maintained even though the lock has been elided. Thread 1 observes a held lock within its critical section while thread 2 observes a held lock outside its critical section.*

The above predictions do not rely on semantics of the program (a silent store-pair predictor is used to identify loads/stores as candidates for prediction 1 but is not integral to the idea and software could alternatively provide these hints).

If another thread explicitly acquires the lock by writing to it, a misspeculation is triggered because the write will be automatically observed by all speculating threads (because the lock is also tracked in a shared state by the speculating processors). This trivially guarantees correctness even when one thread is speculating and another thread acquires the lock.

Since SLE does not rely on program semantics (and only relies on silent store-pair predictions), SLE does not require software support for correctness. However, software support can always help performance issues as discussed in a later discussion on false sharing (Section 3.12.3).

SLE does not change program semantics. Two thread executions are shown in Figure 3-16. Assume SLE is successful and the lock is elided and the two critical sections did not observe any data conflicts while executing. No timing assumptions are made among the two threads. In other words, the executions of the two threads can be arbitrarily interleaved in physical time and SLE guarantees they will be logically ordered in a serializable manner. Thread 1's assertion will evaluate to true because the speculating thread observes a held lock. It is legal for the thread while in speculative mode to read the value of the lock and it must find the value of the lock to be the value the thread “wrote” even though the processor has no permission to actually write the value to the cache. This is because program order is maintained. The thread of course knows no other thread wrote that location because the lock location is cached locally in a shared state. Thread 2’s assertion for a free lock outside its critical section will also evaluate to true because the lock has been elided and thus not actually been acquired by thread 1.

3.11.3 Interactions with programs written with timing assumptions

We now discuss the impact of using SLE on programs that make assumptions about timing properties of the multiprocessor system. An example is a situation where a programmer uses a loop (for example a `for` loop in a critical section looping for a fixed number) to introduce delay. The example is shown in Figure 3-17. Here, the programmer is making an assumption about the time to execute the `for` loops. Even though these two critical sections do not share any data structure, under conventional locking implementations, these executions would not occur concurrently because both critical sections are protected by the same lock. Under SLE, both would execute concurrently.

Note, this is valid behavior under the architectural specification. While the naïve programmer may assume such code sequences will stagger executions, such a guarantee is not provided by any implementation because multiprocessor memory ordering issues are always based on logical ordering and independent of any timing assumptions. This program segment will not provide the assumed behavior under any implementation.

Thus, SLE does not change the semantics of the operations based on timing assumptions—rather the program itself is making an incorrect assumption in such cases. Traditionally, newer implementations of processors may speed up some operations while slowing others down and thus writing code assuming certain timing properties of the system is error-prone, and hazardous.

3.11.4 Interactions with different locking algorithms

We have focused on test&set based locking algorithms (this includes both test&set, test&test&set, and their variants). To elide locks from execution streams, they must first be identified. Since processors lack instructions to unambiguously identify lock instructions, prediction is employed. Test&set-based locks are easy to identify in hardware because these locks demonstrate critical section 1

```
LOCK_ACQUIRE(lock);
.
for (i = 0; i < 200; i++);
.
LOCK_RELEASE(lock);
```

critical section 2

```
LOCK_ACQUIRE(lock);
.
for (j = 0; j < 600; j++);
.
LOCK_RELEASE(lock);
```

Figure 3-17: Critical sections written with timing assumptions. *Executing this program may give different timing results on different systems and thus is a broken program if the programmer is trying to exploit timing assumptions in the program.*

a simple silent store-pair pattern (Section 3.6.2). By doing so, SLE does not rely on the knowledge of locks and can use silent store-pair predictors

The simplicity and portability of test&test&set locks make them quite popular. Hardware architecture manuals recommend [28, 31, 54, 73] and database vendors are advised [83] to use these simple locks as portable locking mechanisms. The POSIX threads standard recommends synchronization be implemented in library calls such as `pthread_mutex_lock()` and these calls implement the test&set or test&test&set locks.

While test&set-based locks are quite popular and widely used, other locking algorithms have also been proposed and sometimes are used when the occasion demands it. Common examples of such algorithms are MCS locks and ticket locks. If a program using such algorithms are run on hardware supporting SLE relying on silent store-pair detection, these algorithms may not necessarily be easily identified but will nevertheless execute correctly. SLE aims at executing lock-free critical sections to expose and exploit concurrency. Thus, in situations where SLE is advantageous, the use of simple algorithms is recommended because silent store-pairs is a simple property to detect and exploit. Alternatively, for identifying more complex algorithms, either more hardware would be required (which would make the hardware too specialized), or the program must be written in a form exploitable by simple silent store-pair predictors.

3.11.5 Interactions with operating systems

We briefly discuss SLE interactions with thread scheduling at the operating system and user space when multiple threads are executed in the same hardware context (a uniprocessor).

Thread scheduling. If the operating system deschedules a process, certain book-keeping is performed, for example, certain registers are set, processor contexts are saved, and the address space mapping is changed. While the exact operations depend upon the underlying implementation, in general when such a context switch occurs and is detectable, a misspeculation is triggered if the processor was in SLE mode. Even if a misspeculation is not triggered, and the processor remains in SLE mode, eventually it will run out of resources and trigger a misspeculation. The address space is handled conventionally and no special support is required.

Regarding multiple user-level software threads on the same “hardware” context, if a user-level thread switch occurs (and the operating system is not involved at all), the address space does not change, and no misspeculation is otherwise triggered, then the newly scheduled thread will see a held lock (remember, the speculative values do not leave the hardware context but are visible only within the hardware context). If the new thread was outside a critical section then it would not be able to enter the critical section. The new thread could not have been inside the critical section. If the new thread accesses data without a lock, then it would be a data race in the application itself and the execution would be a legal execution. There won’t be a starvation issue because when the older thread runs again, it will complete. For two user-level threads running on the same hardware context (without a misspeculation having been triggered), one would see a held lock and thus experience blocking behavior.

If a user level thread from some other processor is scheduled on this processor, then the OS is invoked and because speculative state is never allowed to leave a processor, a misspeculation is triggered. Thus, correct execution is always guaranteed.

Yields. If a lock is held for some time, the thread spinning on the lock may invoke a `yield` command to deschedule itself and give the processor back to the operating system. This is done so as to prevent idle spinning. The hardware itself is unaware of the yield and the operating system manages the descheduling operation. While the issue of yields is orthogonal to SLE, and SLE does not change the semantics or is affected by yields, we briefly discuss yields. If SLE encounters a held lock, the test of the test&test&set algorithm will automatically spin and if necessary a yield would automatically occur. SLE only elides the store of the lock acquire and executes the other load instructions normally—these instructions may correspond to the spin loop. If SLE is successful and does not result in a lock acquisition, no yielding occurs as there is no spinning. Often yielding is useful if a long latency operation such as I/O is occurring within the critical section. We do not expect this to be a major issue because yielding is useful if a thread is in a critical section for a long period—something that often involves I/O within the critical section. In such a case, SLE cannot provide any speculative execution behavior (I/O cannot be undone) and hence we would fall back on the lock acquisition sequence. Recent work has suggested that yielding may not be a good idea for performance reasons if only memory accesses are being performed within the critical section because the overhead associated with yielding is quite high [91].

3.12 SLE interactions with hardware implementations

We now discuss SLE interactions with various implementation specific hardware features.

3.12.1 Implementation with different synchronization primitives

While SLE as discussed relies only on silent store-pair identification, lock acquire algorithms employ synchronization instructions such as the load-linked/store-conditional [28, 31, 54], swap, and compare&swap [166], among others. For load-linked/store-conditional instructions, the store-conditional is treated as a store and thus is also considered as part of the store-pair identification process. Instructions like the compare&swap are implemented as atomic read-modify-write operations. Processor cores often split these instructions into a read and a write micro-operation for efficient implementation. Thus, these instructions can also be considered as candidates for store-pair identification. The precise implementation depends upon the underlying instruction implementation.

3.12.2 Interactions with memory consistency

Informally, no memory ordering problems exist because speculative memory operations under SLE have the appearance of atomicity. Regardless of the memory consistency model, it is always correct for a thread to insert an *atomic* set of memory operations into the global order of memory operations as shown in Figure 3-4 earlier. For weakly ordered systems, a fence usually occurs at the beginning and end of the critical sections and these must still be observed. We discuss this below.

The appropriate memory consistency model is maintained for ordering the atomic critical section with operations prior to the critical section and operations after the critical section. The ordering constraints are dependent upon the underlying memory model implemented.

The store elision also works correctly because the two elided stores form a silent store-pair. Thus, these two stores can be combined with the atomic region between the two stores to form one giant atomic region. Note, all locations within the region are monitored for conflicting accesses and thus in the absence of any conflicting access, a correct atomic execution is guaranteed.

We selectively impose the requirement of atomicity for a set of memory operations. If the underlying model employs fences for ordering request, we obey these fences as defined by the model. Commonly, fences are used to force visibility of certain memory operations to the coherence protocol. Our technique maintains these conditions if necessary. While fences force operation visibility, a processor may not provide any speculatively modified values associated with these operations—the protocol transitions are independent of the cache block data.

3.12.3 Interactions with false sharing

Cache coherence protocols typically maintain coherence at the granularity of a cache block also known as the coherence granularity. Cache block fetches and invalidations are performed at the granularity of a cache block. While a larger granularity helps when good spatial locality in data accesses is present, poor spatial locality may result in a performance degradation due to false sharing. Goodman and Woest [43] coined the term false sharing to describe the situation when two processors alternately read and at least one writes different parts of the same coherency block, resulting in the block's being moved repeatedly between the two processors as if the data were shared when in fact no sharing is occurring.

Since we use the cache coherence protocol to detect data conflicts, a false-sharing-induced conflict is treated as a data conflict and thus is treated identically to true sharing, (i.e., it is correctly handled). This sometimes serializes critical sections that do not share data but the data maps to the same common unit of coherency. False sharing has performance implications even without SLE due to unnecessary memory traffic and latency. Programmers often address this performance degradation by appropriately padding the data structures to ensure data objects accessed by different processors do not lie on the same cache block. False sharing does not introduce any correctness issues with SLE.

3.12.4 SLE and hardware multithreaded processors

For implementing SLE on hardware multithreaded processors, additional hardware support is necessary because now multiple thread contexts are concurrently executing on the same core. SLE tracks cache blocks accessed using an extra bit. This bit now needs to be unique for every hardware thread executing. To support memory consistency models correctly in hardware multithreaded processors, a form of coherence activity must be locally invoked. For example, for two hardware threads T1 and T2 on a processor, if T1 writes to a cache block X, then depending upon the memory consistency model, an invalidation will need to be sent to T2 (and any other hardware threads) to ensure the memory consistency model is enforced. This will be true in any aggressive implementation of sequential consistency, processor consistency, or release consistency for a multithreaded processor. The solution is specific to the implementation but SLE can use support that will already be present for supporting aggressive memory consistency implementations.

3.12.5 Implementation-specific issues

Numerous implementation-specific issues may exist, such as certain translation look-aside buffer issues, operations that cannot be delayed or undone, certain interrupts, and special semantics for certain memory operations that result in a misspeculation. All these conditions can, as usual, be handled by simply not speculating and acquiring the lock.

3.13 Related work

Software only lock-free. Lamport introduced lock-free synchronization [95] and gave algorithms to allow multiple threads to work on a data structure without a lock. Operations on lock-free data structures support concurrent updates and do not require mutual exclusion. Lock-free data structures have been extensively investigated [17, 65]. Experimental studies have shown software implementations of lock-free data structures do not perform as well as their lock-based counterparts primarily due to excessive data copying involved to enable rollback, if necessary [5, 66].

Hybrid lock-free. Transactional Memory [66] and the Oklahoma Update protocol [158] were the initial proposals for hardware support for implementing lock-free data structures. Both provided programmers with special memory instructions for accessing these data structures. Although conceptually powerful, the proposals required instruction set support and programmer involvement. The programmer had to learn the correct use of new instructions and the proposal required coherence protocol extensions. Additionally, existing program binaries could not benefit. The proposals relied on software support for guaranteeing forward progress. These proposals were both direct generalizations of the load-linked and store-conditional instructions originally proposed by Jensen et al. [78]. The load-linked/store-conditional combination allows for optimistic atomic read-modify-write sequences on a word.

In contrast to the above proposals, our proposal does not require instruction set changes, coherence protocol extensions, or programmer support. As a result, we can run unmodified binaries in a lock-free manner in most cases when competing critical section executions have no conflict. We do not have to provide special support for forward progress because, for conflicts, we simply fall back to the original code sequence, acquiring and releasing the lock in the conventional way.

Hardware-only lock-free. To the best of our knowledge, Speculative Lock Elision [139] is the first hardware-only lock-free proposal that executes lock-based programs in a lock-free manner in the absence of data conflicts.

Database optimistic concurrency control. Extensive research has been conducted in databases on concurrency control and Thomasian [161] provides a good summary and further references. Kung and Robinson [90] proposed Optimistic Concurrency Control (OCC) as a major alternative to locking in database management systems. OCC involves a read phase where database objects are accessed (with possible updates to a private copy of these objects) followed by a serialized validation phase to check for data conflicts (to check for read/write conflicts with other transactions). This is followed by the write phase if the validation is successful. In spite of the extensive research, there are no database systems that use OCC as a concurrency control mechanism. Haerder [60] was the first to point out potential problems with OCC schemes. Mohan [124] provides an excellent discussion regarding the issues involved with OCC approaches and their shortcomings which make OCC unattractive for high performance database systems is provided by The special requirements and guarantees required by database systems, specifically for storage management, access path maintenance, recovery models, fine-granularity conflict checking, fine-grain locking, and semantically-rich lock modes [125], make OCC very hard to use for high performance. To provide these guarantees about database transactions, substantial state information must be stored in software resulting in large overheads in executing transactions. In addition, with OCC, the validation phase is serialized, thus limiting performance.

Our proposal is quite different from database OCC proposals. We are not providing an alternative to lock-based synchronization: we detect instances when these synchronization operations are unnecessary, and eliminate them. The requirements imposed on critical sections are far less strict than those mentioned above for database systems. Since we do not require re-execution or explicit acquisition of a lock to determine success, we do not have a serialized validation phase.

Using cache coherence protocols for conflict detection. Knight proposed using cache coherence protocols in the context of speculatively parallelizing sequential code [86]. Subsequently the Herlihy and Moss [66] used the same mechanism for implementing transactional memory. Gharachorloo et al. [45] used cache coherence protocols for detecting violations to memory ordering. Franklin proposed the use of the address resolution buffer for detecting data races in shared-memory multiprocessors [40].

Speculative buffering and retirement. Prior work exists in microarchitectural support for speculative retirement [48, 143] and buffering speculative data in caches [42, 52]. Our work can leverage these techniques and coexist with them. However, none of these earlier techniques dynamically remove conservative synchronization from the dynamic instruction stream.

Value prediction. Our scheme of silent store-pair elision is an extension to the *silent store* proposal of Lepak and Lipasti [111]. While they squashed individual silent store operations, we elide pairs of stores that individually are not silent but when executed as a pair are silent. The notion of silent store-pairs employed by SLE is an example of the notion of Temporal Silence recently investigated by Lepak and Lipasti [112].

3.14 Chapter summary

In this chapter we proposed Speculative Lock Elision—a microarchitectural technique to remove unnecessary serialization from a dynamic instruction stream. The key idea behind SLE involves using the cache coherence protocol to obtain appropriate permissions on the necessary cache blocks, accessing and modifying data speculatively if needed, and then providing the appearance of instantly committing the critical section by making updates visible to other processors at a single commit point.

SLE has the following key features

1. *Enables highly concurrent multithreaded execution*. Multiple threads can concurrently execute critical sections guarded by the same lock. Additionally, correctness is determined without acquiring (or modifying) the lock. No write permissions are required on the lock variable in the event of a successful speculation.

2. *Simplifies correct multithreaded code development*. Programmers can use conservative synchronization to write correct multithreaded programs without significant performance impact. If the synchronization is not required for correctness, the execution will behave as if the synchronization were not present.

3. *Can be implemented easily.* SLE can be implemented entirely in the microarchitecture, without instruction set support and without system-level modifications (e.g., no coherence protocol changes are required) and is transparent to programmers. Existing synchronization instructions are identified dynamically. Programmers do not have to learn a new programming methodology and can continue to use well understood synchronization routines. The technique can be incorporated into modern processor designs, independent of the system and the cache coherence protocol.

With SLE, the control dependence implied by the lock operation is converted to a true data dependence among the various concurrent critical sections. As a result, the potential parallelism masked by dynamically unnecessary and conservative locking imposed by a programmer-based static analysis is exposed by a hardware-based dynamic analysis.

The technique proposed does not require any coherence protocol changes. Additionally, no programmer or compiler support and no instruction set changes are necessary. The key notion of atomicity of memory operations enables the technique to be incorporated in processors without a dependence upon the memory consistency model as correctness is guaranteed because of atomic commit of memory operations.

SLE is a step towards enabling high performance multithreaded programming. With multiprocessing becoming more common, it is necessary to provide programmers with support for exploiting these multiprocessing features for functionality and performance. SLE permits programmers to use frequent and conservative synchronization to write *correct* multithreaded code easily; our technique automatically and dynamically removes unnecessary instances of synchronization. Synchronization is performed only when necessary for correctness; and performance is not degraded by the presence of such synchronization. Since SLE is a purely microarchitectural technique, it can be incorporated into any system without any changes to the underlying coherence protocols or without dependence on any system design issues.

Chapter 4

Transactional Lock Removal

SLE breaks a critical performance barrier by allowing non-conflicting critical sections to execute and commit concurrently. SLE showed how lock-based critical sections can be executed speculatively and committed atomically without acquiring locks if no data conflicts were observed among the critical sections. While SLE provided concurrent completion for critical sections accessing disjoint data sets, data conflicts result in threads restarting and acquiring the lock serially. Thus, when data conflicts occur, SLE suffers from the key problems of locks due to lock acquisitions.

This chapter proposes Transactional Lock Removal—a technique that uses SLE as an enabling mechanism but in addition provides a successful lock-free execution of lock-based critical sections in the presence of data conflicts if sufficient resources are available for buffering speculative state. TLR elides locks using SLE to construct an optimistic lock-free critical section execution (and treats the lock-free critical section as a lock-free transaction) but in addition also uses a timestamp-based conflict resolution scheme to provide lock-free execution even in the presence of data conflicts. A single, globally unique, timestamp is assigned to all memory requests generated for data within the optimistic lock-free critical section. Existing cache coherence protocols are used to detect data conflicts. On a conflict, some threads may restart (employing hardware misspeculation recovery mechanisms) but the same timestamp determined at the beginning of the optimistic lock-free critical section is used for subsequent re-executions until the optimistic lock-free critical section is successfully executed. A timestamp update occurs only after a successful execution. Doing so guarantees each thread will eventually win any conflict by virtue of having the earliest timestamp in the system and thus will succeed in executing its optimistic lock-free critical section. If the speculative data can be locally buffered, all non-conflicting transactions proceed and complete concurrently without serialization or dependence on the lock. Transactions experiencing data conflicts are ordered without interfering with non-conflicting transactions and without lock acquisitions.

4.1 Chapter roadmap

Section 4.2 provides the motivation for Transactional Lock Removal by discussing the performance and stability limitations of SLE in the presence of data conflicts. Section 4.3 presents the concepts of TLR. Details of conflict resolution policies and timestamp schemes are also discussed and the TLR algorithm is presented. Section 4.4 discusses an implementation of TLR. The section discusses mechanisms for retaining ownership of cache blocks using the coherence protocol, presents an implementation of TLR, qualitatively analyzes TLR performance potential, and studies implementation specific constraints. Section 4.5 presents implementation independent invariants and the programmability and stability issues of TLR are discussed in Section 4.6. We survey related work in Section 4.7 and summarize the chapter in Section 4.8.

4.2 Motivation

The motivation for Transactional Lock Removal lies in the critical limitation of Speculative Lock Elision—in the presence of data conflicts, the lock may have to be acquired. Data conflicts are inherent in multithreaded programs and thus numerous situations requiring lock acquisition exist. We now discuss the impact of lock acquisitions on performance and stability.

4.2.1 Performance limitations of lock acquisition under conflicts

When a data conflict triggers a misspeculation in SLE, rather than acquiring the lock, SLE may be retried a finite number of times. While retrying SLE in these situations may successfully elide locks, such a retry policy involves a careful trade-off. Applying SLE to situations where high data conflict rates occur may result in performance degradation because of coherence protocol interference among conflicting critical sections. Cache blocks may ping-pong among various caches before any thread successfully completes its critical section execution. Further, multiple threads may repeatedly restart. SLE automatically handles these situations by acquiring the lock after a certain number of retries. While this guarantees forward progress trivially, the corresponding locking overhead still limits performance.[1]

Additionally, lock acquisitions serialize execution of multiple threads thus limiting performance. While there may be opportunity to overlap some computation within the critical section with communication of the lock, the threads cannot commit until the lock acquisitions are serialized by the coherence protocol. Under high conflict conditions, such as cases where fine-grain locking is present and no concurrency can be extracted, the performance is limited to that of the underlying synchronization algorithm. Further, if dynamic, hard-to-detect concurrency is present in the application, identifying opportunity for concurrent execution is a non-trivial task [66].

4.2.2 Stability limitations of lock acquisition under conflicts

As discussed in Section 1.2, the semantic operation of writing a new value to a lock variable and of waiting for the value to change (by spinning on it) inherently limits the stability of the system. The inherent limitation stems from the *wait action* while some thread is in the critical section—a lock marked held forces other threads to wait for the lock value to be free. If the value does not change for an arbitrarily long time (the thread holding the lock may have been descheduled) or does not ever change (the thread holding the lock may have aborted), system performance is affected and the system may fail to perform as expected. Data conflicts occur in multithreaded programs and by acquiring locks in such situations, the limitations of locks are exposed.

4.3 Transactional lock-free execution of critical sections

TLR aims to achieve a serializable schedule of lock-free critical sections, even in the presence of data conflicts, where all memory operations within a critical section appear to be atomically inserted into some global order. This is illustrated in Figure 4-1.

Serializability requires the result of executions of concurrent transactions to be as if these transactions executed in some serial order. In the absence of data conflicts, serializability can be

[1] This overhead is essentially the overhead of lock acquisitions present in the base algorithm used in the system and is fundamental to a lock-based approach. Further, when locks are contended, their release also frequently results in a cache miss and excessive coherence traffic.

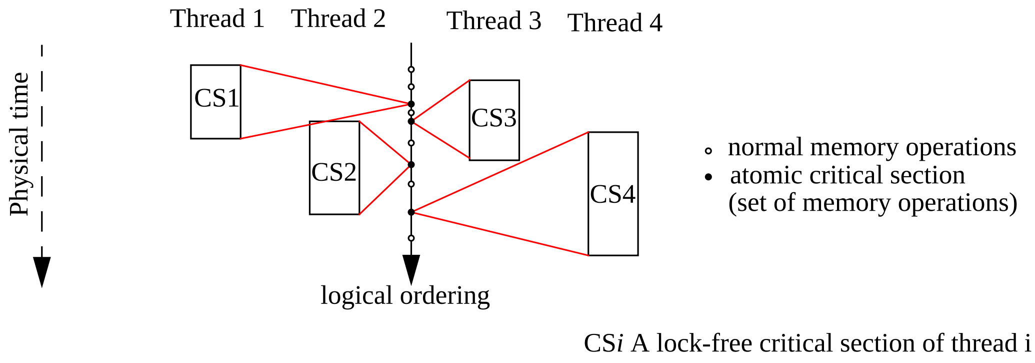

Thread 1
Thread 2
Thread 3
Thread 4
Physical time
CS1
CS2
CS3
CS4
logical ordering
○ normal memory operations
● atomic critical section
(set of memory operations)
CS*i* A lock-free critical section of thread i

***Figure 4-1: TLR and global memory ordering.** While critical section executions (without lock acquires) overlap in physical time (with or without data conflicts), each critical section logically appears to be inserted atomically and instantly in a logical ordering of memory operations with respect to other atomically inserted critical sections and individual memory operations.*

ensured using a technique such as SLE but the presence of data conflicts among concurrently executing threads requires additional mechanisms provided by TLR.

The basic idea behind TLR is as follows:

a) Treat locks as defining scope of a transaction

b) Speculatively execute the transaction without requesting or acquiring locks

c) Use a conflict resolution scheme to order conflicting transactions

d) Use a technique to give the appearance of an atomic commit of the transaction, such as is provided by SLE

TLR performs active concurrency control to ensure correct coordinated access to the data experiencing conflicting access by using the data itself rather than locks. Unlike TLR, SLE only identifies situations where lock-based concurrency control is not necessary—namely the absence of data conflicts among threads—and relies on the default lock-based concurrency control mechanisms if data conflicts occur.

Since TLR implements a concurrency control algorithm, it must provide the following two properties (Section 2.3):

1. *Safety.* The algorithm must guarantee “nothing bad ever happens” [96]. We show how to provide serializability of transactions, thus achieving the behavior of critical sections without lock acquisitions.

2. *Liveness*. The algorithm must guarantee “something good will eventually happen” [96]. We show how TLR is free from livelock and further, how TLR provides starvation freedom.

In the discussion in this section, we refer to a lock-free optimistic critical section as a transaction. In Section 4.3.1 we discuss achieving serializability of transactions in the presence of data conflicts. Section 4.3.2 presents the TLR algorithm and Section 4.3.3 illustrates the algorithm using an example.

4.3.1 Achieving serializability in the presence of conflicts

An execution of an optimistic lock-free transaction can be made serializable if the data speculatively modified by any transaction are not exposed until after the transaction commits and no other transaction writes to speculatively read data. A serializable execution can be achieved trivially by acquiring exclusive ownership of all required resources. If the thread executing the transaction does so for all required resources, the thread can operate upon the resources and then commit the updates atomically and instantly, thus achieving serializability.

In cache-coherent shared-memory multiprocessors, the above requires:

1. Acquiring all cache blocks that are accessed within the transaction in an appropriate ownership state

2. Retaining such ownership until the end of the transaction

3. Executing the sequence of instructions forming the transaction

4. Speculatively operating upon the cache blocks if necessary

5. Making all updates visible atomically to other threads at the end of the transaction

However, as we shall see next, the presence of conflicts may prevent resources from being retained thus preventing a successful execution of the lock-free transaction.

4.3.1.1 Necessity for conflict resolution

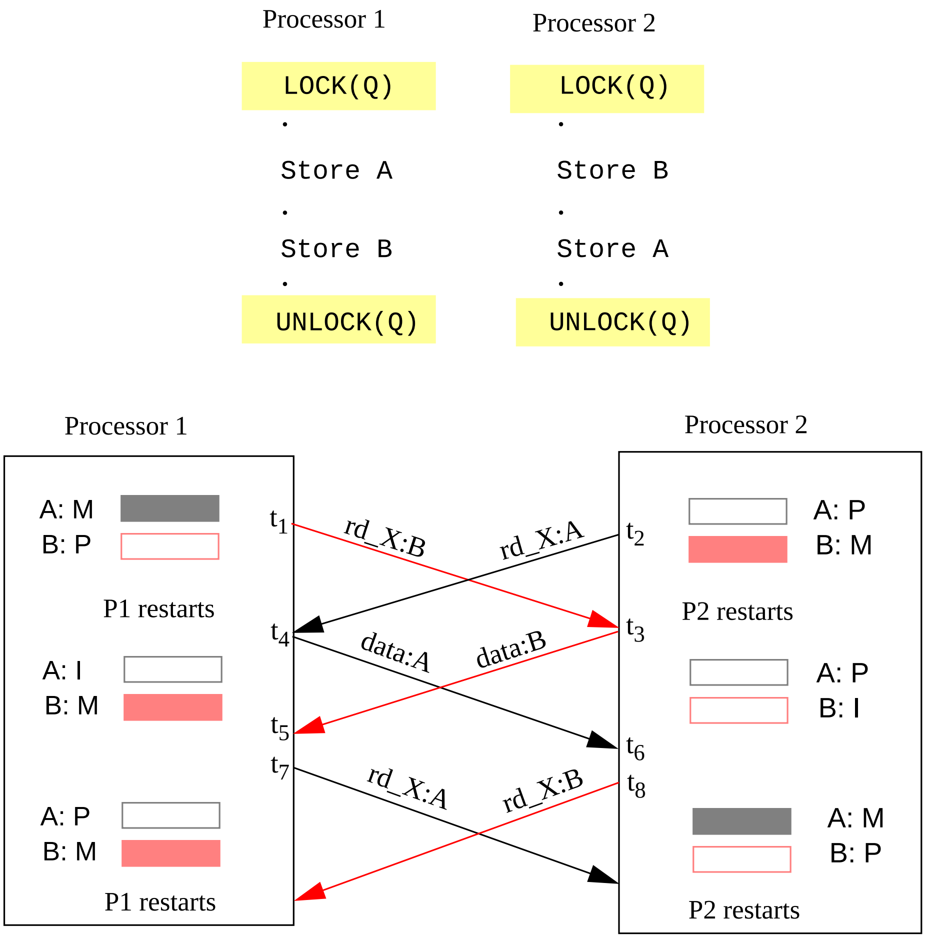
Livelock can occur if processors executing critical sections speculatively and in a lock-free manner repeatedly experience conflicts. As with SLE, the lock can always be acquired and forward progress is guaranteed but we require a solution that does not rely on lock acquisitions for forward progress.

Consider two processors, P1 and P2, each executing a lock-free critical section and both accessing (and writing) shared memory locations *A* and *B* in the critical sections. The two processors write the two locations in reverse order of each other—P1 writes *A* first and then *B* while P2 writes *B* first and then *A*. The code sequence within the critical section is shown in Figure 4-2. The messages and the state transitions for the corresponding blocks are also shown in Figure 4-2.

Time instances are labeled as t_i where *i* denotes progressing instances. Physical time progresses down and the changing cache block coherence state is shown over time. Assume both P1 and P2 have elided the lock by employing SLE and are in an optimistic lock-free execution mode. P1 has speculatively accessed block *A* and cached it in exclusive state (M). P2 has speculatively accessed block *B* and cached it in the M state.

At t_1, P1 issues a request for exclusive ownership (`rd_X`) for block *B* corresponding to the write operation to *B* within P1's critical section and at t_2, P2 issues an `rd_X` block *A* corresponding to the write operation to *A* within P2's critical section. The corresponding requests are accompanied by a transition of the respective cache blocks into a transient (pending P) state. At t_4, P1 receives P2's `rd_X` request for block *A*. P1 detects this as a data conflict (block *A* speculatively written to by P1 is accessed by another thread before P1 has completed its optimistic transaction). P1 triggers a misspeculation and restarts its optimistic lock-free execution. Similarly, P2 receives P1's `rd_X` for *B* at t_3 and P2 restarts execution. Both P1 and P2 respond with the valid non-speculative data. This sequence may occur indefinitely with no processor making forward progress because each processor repeatedly restarts the other processor.

Livelock occurs because neither processor obtains ownership of *both* cache blocks *simultaneously* in order to execute the transaction in a serializable manner and commit it atomically without locks. Cache coherence protocols can be used to allow processors to retain ownership of cache blocks. To ensure livelock freedom, among conflicting processors one processor must win the con-



```
Processor 1      Processor 2
LOCK(Q)          LOCK(Q)
.                .
Store A          Store B
.                .
Store B          Store A
.                .
UNLOCK(Q)        UNLOCK(Q)
```

Figure 4-2: Livelock in a lock-free optimistic transaction. *In this example, both processors repeatedly restart. A and B are memory locations. M corresponds to the modified state of the cache block and P corresponds to a pending (transient) state of the cache block. I is the invalid state. Time progresses downwards. The contents of the cache blocks are not shown.*

flict and retain ownership. To do so, TLR assigns priorities to the lock-free transactions and employs the following key idea:

> “Transactions with higher priority never wait for transactions with lower priority. In the event of a conflict, the lower priority transaction is restarted or forced to wait.”

Consider two transactions T_1 and T_2 executing speculatively. Suppose T_2 issues a request that causes a data conflict with a request previously made by T_1, and T_1 receives T_2’s conflicting request. The conflict is resolved as follows: if T_2's priority is lesser than T_1's priority, then T_2 waits for T_1 to complete (T_1 wins the conflict), else T_1 is restarted (T_2 wins the conflict). The "wait" mechanism may either involve an explicit negative acknowledgement or a delayed processing of the request.

The above uses concepts developed by Rosenkrantz et al. [144] (discussed earlier in Section 2.3.1.2) and specifically we adapt some of the key ideas in their *wound-wait* proposal for distributed concurrency control.

In the above approach, there cannot be any deadlock because for any finite set of transactions, the oldest cannot wait for any other transaction unless that transaction has first been wounded (i.e., restarted). The wounded transaction cannot be part of the deadlock because it is restarted and it relinquishes ownership of the block in question. In the wound-wait system, an older transaction never waits for a younger one except when the older transaction has wounded the younger transaction and is waiting for the wound to take effect. The oldest transaction therefore runs through the system wounding any younger transaction in its path. Thus the older transaction acquires all the resources it needs.

Rosenkrantz et al. [144] also proposed the *wait-die* approach. Suppose T_1 issues a request in conflict with T_2. Under wait-die, the conflict is resolved as follows. If T_1 has lower priority than T_2, then T_1 is permitted to wait; else it is aborted and forced to restart ("dies").

Restart behavior of wait-die and wound-wait. We contrast the wait-die scheme and the wound-wait scheme and discuss why we use the wound-wait scheme. See Section 2.3.1.2 and [144] for detailed difference between wound-wait and wait-die schemes. Suppose transactions T_1 and T_2 have a conflict and T_1 is restarted. The new sequence of requests issued by T_1 may be the same as the original one and the same sequence reach the site of the previous conflict. At this site, a new conflict will result if T_2 is still executing.

In the wait-die system, T_1 was the requestor that caused the original conflict (only the requestor can die in the wait-die system). In the new conflict T_1 is still the requestor and dies again. Thus there can be a long sequence of "dies" and while both T_1 and T_2 will eventually terminate, repeated attempts to run T_1 will consume system resources.

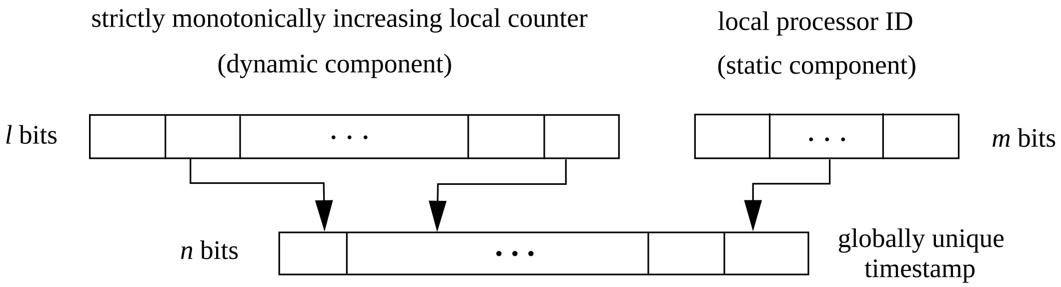


Figure 4-3: Constructing timestamps. *Bits from a strictly monotonically increasing counter are concatenated and combined with all bits from the local processor ID to construct a globally unique timestamp.*

By contrast, in the wound-wait system, T_1 was not the requestor of the original conflict and T_1 was younger than T_2. In the new conflict, T_1 is still younger than T_2 but this time T_1 is the requestor and hence waits. Transaction T_1 presumably consumes far less system resources if it is waiting than if it is continually being restarted.

For starvation freedom, the resolution mechanism must guarantee all contenders eventually succeed and become winners. We use timestamps for conflict resolution and we discuss them next.

4.3.1.2 Conflict resolution using timestamps

We use timestamps for resolving conflicts to decide a conflict winner—earlier timestamp implies higher priority. Thus, the contender with the earlier timestamp wins the conflict.

The timestamps we use have two components: a local logical clock and processor ID. The logical clock is a way of assigning a number to an event and the number is thought of as the time at which the event occurred. An event in our case is a successful execution of a TLR instance. The local logical clock value is increased by 1 or higher on a successful TLR execution and captures time in units of successful TLR executions on a given processor. Since these logical clocks are local, the logical clocks on different processors may have the same value. Such ties are broken by using the processor ID. Thus the timestamp comprising of the local logical clock and the processor ID are globally unique. Timestamp construction is shown in Figure 4-3. A strictly monotonically increasing sequence is defined as a sequence a_n if $a_{n+1} > a_n$ for all $n \in \mathbf{N}$. To construct a globally unique timestamp (*n* bits), bits from the local counter (*l* bits) and the local processor identifier (*m* bits) are concatenated together.

All requests generated from within a given transaction on a processor are assigned the same timestamp—namely the value of the timestamp at the start of the transaction. On a successful TLR execution, the processor increments its local logical clock to a value higher than the previous value (typically by 1) or to a value higher than the highest of all incoming conflicting requests received from other processors, whichever is larger. Doing so keeps the local logical clocks on the various processors loosely synchronized whenever a conflict is detected.

Our use of timestamps is similar to that proposed by Lamport [97]. Lamport used timestamps derived from logical clocks to implement distributed mutual exclusion with a starvation freedom guarantee. However, we only require timestamps for conflict resolution while Lamport used timestamps for *explicitly* ordering the execution of mutual exclusion regions among different processors. Thus with TLR, transactions that conflict in their data sets but do not actually observe any detected conflicts during their execution can execute in *any* order independent of the timestamps of the transactions. Since TLR does not require synchronized clocks, real-time system clocks can also be used.

The static component need not be explicitly exchanged and can be deduced simply from inspecting the sender identifier of the message. Thus the actual timestamp exchanged is *l* bits. We next discuss Lamport's logical clock construction and its application to TLR.

Lamport's logical clock construction. The simplest timestamp generation algorithm is due to Lamport [97] and is reproduced here. For now, assume unbounded timestamps.

Lamport defined $\rightarrow$ as a "happened before" relation. Informally, $a \rightarrow b$ means that is is possible for event *a* to causally affect event *b*. Two events are concurrent if neither can causally affect the other. Define a clock C_i for each process P_i to be a function which assigns a number $C_i\langle a \rangle$ to any event *a* in that process. The entire system of clocks is represented by the function C which assigns to any event *b* the number $C\langle b \rangle$, where $C\langle b \rangle = C_j\langle b \rangle$, if *b* is an event in processor P_j. Each C_i is implemented as counters with no actual timing mechanism.

A *clock condition* is defined as: for any events *a*, *b*: if $a \rightarrow b$ then $C\langle a \rangle < C\langle b \rangle$

To satisfy the clock condition, two conditions must hold:

C1. If *a* and *b* are events in process P_i, and *a* comes before *b*, then $C_i\langle a \rangle < C_i\langle b \rangle$.

C2. If a is the sending of a message by process P_i and b is the receipt of that message by process P_j, then $C_i\langle a\rangle < C_j\langle b\rangle$.

The process clock "ticks" through every number, with the ticks occurring between the process' events. Now assume the processes are algorithms, and the events represent certain actions during their execution. Process P_i's clock is represented by a register C_i so that $C_i\langle a\rangle$ is the value contained by C_i during the event a. The value of C_i will change between events, so changing C_i is not itself an event.

Lamport defined two implementation rules for ensuring clock conditions C1 and C2.

IR1. Each process P_i increments C_i between any two successive events.

IR2. (a) If event a is the sending of a message m by process P_i, then the message m contains a timestamp $T_m = C_i\langle a\rangle$. (b) Upon receiving a message m, process P_j sets C_j greater than or equal to its present value and greater than T_m.

Lamport further used these clocks to provide a distributed mutual exclusion algorithm guaranteeing starvation freedom because a process would eventually have the earliest timestamp in the system (all others, on a successful event execution, would increment their clocks as per the rule IR2 above).

Lamport's logical clocks applied to TLR. Lamport's logical clocks can be utilized for TLR. A successful lock-free execution of a critical section is considered to be an event. Thus, the timestamp update occurs at the completion of a successful lock elision. The new timestamp update occurs according to the rule IR2 described above—this requires the local node to keep track of the highest incoming timestamp from other processors. In TLR, all nodes are not guaranteed to see all timestamps at any time, only the ones that conflict. Nevertheless, only one timestamp—the highest—need be tracked.

By using Lamport's logical clocks, at any time, two clocks will not drift[2] arbitrarily far because they synchronize on every conflict in which they both participate. For example, consider processor P0 executes 1000 successful lock-free executions and has now set its local counter to 1000. Meanwhile P1 executes no critical sections. Subsequently, P1 executes a critical section,

[2] When two clocks tick at different rates, it creates an ever-widening gap in perceived time. This is called clock drift. Clock skew is the difference between two clocks at one point in time.

conflicts with P0, and wins the conflict because P1's timestamp is 0 and is earlier than P1's timestamp. P0 restarts and issues its request again and the request for the conflicting data is forwarded to P1. Thus, while P1's current clock value is 0, on a successful execution, its new clock value will be 1001 (> 1000). The drift between the two clocks is contained and they re-synchronize.

Achieving starvation freedom. Starvation freedom is achieved by retaining and reusing timestamps in the event of a misspeculation and restart. By reusing timestamps, processors retain their position. By updating timestamps as above, a processor will eventually have the earliest timestamp in the system and thus will eventually win all conflicts. TLR uses timestamps solely for the purpose of comparing priorities of two conflicting threads to determine which has a higher priority. The starvation freedom property follows from the use of Lamport's logical clocks.

4.3.2 TLR algorithm

We assume a processor with support for SLE. A processor executing the TLR algorithm is considered to be in TLR mode. All operations executed by a processor in TLR mode are part of the optimistic lock-free transaction and are speculative. For brevity, we will refer to an optimistic lock-free transaction as simply a transaction. Conventional cache coherence protocols are used to allow processors to retain ownership of cache blocks. In an invalidation-based cache coherence protocol, a processor with an exclusively-owned cache block receives and must respond to subsequent requests for the block. The processor controls the block and can appropriately respond. Figure 4-4 shows the TLR algorithm. In the discussion below, we use the term deferred to imply the processor retains ownership.

The first step is calculating the globally unique local timestamp.

The second step is identifying start of a transaction. We use SLE to identify the start and end of transactions. SLE does so by exploiting *silent store-pairs*: a pair of store operations where the second store undoes the effects of the first store and the intervening operations appear to execute atomically. The first store of the pair corresponds to the start of the transaction and the second store of the pair corresponds to the transaction end. Once the start is identified, the lock is elided thus leaving the lock *free*. The processor register state is saved for recovery in the event of a misspeculation.

The third step comprises actions that may occur concurrently and are listed below.

```
1. Calculate local timestamp

2. Identify transaction start
   a) Initiate TLR mode (use SLE to elide locks).
   b) Execute transaction speculatively.

3. During transactional speculative execution
   • Locally buffer speculative updates.
   • Append timestamp to all outgoing requests.
   • If incoming request conflicts with retainable block and has later timestamp, retain
     ownership and force requestor to wait.
   • If incoming request conflicts with retainable block and has earlier timestamp, service
     request and restart from step 2b if necessary. Give up any retained ownerships.
   • If insufficient resources, acquire lock.
     •  No buffer space
     •  Operation cannot be undone (e.g., I/O)

4. Identify transaction end
   a) If all blocks available in local cache in appropriate coherence state, atomically commit
      memory updates from local buffer into cache (write to cache using SLE).
   b) Commit transaction register (processor) state.
   c) Service waiters if any.
   d) Update local timestamp.
```

Figure 4-4: TLR algorithm. *A mechanism for retaining ownership of cache blocks is assumed to be present. A retainable cache block is defined as a block in an exclusively owned coherence state. Requests are forwarded to the cache with the writable copy of the block.*

- A cache miss generated for data within the speculative execution carries with it the processors timestamp.

- Requests from other processors that result in a data conflict for data accessed within the transaction are checked for priority. If the incoming request has a later timestamp than the local processor, the incoming request's response is deferred. If the incoming request has an earlier timestamp, the local processor loses the conflict. It must service earlier deferred requests in the order they were received, thus maintaining the coherence protocol ordering, and then service the conflicting incoming request. By ensuring we always maintain coherence protocol ordering, we do not change the coherence protocol correctness conditions discussed earlier in Section 2.1.2.3. The execution may restart but the local clock is not updated.

- If any resource constraints, or operations that cannot be undone, are encountered, TLR cannot be applied. The processor requests the lock by exposing the elided writes and exits TLR mode. Since the lock is kept in shared state under TLR, any write to the lock triggers invalidations thus automatically informing other participating processors of the violation of the silent store-pair elision under TLR. During speculative execution, data modified is buffered in the write buffer and exclusive requests for the cache block are issued to the memory system.

Finally, when a transaction end is identified, the transaction is committed. If all appropriate blocks have been brought into the cache in appropriate state (exclusive or shared), then the buffered data in the write buffer is *atomically* committed into the cache—all required blocks are already in writable state in the cache. If not, then speculative execution can proceed until the blocks corresponding to the write buffer are available in appropriate state. After the speculative data has been committed into the cache, deferred requests from before are then serviced in order. The local logical clock update is performed as discussed in Section 4.3.1.2.

Up to now, we have focused on interaction among timestamped requests—requests that are part of critical sections. However, in some programs, the data protected by locks may be accessed from outside a critical section and hence without locks, and may conflict with timestamped requests. While this is a data race, it may be acceptable for the program. Such situations may be correctly handled in various ways. One approach is to trigger a misspeculation when an un-timestamped request is received. Thus, if any thread performs a conflicting access from outside a critical section, then TLR cannot be applied because a data race exists. Another approach is to treat un-timestamped requests as deferrable and thus achieve successful lock-free execution even in the presence of data races. Such a request is assumed to have the latest timestamp in the system (and thus the lowest priority) and the un-timestamped request is atomically ordered after the current critical section. Since a data response is not sent until after the critical section, the requestor cannot consume the data and hence is ordered with the correct value. TLR, in effect, is masking the data race and forces the data race to be ordered after the critical section completes. We discuss the stability implications of this in Section 4.6 where subtle unwanted data races may be prevented from being exposed in the system for a given execution thus resulting in stable software.

4.3.3 TLR algorithm example

We revisit the example of Figure 4-2 and apply the algorithm outlined in Figure 4-4 to it. Consider Figure 4-5. Two processors, P1 and P2, execute a lock-free critical section and both write shared memory locations *A* and *B* in the critical section. Both the processors have a unique timestamp—TS1 for P1 and TS2 for P2 where TS1 < TS2 (processor P1 has higher priority than processor P2 and wins all conflicts). Assume both processors have the additional ability to buffer and delay responding to incoming requests. As in the earlier example, the two processors write the two locations, *A* and *B*, in reverse order of each other. Assume both P1 and P2 have elided the lock by employing SLE and are in the optimistic lock-free execution mode. P1 has speculatively accessed block *A* and cached it in exclusive state (M). P2 has speculatively accessed block *B* and cached it in the M state.

At t_1, P1 issues a `rd_X` for block *B* corresponding to the write operation to *B* within P1's critical section and at t_2, P2 issues a `rd_X` for block *A* corresponding to the write operation to *A* within P2's critical section. The respective cache blocks transition into a transient (pending P) state. All memory operations within the transaction are assigned the same timestamp. Therefore P1's `rd_x` for *B* has TS1 appended and P2's `rd_X` for *A* has TS2 appended. At t_3, P2 receives P1's request and compares the incoming request's timestamp TS1 with its local timestamp TS2. Since the incoming request has an earlier timestamp than P2, P2 services the request and responds with the data for block *B* (non-speculative value). On applying the incoming request, a data conflict is triggered at P2 and P2 restarts execution of its transaction. At t_4, P1 receives P2's `rd_X` request for block *A*. Since TS1 < TS2, P1 wins the conflict and defers the request by buffering it. The cache block for *A* stays in state M. At t_6 P1 receives data for block *B* from P2. P1 has acquired and retained permissions on *both* cache blocks *A* and *B* and can successfully execute and atomically commit the transaction. At t_8, P1 completes its transaction, architecturally commits its speculative state and services P2's deferred request. P1 responds with the latest architecturally correct data. Meanwhile, P2 has restarted and is re-executing its transaction. The key difference between Figure 4-2 and Figure 4-5 is P1's ability to retain exclusive permissions in the latter example.

```
Processor 1          Processor 2

 LOCK(Q)              LOCK(Q)
.                    .
Store A              Store B
.                    .
Store B              Store A
.                    .
UNLOCK(Q)            UNLOCK(Q)
```

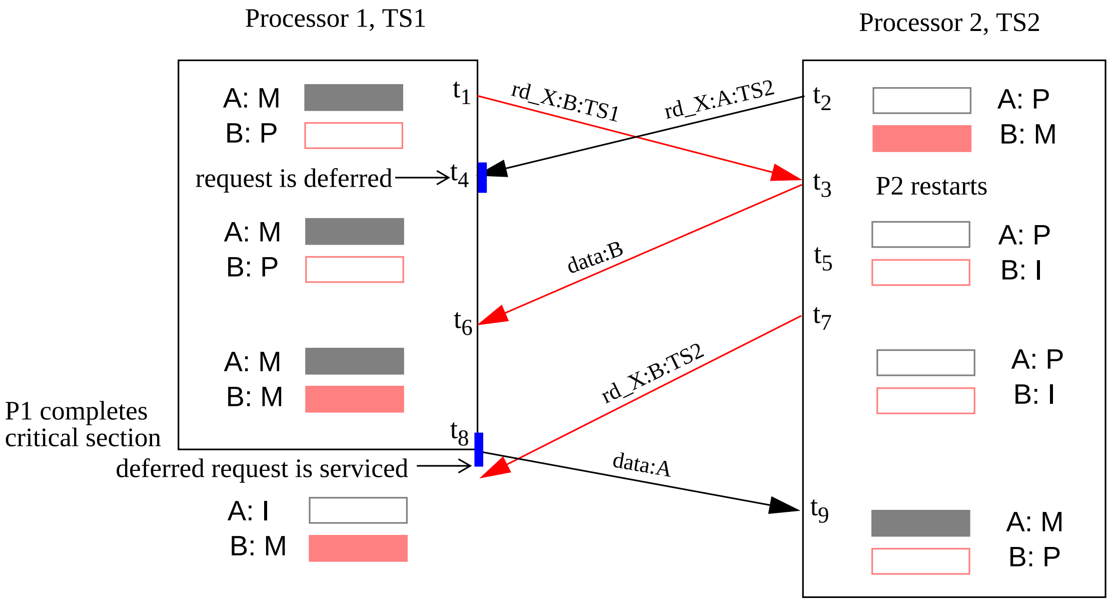


Figure 4-5: Serializable execution in the presence of conflicts. *A conflict resolution scheme is employed allowing processor 1 to retain exclusive ownership of both cache blocks A and B. By deferring a response, conflicts are masked and a successful atomic execution is achieved. A and B are memory locations. M corresponds to the modified state of the cache block and P corresponds to a pending (transient) state of the cache block. I is the invalid state. Time progresses downwards. The contents of the cache blocks are not shown.*

4.4 A TLR implementation

In this section, we discuss how TLR can be implemented. The algorithm outlined earlier in Figure 4-4 relies on the ability of a processor to retain ownership of a cache block. In Section 4.4.1 we discuss various mechanisms for retaining ownerships of cache blocks and in Section 4.4.2 we discuss one such mechanism in detail. Until now we have not discussed the interactions of TLR with the shared coherence state and in Section 4.4.3 we discuss handling the shared coherence state under TLR. We qualitatively discuss the performance implications of the interactions between timestamp-enforced order and the coherence-protocol-enforced order in Section 4.4.4. TLR enforces fair and deadlock-free concurrency control using timestamps. However, if mechanisms exist otherwise for achieving such concurrency control, the use of timestamps may be relaxed and the coherence protocol order may itself be used. In Section 4.4.5 we discuss this in detail and show when timestamp order may be relaxed for better performance. In Section 4.4.6 we discuss an optimization for controlling misses. Finally in Section 4.4.7 we discuss implementation-specific constraints for TLR.

4.4.1 Mechanisms for retaining ownerships

Two policies to retain exclusive ownership of cache blocks are NACK-based and deferral-based. With NACK-based techniques, a processor refuses to process an incoming conflicting request (and thus retains ownership) by sending a *negative acknowledgement* (NACK) to the requestor. Doing so forces the requestor to retry at a future time. With deferral-based techniques, a processor defers processing an incoming request by buffering the request and masking any conflict. The requestor assumes the request is being processed and has been ordered by the coherence protocol but the requestor does not get a response right away. We discuss retaining ownership using NACKs in Section 4.4.1.1 and retaining ownership using request deferral in Section 4.4.1.2.

4.4.1.1 Retaining ownership via negative acknowledgements

In the scheme for retaining ownership using NACKs, the conflict-winning processor with an exclusively owned cache block responds to the incoming conflicting request to the cache block by sending a NACK message to the requestor. This message informs the requestor to retry the request again after a bounded time.

The advantages of a NACK-based approach are:

+ The approach is conceptually simple and all interactions among various processors are explicitly handled. No deadlock dangers due to ownership retention exist (as discussed in Section 2.3.1) because of explicit handshaking among conflicting processors.

+ Many protocols already support NACKs. Thus, adapting the TLR to such protocols would be straightforward.

The disadvantages of a NACK-based approach are:

- A NACK-based approach requires the ability of the processor to prevent a cache block coherence state transition, that results in ownership loss, from occurring. Doing so allows a processor to retain exclusive ownership even if another processor requests the block. Preventing state transitions from occurring may be difficult to achieve in some systems such as modern broadcast-snooping systems built using high-performance indirect networks. Such systems do not often have the ability to NACK requests and state transitions are considered to have implicitly occurred depending upon the serialization point (often a logical bus) of the coherence protocol. Since the transitions occur implicitly, preventing such transitions from occurring may be difficult. In the absence of explicit messages (as in the case of directories), implementing NACKs may be a non-trivial task in such systems.[3]

- Apart from the implementation difficulty on some systems, the protocol may need to be changed in systems that do not support NACKs for other reasons. Such a change may not be desirable since the NACKs may have been avoided for a specific reason.

- Since NACK-based approaches rely on retrying the request, the timing of the retry is a critical factor for performance. A retry that is too early will result in unnecessary network traffic and coherence protocol interference resulting in additional latency, while retrying too late will result in unnecessary delay. Often, this is similar to the exponential backoff problem—the time interval between two attempts may be sensitive to the workload itself and may be difficult to tune.

[3] Logical bus designs do exist that allow a request to be NACKed even in high performance broadcast systems. The Gigaplane has support for the ignore signal that prevents coherence input queues in the system to avoid observing a request on the bus for optimal global request ordering [152]. Such support can be used for implementing NACK-based ownership retention schemes on modern broadcast systems.

4.4.1.2 Retaining ownership via request deferrals

With deferrals, the conflict-winning processor with an exclusively owned cache block delays processing the incoming request for a bounded time (preferably until the processor has completed its transaction) and thus defers the request. The coherence transitions (and state transitions as seen by the "outside world") are assumed to have occurred but the processor does not locally apply the incoming request. Request deferral and delayed responses works in split-coherence-transaction[4] systems where the address request processing is split into two sub-coherence-transactions—request and response. The response (often the data) may appear an arbitrary time later and any number of other requests and responses may occur between the two sub-coherence-transactions.

The advantages of a deferral-based approach are:

+ Since coherence protocol processing is delayed, the coherence protocol itself is essentially unchanged. The existing state transition tables do not require changes nor do any philosophical design decisions of the coherence protocol. Doing so allows the policy to be implemented in just about any protocol without changing the protocol itself.

+ By deferring requests until the end of the transaction, traffic is reduced to the minimum because the external request is serviced at the *right* time. Retries are not necessary for that request and therefore the requestor does not have to worry about the accurateness of the timing of the retry. This provides performance benefits and we discuss them later.

+ Ordering can be easily maintained, providing benefits such as fairness, starvation freedom, etc.

The main disadvantages of a deferral-based approach are:

- Additional hardware is required to buffer the incoming request.

- Request deferral introduces deadlock possibilities in the protocol because we now have waiting processors and thus a danger of a cyclic waits-for graph exists (see Section 2.3.1.2). Special mechanisms are required to handle such situations.

[4] We use the term coherence-transactions and sub-coherence-transactions to differentiate them from our use of the term transactions in this dissertation. While coherence-transactions are also commonly referred to as transactions, they are a low level representation consisting of individual requests and responses in the coherence protocol. Our use of the term transactions refers to a high level concept.

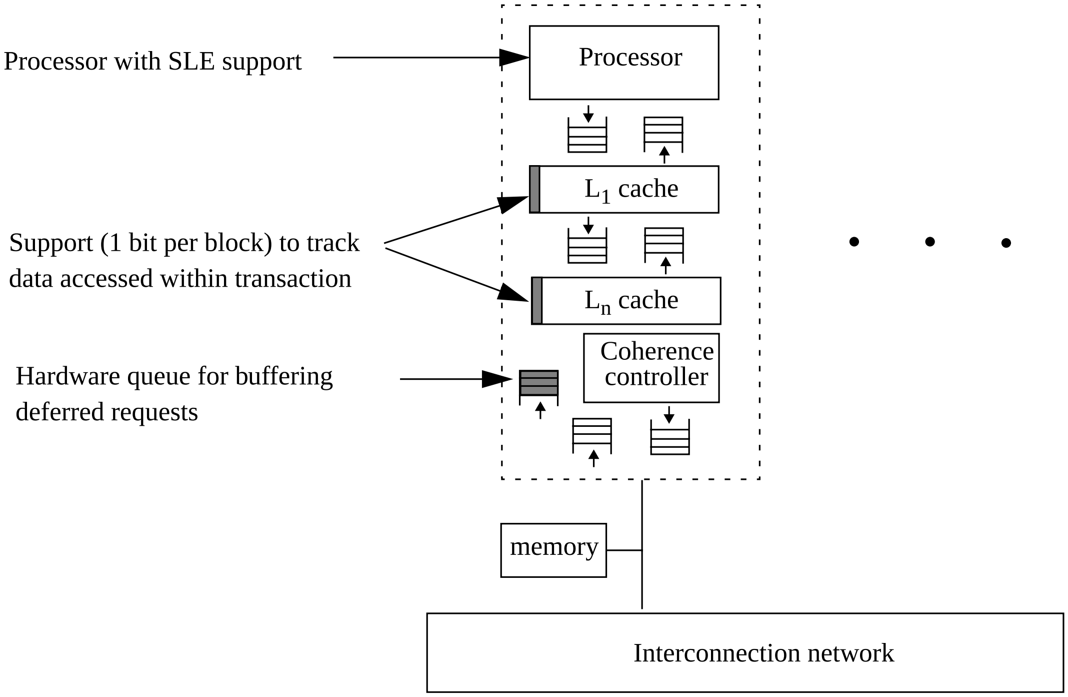

Processor with SLE support
Processor
L_1 cache
Support (1 bit per block) to track data accessed within transaction
L_n cache
Coherence controller
Hardware queue for buffering deferred requests
memory
Interconnection network

Figure 4-6: TLR implementation details. *The additional hardware structures are shown shaded.*

4.4.2 A deferral-based implementation

In this thesis, we use a deferral-based scheme because it does not require coherence protocol support (such as NACKs). We now discuss a deferral-based implementation of the algorithm. Figure 4-6 shows a shared-memory multiprocessor where every processor has a local cache hierarchy and they are connected together via an interconnection network. We make no assumptions regarding the memory consistency model, coherence protocol, or interconnection network. The protocol may be snoop-based or directory-based and the interconnect may be ordered or unordered. The processor is assumed to have SLE capability: support for predicting regions as transaction, support for buffering local speculative updates, mechanism to track data accessed within transactions (an access bit per cache block tracks data accessed during the transaction), and ability to detect data conflicts.

TLR support is required at the coherence controller where decisions for deferrals are made. We do not require changes to the coherence protocol state transitions. The TLR concurrency control algorithm runs in parallel and along with the coherence protocol and only performs deadlock-free concurrency control.

Misses generated within a transaction carry a timestamp. An additional deferred coherence input queue is present to buffer incoming requests that have been deferred by the local processor. Two messages sent only within the local cache hierarchy (*start_defer* and *end_defer*) from the processor to the cache controller are needed. The *start_defer* is sent when the processor transitions into speculative lock-free transaction mode and *end_defer* is sent on exiting such a mode. The *end_defer* message may clear the access bits in the local cache hierarchy if necessary. These messages are ordered with respect to each other and multiple pairs of messages may be present in the local hierarchy.

In section Section 4.4.2.1 we discuss implementation-specific coherence protocol interactions with TLR. We base our discussion around a modern broadcast snooping protocol, the Sun Gigaplane [151]. This choice does not take away from the generality of our discussion. Interactions because of the presence of transient states in coherence protocols may result in priority information not being completely propagated. In Section 4.4.2.2 we discuss a way to address this issue. In Section 4.4.2.3 we give an example to better understand how TLR works with coherence-protocol-specific aspects.

4.4.2.1 Deadlock danger

In this section we discuss how deadlock[5] may occur because of the interaction of TLR with transient states in cache coherence protocols. Transient states in cache coherence protocols do not have valid data available yet and may not have a readable/writable copy of the cache block even though the cache block state is valid and the request that initiated this transition state has been ordered by the coherence protocol. On a cache miss, the cache block performs a transition from invalid to a pending state and it stays in a pending state between the request initiation and completion. At some time between the two phases, the request gets ordered by the coherence protocol and the cache may become the owner of the cache block according to the coherence protocol, even

[5] This deadlock is not related to software deadlocks that may arise due to incorrect locking methodologies.

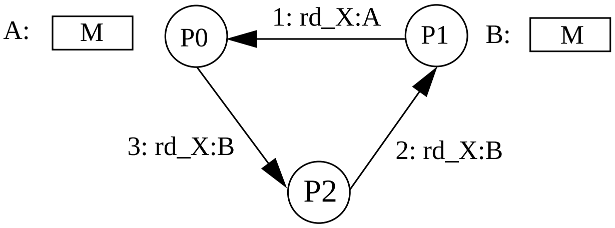


Figure 4-7: Deadlock with three processors. *Unlike the earlier example with 2 processors, the presence of an additional processor complicates issues because now all requests are distributed in the system and all processors are not guaranteed to observe all other requests.*

though data is unavailable. This request-response decoupling introduces a complication because even though a processor may lose a conflict under TLR, it does not have data to provide to the conflict-winning requestor.

In mechanisms where processors delay servicing incoming requests, if only two processors are involved (as shown in Figure 4-5) deadlock is not a problem. This is because both processors are aware of each others requests and can make a determination based completely on the incoming identifier and the local identifier. The situation is however complicated by the addition of another processor.

Consider Figure 4-7 where three processors P0, P1, and P2 are shown executing transactions. The arcs correspond to requests generated within the transaction. The arc labelling "1:rd_X:A" means a read for exclusive ownership (`rd_X`) request for block *A* was issued at time t_1. Assume the priority ordering among the processors is as follows: P0 > P1 > P2 where P0 has the highest priority. P0 has cache block *A* in exclusive owned (M) state and P1 has cache block *B* in M state.

At time t_1, P1 issues a `rd_X` request for cache block *A*. As per the cache coherence protocol, P0 owns the cache block and thus P1's request is forwarded to P0. P0 compares its local identifier with P1's incoming message and wins the conflict. P0 buffers P1's `rd_X` request for *A* and delays responding to the request. According to the cache coherence protocol P1 exclusively owns the cache block *A* but the data (and hence the actual write permissions to the block) are still with P0. *P1 is waiting for P0 for cache block A.*

At time t_2, P2 issues a `rd_X` request for *B*. According to the cache coherence protocol, P1 owns the cache block and thus P2's request is forwarded to P1. P1 compares its local identifier with P2's incoming message and P1 wins the conflict. P1 buffers P2's `rd_X` for *B* request and delays a response. Now, according to the cache coherence protocol, P2 exclusively owns the cache block *B* but the write permissions to the block are still with P1. *P2 is waiting for P1 for cache block B.*

At time t_3, P0 issues a `rd_X` request for *B*. According to the cache coherence protocol, P2 owns the cache block (even though the data is still with P1) and thus P0's request is forwarded to P2. P2 compares its local identifier with P0's incoming message and loses the conflict. P2 must service P0's request by responding with data. However, P2 cannot do so because P2 is waiting for P1 to release cache block *B*. P1 will not release the cache block because P1 won the conflict (for cache block *B*) but P1 is itself waiting for P0 for cache block *A*.

P2 is waiting for P1 (for cache block *A*) which is waiting for P0 (for cache block *B*) which is waiting for P2 (for cache block B). If this wait is uncontrolled, deadlock is present. The waiting processors are unaware of other waiting processors and inadvertently form a *cyclic waits-for graph* [71].

Deadlock danger exists only if more than two processors are involved and only if more than one cache block is involved. If only a single cache block is under conflict, then a cyclic waits-for graph cannot exist because a processor cannot have more than one request outstanding for a given block at any time.[6] One processor will have the cache block in exclusive owned state (and thus will not "wait for" any other processor) and will complete its optimistic lock-free transaction. On completing the transaction, any deferred requests will be serviced.

The deadlock problem discussed above has strong parallels to the database concurrency control problem. Each cache block in our example can be conceptually treated as a lock in a database. The process of retaining ownership of a cache block by deferring incoming requests is analogous to that of acquiring a lock in a database. Hence, multiple cache blocks conceptually correspond to multiple locks. Deadlock occurs because the concurrency control mechanism in our example above did not coordinate the acquisition of exclusive ownership of the caches blocks in a manner guaranteeing forward progress (the same way a database system deadlocks if the locks are not managed and acquired properly).

[6] In processors with blocking caches, a second request cannot be issued until the first outstanding request is serviced. In processors with non-blocking caches subsequent requests (secondary misses) to a block that already has a request outstanding (the primary miss) for it, are merged with the primary miss [88].

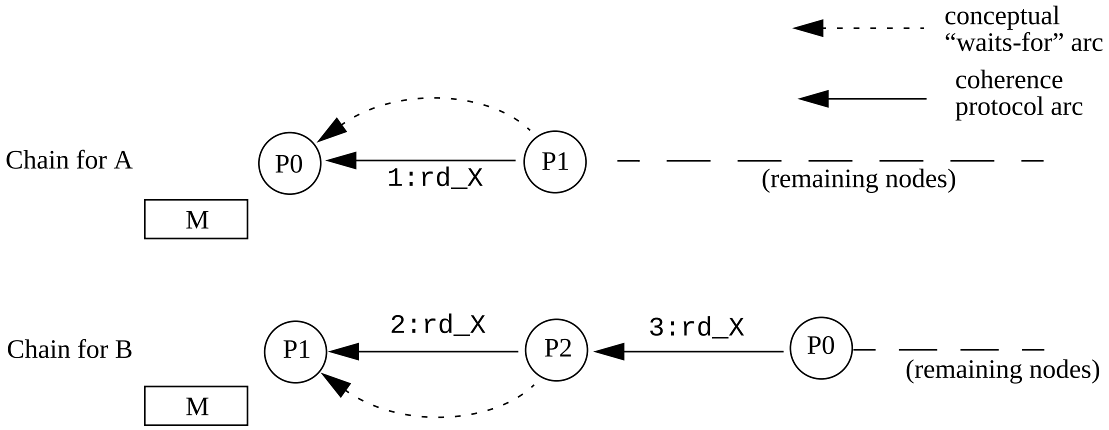


Figure 4-8: Understanding deadlock with request deferrals. *This is similar to the earlier figure except here the chains are shown separately. To prevent a deadlock, P1 must be aware of P0's request for block B. In the example shown, P2 receives P0's request and thus prevents P1 from observing the conflicting request.*

We first reproduce the example shown above again in Figure 4-8. The coherence protocol chains for two cache blocks, *A* and *B*, are shown. The protocol chain for any coherence block is always rooted at a stable block; in the figure the stable state is the modified (M) state of the MOESI classification (Section 2.1.2). Further assume the conflict resolution priority for the processors is as follows P0 > P1 > P2. Therefore P0 can defer P1's requests and P1 can defer P2's requests. We do not show data responses since we assume a split-transaction system and the coherence protocol transitions (or the appearance of such transitions) occur at the time the coherence request is ordered at the point of serialization either at the directory or the broadcast network. No assumptions are made about the implementation of the coherence protocol, namely whether it is snoop-based or directory-based.

At t_1, processors P0 and P1 have blocks *A* and *B* respectively in the M state. P1 subsequently requests ownership of the block *A* as shown by the solid arc between P1 and P0. Since P0 defers P1, the waits-for arc, shown dotted, goes from P1 to P0. P2 requests ownership of the block B and similarly the arcs are constructed. At t_3, P0 now requests block *B* and the request is forwarded to P2. P2 cannot respond to P0 because P2 does not have data. P2 cannot defer P0 because P2 has lower priority than P0. No waits-for arc exists between P0 and P2. An important point to note is that P1 does not know that P0 is waiting for block *B* because the request from P0 was forwarded to P2.

4.4.2.2 Propagating priority information

The key idea for implementing a deferral-based concurrency control mechanism is to propagate information about processor priorities along the coherence protocol chains to prevent cyclic waits. On a miss, a processor allocates a pending buffer, a miss status handling register (MSHR), and tracks the request. If the processor receives a request (an intervention) from another processor for the outstanding block, an intervention buffer or the MSHR tracks the incoming request. When the processor receives data for the block, the processor operates upon the data and sends it to the requestor based on the information stored in the local MSHR. In Figure 4-8, for the chain for block *A*, P0 is aware of P1 but P1 is not aware of P0. Similarly, for block *B*, P1 is aware of P2 but not vice versa and P2 is aware of P0 but not vice versa. P0 can send information to P1 (regarding deadlock-free concurrency control) but P1 cannot send information to P0 because P1 is unaware of P0.

P0 must inform P1 that P0 has higher priority and must not be forced to wait for block *B*. The presence of P2 in the chain prevents P1 from observing P0's request. Mechanisms can be added to propagate such information along the chain. The conflicting requests must propagate along the coherence chain towards the root (i.e., the stable block) to "restart" lower priority requests. We use special messages, we call *marker messages*, for doing so.

Marker messages are directed messages sent in response to a request for a block under conflict for which data is not provided immediately. The delay may be because either the processor is forcing the requestor to wait or the processor does not have the data for the block in question but is considered to be the owner of the block. The idea behind marker messages is to make processors aware of their immediate neighbors in a chain. These messages have no coherence interactions. The marker messages are *only* required when the processor is doing TLR and receives a conflicting request for an exclusively-owned block. If a marker message is sent, the subsequent data response (which is sent at some unspecified but finite time later) must carry the information that a marker message was sent. This is used to match up a marker message with its data response in the memory system.

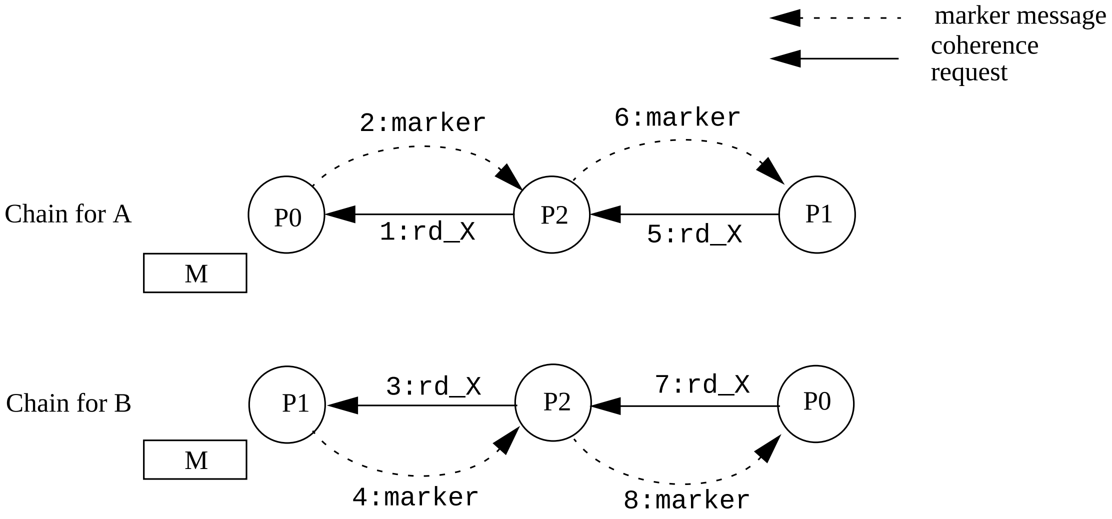


Figure 4-9: Role of marker messages. *Marker messages are primarily used to construct backward pointers: the requestor is informed about the node that is participating in the waits-for graph.*

Consider Figure 4-9. P0 sends a marker message to P2 informing P2 of the waits-for graph. Similarly, P2 sends a marker message to P1 informing P1 of the waits-for graph. Consider the chain for block *A*. P0 sends a marker message to P2 because P0 is deferring P2. P2 sends a marker message to P1 because P2 cannot provide data (P2 is waiting for P0). Now consider the chain for block *B*. P1 sends a marker message to P2 (because P1 is deferring P2) and P2 sends a marker message to P0 because P2 cannot provide data yet.

We have a mechanism to propagate timestamps requests upstream (*probes*) to the cache that has the block with valid data. Probes are only used to propagate a conflict request upstream in a cache coherence protocol chain. Thus, when P2 receives P0's request for *B*, P2 forwards the probe (with P0's timestamp) to P1 since P2 received a marker message from P1. P1 receives P0's forwarded probe (via P2) and loses the conflict because P0 has higher priority than P1. P1 releases ownership of block *B* and the cyclic wait is broken.

4.4.2.3 An example

We step through an example to show how TLR works with a coherence protocol with transient states. The algorithm is based on the algorithm discussed in Section 4.3.2 and uses the terminology of the wound-wait algorithm discussed in that section. The example is split over two figures: Figure 4-10 and Figure 4-11. The system state is shown at various times. Four processors P0, P1, P2 and P3 are part of the system. In priority ordering, P0 > P1 > P2 > P3; P0 has the highest priority and P3 has the lowest. The solid lines are the coherence messages and the dotted lines are the new TLR messages. The system state is numbered from (i) through (vii) over the two figures.

Initially, in (i), the system state consists of two processors P0 and P1 both executing a lock-free optimistic transaction. Processor P0 has accessed block *A* and has it locally cached in the exclusive modified state (M). Similarly P1 has accessed block *B* and has it locally cached in the M state. In the discussion below, a processor that has been restarted because of an incoming higher priority conflicting request is said to be wounded.

Consider (ii). Two additional processors, P2 and P3, issue requests. At t_1, P3 issues a `rd_X` (read-for-exclusive-ownership request) for block *A* and this request is forwarded to P0. P0 has higher priority and can defer incoming requests to a block exclusively owned by it and accessed within the transaction. Thus, P0 buffers P3's request and at t_2 sends a marker message to P3. The purpose of this message is to inform P3 that P0 will respond to P3's request but after a delay. This message succeeds in creating a backward arc (P0 knows P3 requested the block and now P3 knows P0 will respond). Similarly, at t_3, P2 sends an `rd_X` for block *B* to P1. The sequence is similar as earlier. P1 buffers the request and responds with a marker message to P2. Both, P0 and P1, have retained ownership of blocks *A* and *B* respectively. P3 and P2 now *wait* for P0 and P1.

Now consider (iii). At t_5, P1 issues a `rd_X` for block *A*. Since P3's `rd_X` request for block *A* was the last request for the block ordered by the coherence protocol, P3 owns the block and will respond to P1. P1's request is therefore forwarded to P3. P3 compares priorities, and notes it has a lower priority than P1. P1 thus *wounds* P3. Since P3 does not yet have valid data for the block (the writable copy of the block is still with another processor upstream), P3 forwards the incoming message as a *probe* upstream to P0. P3 is aware of P0 because of the earlier marker message.[7] P3

[7] P3 need not forward the probe to P0 if P3 is aware of P0's priority and P0's priority is higher than P1's. In that case, P3 only sends the marker back because P1 cannot wound P0 and must wait for P0.

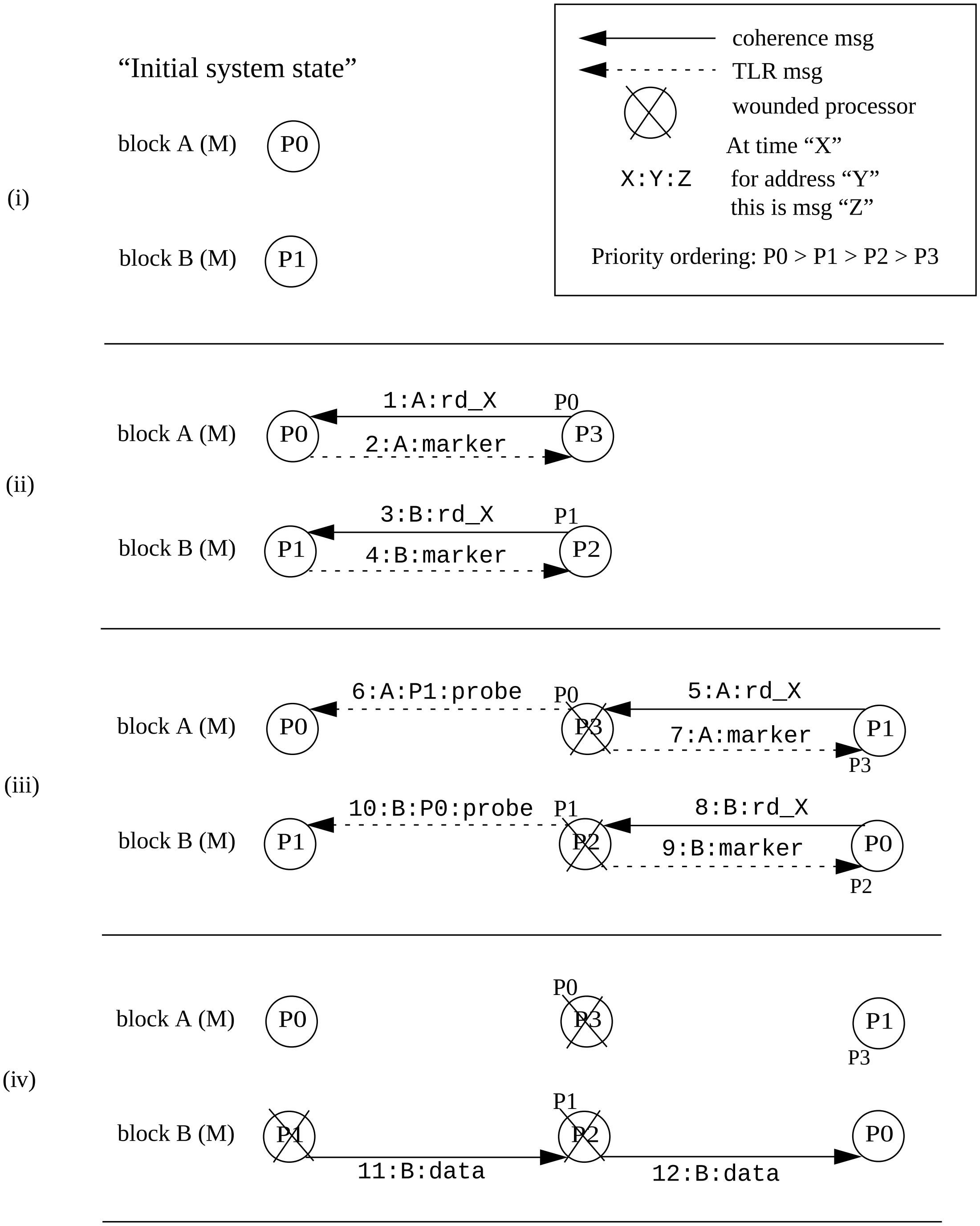


Figure 4-10: Example of a TLR implementation. *The figure shows an application of the TLR algorithm to a cache coherence protocol. Various stages of the algorithm are shown and within each stage a sequence of events is shown. The sequence is continued in the next figure.*

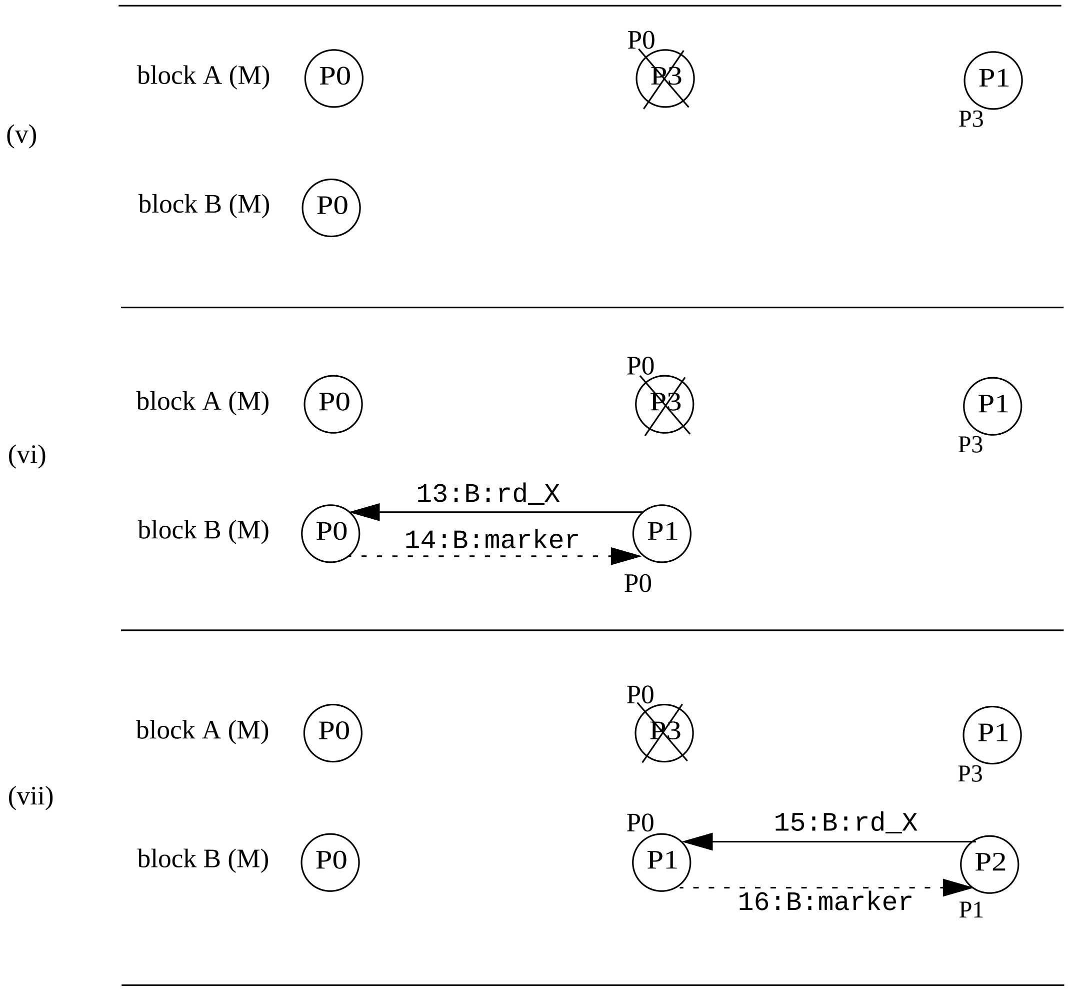


Figure 4-11: Example of a TLR implementation continued. *This figure is a continuation from the earlier figure. As can be seen, the TLR algorithm implementation results in the chain for block B being reordered. Thus, eventually, P0, the processor with the highest priority, gets ownership of the block after "wounding" processors P1 and P2. The stage (vii) shows P1 and P2 re-issuing requests and rejoining the coherence protocol chain. However, in this specific case where P2 re-issued the request after P1, P2 is chained behind P1 and P1 is not wounded. Thus, when P0 completes its transaction, the blocks A and B will be forwarded on to the next requestors as per the coherence protocol chain.*

is shown with a cross mark because P3 has now been wounded.[8] If the probe is forwarded to P0, P0 ignores the probe because P0 has higher priority than the incoming forwarded probe from P1 and P1 is forced to wait for P0. P1 is also waiting for P3 *but* P3 has been wounded and the wounding takes time to be effective—namely when P0 responds with the data to P3, P3 forwards the data along to P1 without using the data.[9] This wait of P1 for P3 is acceptable according to the wound-wait definition because P3 has been wounded. Similarly, at t_8, P0 issues an `rd_X` request for block *B*. P2 receives P0's request. P2, being a lower priority than P0, is wounded. However, P2 forwards P0's request as a probe upstream to P1. Unlike P3's probe message, P0's request must be forwarded because P0's priority is higher than P1's. P2 also responds with a marker message back to P0.

In (iv), P1 has received P0's forwarded probe. P1 has lower priority than P0 and thus P0 wounds P1. P1 is now shown crossed. P1 relinquishes ownership of block *B* and services the buffered request for the block (this was the request by P2 made in (ii)). Note the probe messages are only used for wound-wait algorithm coordination and do not interact with or change the coherence ordering in the system. As per the base coherence protocol, P1 sends the data to P2. Since P2 was also wounded, P2 simply forwards the data downstream, in this case to P0. Again, as discussed in the footnote, P2 may decide to use the data and complete its transaction if possible.

Figure 4-11 (v) shows the system state after the wound has taken effect for block *B*. Now, processor P0 has block *B* in exclusive owned state (M) and is not waiting for any other processor. Meanwhile, P1 and P2 restart because they were wounded. P1's request for block *A* is merged with its earlier outstanding, and not yet serviced, request if necessary (this occurs as per the normal functioning of the MSHR). P1 re-issues the request for block *B* at time t_{13}. The request is forwarded to the owner of the block P0. Now, P0 has higher priority and thus P1 waits for P0. A corresponding marker message is sent at t_{14}.

[8] Optimizations are possible and we discuss them later. Specifically, Rosenkrantz et al. [144] discuss a modified wound-wait algorithm and that can be applied here.

[9] In a modified wound-wait algorithm P3 even though wounded can continue executing and complete because it happens to have all required blocks in its local cache and is not waiting and will not wait for another processor. This breaks down the priority ordering temporarily but may provide better performance. We do not study this optimization in the thesis and leave it as future work.

In (vii), at t_{15}, P2 re-issues the request for block *B* and since P1 was the last requestor to be ordered by the coherence protocol, P2’s request is forwarded to P1. P1 has higher priority than P2 and thus P1 forces P2 to wait. No probe is sent upstream because P2 cannot wound P1.

Interaction of probe and marker messages. Since the probe and marker messages can get out of order, additional book-keeping is required to track these messages. If a processor receives a probe for a block not cached locally, the processor ignores the probe because this implies a data response has crossed the probe in the network and the processor sending the probe up will receive a data response—the coherence protocol queue is being serviced. Multiple probes may be sent upstream on the same coherence protocol chain depending upon the order in which processors enter the protocol queue. Of course, these probes upstream can proceed in any order and they can bypass each other because they only determine whether a block can be retained by a processor upstream. Again, if a probe is encountered by a node which does not have the block locally cached, the probe can simply be ignored. *A probe does not expect any response from the processors upstream.* Further, probes are only sent upstream—thus they will eventually terminate when they either reach the root of the chain or they reach a processor that cannot be wounded (depending upon the priority of the receiving and probing processors).

A corner case is when a processor immediately re-issues a request to the same block which it serviced and then a probe is received from below. For a processor to receive a probe, it must have already sent a marker message. Thus, if a marker message has not yet been sent then this probe can be ignored. If a marker message has been sent, and if the incoming probe’s source is different from the target of the marker message, even then the probe can be ignored. This is because of the way coherence protocol chains are constructed in a non-nack-based protocol. When a processor services a request, and then re-issues the request, the processor goes to the tail of the existing chain—thus it cannot insert itself before the recipient in the chain. A discussion of this was conducted in detail earlier in Section 2.1.2.

4.4.3 Handling the coherence protocol shared state

Often, within a critical section, a processor may read a shared location, operate upon the value and write a new value to the same location. The read operation brings the corresponding cache block locally in a shared state and the subsequent write results in an upgrade operation where the processor requests exclusive ownership of the cache block so that the processor can update the block. External invalidation requests to shared blocks typically cannot be deferred because no processor exclusively owns the block (upgrades in some protocols may not expect an acknowledgement). These requests must be serviced without delay and may trigger a misspeculation (violation in atomicity of the transaction). To reduce the probability of such upgrade-induced misspeculation, we employ instruction-based prediction to reduce the necessity of requiring upgrades following misspeculation.

The basic idea behind the predictor is as follows. Load operations within a critical section (for SLE and TLR, this corresponds to the period they are executing in an optimistic lock-free mode) are recorded and any store operations within the critical section to the same address results in the predictor update occurring corresponding to the appropriate load operation. For out-of-order processors, the predictor update must occur at instruction commit because only then does the processor know for certain if the memory operation occurred within the transaction (out-of-order processors issue memory operations without regard to program order but instruction retirement is in program order). The predictor is indexed by instruction address. Instruction-based predictors for optimizing read-modify-write patterns as above have been proposed earlier [84]. Address-based techniques for optimizing read-modify-write patterns have also been proposed [32, 157].

Cache blocks that are only read within critical sections are brought into the cache in a shared state. If repeated upgrade-induced violations occur, the processor can issue exclusive requests for all blocks accessed within the critical section, obtain the blocks in owned state and defer external requests to such blocks. Doing so guarantees a successful TLR execution even without the above optimization.

We show in the evaluation chapter that the use of the simple read-modify-write predictor as described above substantially improves performance of the base system without TLR as well as with TLR.

4.4.4 Performance interactions of timestamp order and coherence order

The order in which processors execute *conflicting* critical sections is determined by timestamps—called *timestamp-order.* The order in which coherence permission and data for cache blocks move around in the system is determined by the coherence protocol order—the order in which requests were received by the coherence protocol—called *coherence-order.* The defer-

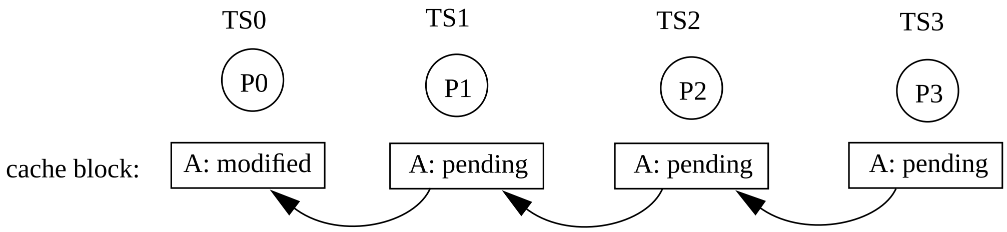


Figure 4-12: Timestamp-order is identical to coherence-order. *The figure shows how a queue is maintained and data transfer occurs. TS0 < TS1 < TS2 < TS3 and thus the coherence ordering is identical to the timestamp ordering. This represents an ideal condition.*

ral-based TLR algorithm maintains a separation between timestamp-order and coherence-order thus leaving the protocol unchanged and maintains the protocol correctness conditions outlined earlier in Section 2.1.2.3. However, if timestamp-order and coherence-order do not match, performance issues may arise. We discuss three cases below. The three cases are: a) timestamp-order is identical to coherence-order, b) timestamp-order is exactly reverse of coherence-order, and c) timestamp-order approximates coherence-order.

Timestamp-order is identical to coherence-order. This situation occurs when the timestamps faithfully represent the order in which various processors issue requests and are ordered by the coherence protocol. In addition to the absence of any locking overhead, the data transfer itself is optimized and occurs with minimum latency. No queue breakdowns occur and all processors issue a single request for the cache block, operate upon the block, and then forward the updates to the next requestor in line without any explicit handshaking.

Figure 4-12 shows four processors P0, P1, P2, and P3 with timestamps TS0, TS1, TS2, and TS3 respectively. All processors request the same cache block *A* thus exhibiting data conflict where P0 is ordered before P1 in the coherence protocol ordering, then P2, and then P3. Assume the timestamp ordering is as follows: TS0 < TS1 < TS2 < TS3; TS0 has the highest priority and TS3 has the lowest priority. P0 is currently executing its optimistic lock-free transaction and has accessed cache block *A*. P0 defers (and buffers) P1's request for *A*. P2's request is buffered by P1 and P3's request is buffered by P2. P0 operates on *A*, completes its critical section and then responds to P1's request with the latest data for *A*. Subsequently, P1 operates upon the data, executes its own transaction, and on completion, respond to P2’s request with the latest data for *A*, and so on.

Thus, while processors attempt to execute the same transaction, they are automatically ordered on the data request itself and no explicit lock requests are generated. This direct transfer of data, coupled with the absence of lock requests and overhead, provides the intuition for high performance in the presence of data conflicts. Further, while P0 is operating on *A*, other processors wait for the latest copy rather than introduce contention in the system by repeatedly requesting lock and data.

The behavior is similar to hardware queue locks [50, 141] but now the queueing is occurring on the data itself and no lock requests are generated.[10] Removing explicit lock requests and locking overhead under contention reduces network contention and latency. The properties of the wound-wait proposal hold here and retries are eliminated because lower priority processors simply wait rather than consume system resources.

Later we discuss an optimization where we can relax timestamp order selectively and achieve the ideal behavior as discussed above even when the coherence-order and timestamp-order do not match.

Timestamp-order is reverse of coherence-order. This situation occurs when various processors issue requests and are ordered by the protocol in a reverse order than their timestamps. While no locking overhead occurs, the data transfer itself is not optimized because a queue breakdown occurs because processors in the queue end up getting wounded repeatedly. Hot spotting still does not exist and quite possibly performance may still be better than the base case. However, the performance is not optimal.

Similar to the earlier example, Figure 4-13 shows four processors P0, P1, P2, and P3 requesting the same cache block *A* thus exhibiting data conflict where P0 is ordered before P1 in the coherence protocol ordering, then P2, and then P3. Assume the timestamp ordering is as follows: TS3 < TS2 < TS1 < TS0; TS3 has the highest priority and TS0 has the lowest priority. P0 is currently executing its optimistic lock-free transaction and has accessed cache block *A*. P0 receives P1’s request for *A*. P0 is wounded and restarted because P1 has higher priority than P0. Similarly

[10] Because no lock overhead is experienced, performance is expected to be better than QOLB and potentially the same as QOLB with collocation.

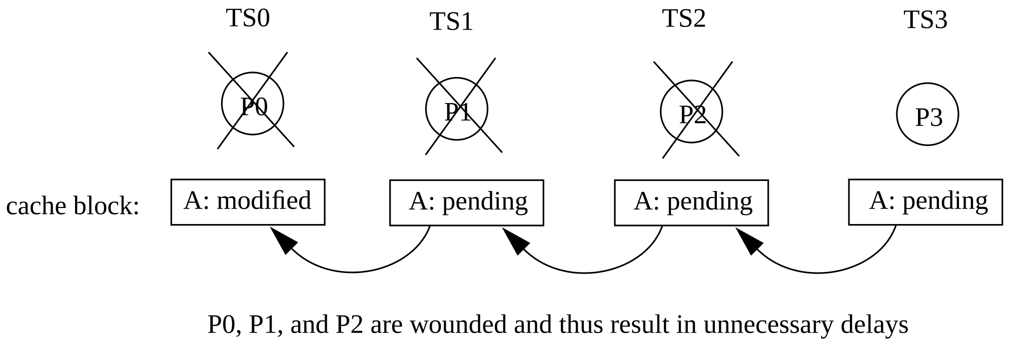


Figure 4-13: Timestamp-order is reverse of coherence-order. *The figure shows how a queue is maintained and data transfer occurs. TS3 < TS2 < TS1 < TS0 and the coherence ordering is exactly reverse of timestamp ordering. Probes from processor later in the coherence chain will propagate upstream wounding processors upstream. The chain repeatedly breaks down although the processor with the earliest timestamp still successfully complete its transaction.*

P1 is wounded by P2 and P2 is wounded by P3. Depending upon the timing in the system, in the worst case P3's request will be forwarded as a probe all the way up to P0. A queue breakdown occurs and P3 obtains the block for writing after a latency of the data transfer from P0 to P1 to P2 and then to P3.

However, the actual probability of such a long chain being created is very low because P0 will be wounded right when P1's request is received by P0. P0 will thus send the block to P1. The probe from P3 to P0 is not serialized but parallelized by the presence of other probes from P1 and P2 that are also flowing upstream. Even though a successful lock-free execution occurs, a delay is experienced before the processors can get through their critical sections.

Timestamp-order is approximate to that of coherence-order. This occurs when the coherence-order and timestamp-order are mostly similar except for one or two processors. An example of this is shown in Figure 4-14 where four processors P0, P1, P2, and P3 are shown. The timestamps are ordered as: TS0 < TS2 < TS3 < TS1. Thus, P1 has the lowest priority. P0 defers P1's request. P2 wounds P1 but since P0 has higher priority than P2, P2 ends up waiting. P3's request is also deferred by P2 because P2 has higher priority than P3. When P0 completes its critical section, the cache block is forwarded to P1 as per the coherence protocol order. However, P1 has been wounded and the data block is forwarded to P2 without P1 operating upon it. An additional latency has thus been experienced by P2 because of the presence of a wounded processor P1

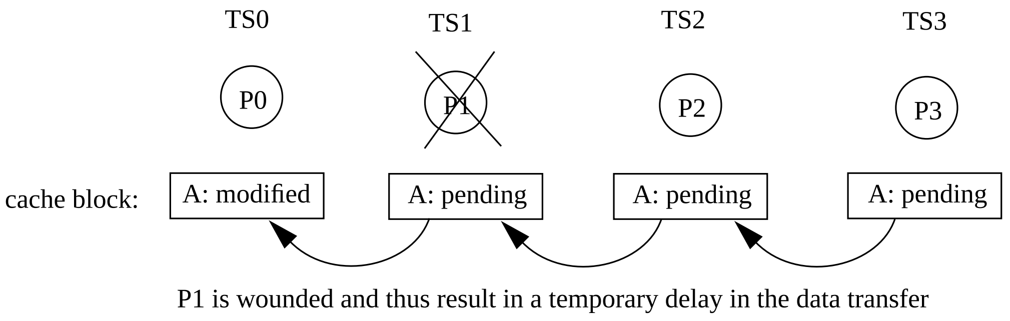


Figure 4-14: Timestamp-order approximates coherence-order. *The figure shows how a queue is maintained and data transfer occurs. TS0 < TS2 < TS3 < TS1 and thus the coherence ordering is approximate to the timestamp ordering.*

in the chain. As discussed earlier in Section 4.4.2.3, P1 might attempt to nevertheless execute its critical section by using the data block it receives even though P1 has been wounded. We do not discuss this optimization. We however discuss another optimization in the next section where the timestamp order may itself be selectively relaxed and prevent a wound from occurring.

4.4.5 Selectively relaxing timestamp order

Deadlock is not possible if only one cache block is under conflict within the transaction because a cyclic wait is impossible (the head node of the coherence chain is always a stable state and does not wait for anyone else). Timestamps serve two functions: providing starvation freedom and deadlock freedom. In protocols such as the Sun Gigaplane (which are non-nacking protocols), a queue of requests is automatically formed for a given block if multiple processors issue ownership requests while the block states are pending and the deferred queue is serviced in a serial order. In such situations, strict timestamp order can be relaxed. Thus, a timestamp-induced restart can be temporarily avoided if only a single cache block is contended for. However, if an additional cache block is accessed that may deadlock (i.e., generates a cache miss), then the timestamp order must be enforced.

4.4.6 Controlling misses

While wound-wait minimizes protocol interference and prevents unnecessary restarts, if a processor has been wounded, it might be useful to not issue an additional request until any previous requests (that were pending and thus resulted in a wound) have been serviced. While the wound-wait approach reduces coherence protocol interference, an optimization such as controlling misses may be useful in NACK-based approaches.

4.4.7 Implementation-specific resource constraints

In this section, we discuss the impact of implementation-specific constraints—cache size and associativity, write buffer size, deferred queue size, operating systems scheduling quantum, and finite size timestamps—on TLR.

4.4.7.1 Cache size and associativity

TLR has resource limitations similar to SLE. If the cache is used to track the lock and data accesses for a critical section, the finite size of the cache restricts the data set size that can be tracked speculatively. The associativity of the cache also places a limit because conflict misses force evictions of cache blocks. Well known and well understood techniques, such as victim caches [79], for handling such situations exist. Victim caches are small, fast, fully associative structures that buffer cache blocks evicted from the main cache due to conflict and capacity misses. The victim cache can be extended with a speculative access bit per entry to achieve the same functionality as a regular cache. For example, if the system has a 16 entry victim cache and a 4-way data cache, the programmer can assume any transaction accessing 20 cache blocks or less is ensured a lock-free execution under TLR.

4.4.7.2 Write buffer size

Since the write buffer buffers speculative memory updates, its size restricts the number of static block addresses that can be written to within a critical section. Since writes are merged in the write buffer and memory locations can be rewritten within the write buffer (because atomicity is guaranteed), the number of unique cache blocks written to within the critical section determines the required size of the write buffer.

4.4.7.3 Deferred queue size

For the implementation we provide, TLR requires sufficient buffering for deferred requests. The size of buffering can be calculated a priori and is a function of the system size and victim cache size. In any case, TLR like SLE can guarantee correctness under all circumstances and in the presence of unexpected conditions can always acquire the lock, but at a loss of transactional properties.

4.4.7.4 Scheduling quantum

Another resource constraint is the operating systems scheduling quantum—it must be possible to execute the critical section within a single quantum. The time determination can be performed a priori using worst case analysis. The quantum length is typically much larger than the time it takes to execute most critical sections and this is not expected to be an issue.

4.4.7.5 Finite size of timestamps

TLR uses timestamps only for conflict resolution. The size of the timestamp eventually only affects the fairness aspects of TLR and not its safety aspects (serializability and deadlock freedom). In this section we discuss issues related to the finite size of timestamps (bounded timestamps).

Bounded timestamps have received considerable attention in the past [1, 37, 38, 74, 77]. The problems introduced by bounded timestamps were critical for the problem domains these researchers were focusing on because they relied on the absolute ordering introduced by the timestamps. Bounded timestamps introduce a problem because of wrap-around that occurs once the timestamp size is exceeded, thus breaking the strict monotonically-increasing property provided by unbounded timestamps. For example, Jacobsen et al. [77] needed to ensure packets did not get lost if received out-of-order. Israeli and Ming [74], Dolev and Shavit [37] and Dwork and Waarts [38] were concerned with problems where timestamps were used to construct a total order of events. TLR however only uses timestamps for determining which thread has a higher priority and thus

P0	P1
110 00	110 01
110 00	110 01
110 00	110 01

P0	P1
110 00	111 01
100 00	111 01
101 00	111 01
⋮	⋮
110 00	111 01
111 00	111 01
100 00	111 01

Figure 4-15: Impact of finite size of timestamps on fairness. *Two examples are shown where one processor may experience starvation.*

will win a conflict. The correctness condition of serializability is not affected by the limited size of the timestamp though a strict notion of fairness may be momentarily compromised.

The timestamp size contributes to the probability of timestamp wrap-around. If the timestamp size is large, wrap-around probability is small and any overhead associated with dealing with wrap-around is amortized. Thus, selecting the size of the timestamp involves a trade-off between wrap-around effects and the overhead of dealing with wrap-around. However, the dynamic component must have a minimum size—a size of 0 results in the timestamp being completely made of the static component and thus being static in nature.

While bounded timestamps do not affect the safety property of TLR, they may introduce fairness issues for TLR. Consider Figure 4-15. Two examples are shown. The timestamps of two processors P0 and P1 are also shown. Consider the example on the left. P0 initially has timestamp 11000 and every successful TLR execution results in the new timestamp also being 11000 because of wrap-around and of an update granularity greater than 1. Thus P0 wins all conflicts, and unfairly executes its critical section repeatedly while P1 keeps failing and restarting. Now consider the example on the right. In this case, P0 will *always* win all conflicts. This happens because P0 and P1 both have reached the maximum count for the left side bits and employ the processor identifier tie breaker to decide a conflict winner. Since the tie-breaker is static, P0 will increment its timestamp beyond 11100, wrap-around, and start at 000000. Meanwhile, P1's timestamp stays at 11101. P1 lost the initial conflict with processor 0 (when its timestamp was 11100) and now will lose all conflicts with P0. Once P0's timestamp reaches 11100, the cycle repeats and P1 starves.

For TLR, the wrap-around time must allow all conflicting processors to succeed at least once in their lock-free execution in order to maintain fairness. Timestamps in TLR need only guarantee that all processors eventually successfully win all conflicts—they do not have to enforce a strict first-come first-served discipline. To ensure freedom from starvation, no pathological condition must occur where a given processor continuously loses conflicts and never becomes the earliest timestamp in the system. The two figures above show examples where such behavior may occur. The wrap-around time can be quite large depending upon the size of the dynamic component of the timestamp. Assume a timestamp update occurs every successful TLR execution.[11] For a 24-bit dynamic component, a wrap-around would occur after 16,777,215 successful TLR executions on a given processor. Assuming 100 cycles per critical section, a timestamp wrap-around would occur approximately every 1.7 billion cycles if the processor executes the critical sections repeatedly in a tight loop.

One approach to handling bounded timestamps is the use of *k*-bit unsigned integers in a modular *k*-bit space as is commonly used in TCP/IP sequence numbers (*k* for TCP/IP is typically 32) [77, 92]. If *s* and *t* are timestamp values, $s < t$ if $0 < (t - s) < 2^{k-1}$, computed in unsigned *k*-bit arithmetic. For fairness, the size of *k* should be sufficient to allow each processor to execute at least one lock-free critical section successfully. The update of the local clocks are again performed using Lamport’s rules outlined earlier.

Other techniques may be employed. On a wrap-around of a local counter, a global reset of all other local clocks may be performed. This requires a message from the processor whose counter is going to wrap-around to all other processors. For the wrap-around time calculated above, the time for the message to reach all other processors must be less than 1.6 billion cycles—a very reasonable assumption. On receiving a wrap-around message, a processor resets its local clock.[12] Doing so eliminates the problem illustrated in Figure 4-15 where a processor stays at the saturated counter value and keeps restarting because another processor wraps around and repeatedly beats

[11] Since timestamps affect fairness, starvation freedom can still be guaranteed if this update condition is relaxed. Further, locally succeeding non-conflicting critical sections do not have to increment the clock.

[12] This will involve a misspeculation if the processor is in TLR mode because a processor in TLR mode must not update its local clock while it is speculating. A misspeculation is not necessary if no conflicts have been detected as yet.

out this processor. The overhead of sending a broadcast message every 1.6 billion cycles is quite small. This message is a point-to-point message and does not require a coherent bus.

The key as to why a global reset can be arbitrarily performed lies in the way TLR employs timestamps. TLR does not use timestamps to explicitly order events and does not use them for causality. TLR only uses timestamps for conflict resolution and providing a degree of fairness (primarily starvation freedom). Thus, a global reset may result in a temporary jitter where fairness is lost because the priorities get reordered temporarily but correctness is always guaranteed by the SLE commit mechanism.

Another way to handle wrap-around is for each node to remember the last timestamp received from a given node. Then a comparison with that number for every subsequent timestamp helps the current node detect a wrap-around. At such a time, the node may decide to reset its own counter.

As we have seen, numerous mechanisms exist to handle the issue of fairness that may result due to the limited size of timestamps in TLR.

4.5 Algorithm invariants

In the TLR algorithm described in Section 4.3.2, three key invariants must hold:

a) The timestamp is retained and reused following a conflict-induced misspeculation

b) Timestamps are updated in a strictly monotonically increasing order following a successful TLR execution

c) The earlier timestamp request never loses a resource conflict and thus succeeds in obtaining ownership of the resource

If TLR is applied, these invariants collectively provide two guarantees:

1. A processor eventually has the earliest timestamp in the system
2. A processor with the earliest timestamp eventually has a successful lock-free transactional execution

The two properties above result in the following observation:

> "In a finite number of steps, a node will eventually have the earliest timestamp for all blocks it accesses and operates upon within its optimistic transaction and is thus guaranteed to have a successful starvation-free lock-free execution."

In the next section we discuss the implications of the above observation on the programmability and stability of multithreaded programs.

4.6 Programmability and stability impact of TLR

We will now discuss the implications of TLR on programmability and stability of multithreaded programs. The guarantees discussed in the earlier section hold only if TLR can be applied. In the presence of constraints, such as resource limitations and uncacheable requests, the guarantee of stability properties become conditional. These limitations make the guarantee of stability properties conditional. Such a guarantee can be constructed using the size of the victim cache and the scheduling quantum. Some of these parameters can be architecturally specified. For example, if the system has a 16 entry victim cache and a 4-way data cache, the programmer can be assured any transaction accessing 20 cache blocks or less is ensured a lock-free execution. A programmer expecting guaranteed behavior will need to be aware of precise specifications. For a critical section to be executed in a wait-free manner, the lock must be positively identified. TLR uses SLE, which must be implemented to identify all locks that satisfy a certain idiom. The spin-wait loop of the lock acquire will only be reached if TLR has failed, thus giving the programmer a reliable method of detecting when wait freedom has not been achieved. This is an area of future work.

Until now we have assumed the lock and the protected data are in different cache blocks. While this is the most common implementation for performance reasons, situations may occur where the lock and data are collocated. In these situations, write operations to data collocated in the same cache block as the lock may result in a misspeculation being triggered because the hardware assumes the lock is being acquired (the granularity of conflict detection is the coherence granularity). TLR properties can still be guaranteed. In the event that the processor needs to acquire a lock because it has received an invalidation to the cache block containing the lock variable, the processor can simply request exclusive permissions for the lock variable but still apply partial lock elision. In other words, the processor acquires the cache block holding the lock in exclusive state but does not write the lock variable. The processor defers incoming requests to the cache block containing the lock the same way it defers requests to the data by employing TLR. While concurrent execution will not occur because the lock is temporarily unavailable to other processors, the properties of TLR are maintained because the lock is never written to. By not writing the lock, in the event of a failure or operating systems descheduling event, the lock is left in a free state and all speculative updates discarded.

The behavior of this mechanism becomes quite similar to that of IQOLB [141] because now the processors are queueing up on the lock. The difference now is, unlike IQOLB, the lock is not written to and explicitly acquired.

Multiple nested locks can also be elided if hardware for tracking these elisions is sufficient. This was discussed in Section 3.10. If some inner lock cannot be elided due to an inability to track multiple elisions, the inner lock is simply treated as data. This does not change TLR's properties: the execution is still lock-free and lower priority threads will be deferred by higher priority threads temporarily. It is the outermost lock that controls whether TLR's properties are met because the outermost lock demarcates the lock-free transaction. Eliding the outermost lock is sufficient to maintain TLR properties.

4.6.1 Restartable critical sections

SLE provides light-weight support for restartable critical sections. This is a direct result of the failure atomicity guarantee provided by SLE (in the absence of conflicts) and TLR (even in the presence of conflicts). Sometimes it is desirable for the operating system to restart certain threads from some point of execution without affecting correctness—for example if the thread executions are deadlocked. The presence of locks makes this difficult because the thread might be in a critical section and may have modified shared memory.

SLE provides hardware support for buffering speculative updates within critical sections and exposes these values only at the time the critical section execution is committed. Thus, if a thread in SLE mode is terminated, the speculative updates can be discarded and the execution can be thought to restart from just before a lock acquisition. The concept of a lightweight restartable critical section is quite powerful and a useful functionality for the operating system to exploit.

Restartable critical sections allow the underlying blocking synchronization primitive to be made non-blocking and we discuss this next.

4.6.2 Non-blocking behavior

As discussed in Section 2.2.3, a synchronization technique is non-blocking if some thread will complete an operation in a finite number of steps, regardless of the relative execution speeds of the processes [64]. The non-blocking condition guarantees the system as a whole makes progress despite individual halting failures or delays. If TLR can be applied successfully to the execution, a non-blocking execution of the critical sections can be obtained because TLR guarantees a lock-free execution even in the presence of conflicts if sufficient buffering is available. This behavior is a direct result of the software wait on the lock variable being eliminated. If a process is descheduled, a misspeculation is triggered and the lock is left in a free state with all speculative updates within the critical section discarded. Other threads scheduled may continue to operate on the protected data structure.

Non-blocking behavior is guaranteed only if the critical section can be executed completely within a single operating system scheduling quantum. If the execution is longer than a quantum, then the lock-based execution must be relied upon since the lock-free execution may never complete. This issue is an example of a resource limitation that can be addressed at the operating systems level and is an area of future work. Another situation where non-blocking behavior is useful is when a page fault happens within a critical section. Without operating systems support, this will result in TLR falling back upon the explicit lock acquisition sequence. With some operating systems support, a page fault could be triggered within the lock-free critical section and the execution would need to appear as if the page fault occurred just prior to the start of the critical section. We do not discuss this in detail and leave it as future work.

4.6.3 Wait-free behavior

The wait-free behavior follows from the non-blocking behavior discussed above but subject to a stronger guarantee of starvation freedom. The threat of resource limitations makes this a conditional behavior—conditional on the ability of the processor to buffer critical section data accesses and buffer deferred incoming requests. As discussed earlier, a guarantee can be constructed based on the size of the victim cache.

```
Thread 1         Thread 2

Lock (A)         Lock (B)
    .                .
    .                .
    .                .
Lock (B)         Lock (A)
    .                .
    .                .
    .                .
Unlock (B)       Unlock (A)
    .                .
    .                .
    .                .
Unlock (A)       Unlock (B)
```

Figure 4-16: Deadlock possibility in programs using incorrect locking hierarchy. *Two threads acquiring locks in reverse order will deadlock in most systems. With TLR, since locks are not acquired, lock A and lock B will always be free and thus no cyclic wait will be present. This program will be executed correctly using TLR and no deadlock will occur.*

4.6.4 Handling deadlocks in locking hierarchies

An interesting side effect of TLR is its ability to prevent certain types of deadlocks from being exposed. Specifically, deadlocks that occur due to an incorrect locking methodology. Consider Figure 4-16. Two threads, thread 1 and thread 2, acquire locks A and B in opposite order. Note, this program will deadlock in most systems if these threads are executed concurrently because thread 1 first acquires A and then will wait for B while thread 2 first acquires B and then will wait for A. A cyclic wait will occur. With TLR, since locks are not acquired, A and B will always be free and thus no cyclic wait will be present. This is an example where a deadlocked program will be executed correctly using TLR transparently. TLR can hence improve the robustness of a program and prevent such deadlocks from occurring. Again, this behavior is conditional on sufficient resources being present to buffer the entire atomic region and the nesting being tracked. TLR provides an opportunity to improve the reliability of multithreaded programs by preventing the negative effects of locks from being exposed because TLR can elide locks even in the presence of conflicts. During debugging and testing, the ability to turn of TLR is necessary else finding and correcting such programming errors will be difficult.

4.6.5 Masking data races

By treating un-timestamped requests as deferrable, a successful lock-free execution is achieved even in the presence of data races. Such a request is assumed to have the latest timestamp in the system (and thus the lowest priority) and the un-timestamped request is atomically ordered after the current critical section. Since a data response is not sent until after the critical section, the requestor cannot consume the data and hence is ordered with the correct value. TLR, in effect, is masking the data race and forces the data race to be ordered after the critical section completes. Thus subtle undesirable data races may be prevented from being exposed in the system for a given execution. The limitation of this is when the data race may have been explicitly added by the programmer for performance reasons—TLR will prevent such a data race from occurring.

4.7 Related work

Since TLR builds upon SLE, TLR borrows the related work discussion for SLE from Section 3.13. In this section, we discuss TLR related work beyond SLE.

Lock-free and wait-free synchronization. Software only lock-free schemes have been shown to perform poorly as compared to lock-based schemes because of excessive data copying to allow roll-back [5, 17].

Three lock-free mechanisms using hybrid software and hardware support are the load-locked/ store-conditional instructions, transactional memory, and the oklahoma update.

Transactional Memory [66] used NACKs for performance reasons. To increase the probability of a successful lock-free execution, a processor could refuse to service an incoming conflicting request. However, transactional memory did not use NACKs for livelock avoidance; it used an exponential backoff mechanism implemented in software.

The Oklahoma Update [158] did not provide starvation freedom although it did provide liveness by relying on a two-phase commit process and sorting memory addresses in hardware to order their request and deferring requests appropriately. However the paper does not provide any performance evaluation.

Software lock-free mechanisms such as Software transactional memory [149] uses software primitives to implement transactions but performs poorly with respect to its lock-based counterparts. Software-only proposals suffer from difficulty of use and a lack of generality and often poor performance. Wait-free proposals have suffered from performance limitations in the absence of failures.

Database concurrency control and deadlock issues. Transactions are well understood and well studied in database literature [56]. The use of timestamps for resolving conflicts and ordering transactions in database systems has been well studied [14, 144]. Holt [71] provides a good framework for reasoning about deadlocks in computer systems. Extensive work has been done in optimistic concurrency control (OCC) for database systems [90]. OCC was proposed as an alternative to locking in database management systems. OCC involves a read phase where objects are accessed (with possible updates to a private copy of these objects) followed by a serialized validation phase to check for data conflicts (read/write conflicts with other transactions). This is followed by the write phase if the validation is successful. TLR does not have a serialized validation phase and exploits hardware techniques to provide transactional behavior. In spite of extensive research, OCC techniques have not been popular because of key limitations [124].

Lock-based synchronization. Lock-based synchronization has been extensively studied in literature. These techniques attempt to optimize the lock and data transfer operations [10, 50, 81, 120, 141]. The techniques are not lock-free. These techniques suffer from locking overhead and serialization due to lock acquisitions.

Martínez and Torrellas introduced *Speculative Locks*, allowing speculative threads to bypass a held lock and enter a critical section [117]. At any time the lock is always acquired by one thread which is non-speculative—also called the safe thread. Speculative threads could then become non-speculative after a lock was released by the non-speculative thread if no data conflicts were detected by the speculative threads and the speculative threads had completed their critical sections. In the presence of data conflicts, speculative threads always restart and retry the above sequence, competing for the lock and try to become safe threads by attempting to acquire the lock. A free lock is always written to and acquired explicitly by a thread.

In *Speculative Synchronization* [118], Speculative Locks is extended to include the SLE mechanism to be used in the absence of data conflicts. In the presence of data conflicts, rather than falling back on the underlying scheme as SLE does, it adapts by employing Speculative Locks as described above. In the presence of conflicts, threads attempt to become safe; in other words they compete for the lock. Similar to Speculative Locks, in the presence of resource limitations, the speculative threads in Speculative Synchronization stall and wait to acquire the lock.

The above two schemes provide the same forward progress guarantees as SLE. These schemes are not lock-free, experience the limitations of locks, and do not provide the guarantees provided by TLR.

Delaying responses to requests for lock variables for a short time and thus emulating hardware queued locks was proposed earlier [141]. TLR generalizes that notion by applying deferrals to data and to multiple cache blocks simultaneously.

Speculative execution and parallelization. Speculative execution for aggressive implementation of memory consistency models was proposed by Gharachorloo et al. [45] and later extended [48, 143]. Similarly, work has been done in speculative parallelization of programs [86, 154]. While the buffering and speculative execution mechanisms they use are similar to ours, these proposals do not provide lock-free execution of lock-based code and do not address critical section serialization and thus are orthogonal to our scheme.

4.8 Chapter summary

We have proposed Transactional Lock Removal (TLR), a hardware mechanism to convert lock-based critical sections transparently and optimistically into lock-free optimistic transactions and a timestamp-based fair conflict resolution scheme to provide transactional semantics and starvation freedom, if the data accessed by the transaction can be locally cached and subject to some implementation specific constraints. TLR provides both serializability and failure atomicity. We have presented one deferral-based implementation of TLR that does not require changes to the coherence protocol state transitions.

We summarize the contributions of our mechanism under three categories:

1. *Programmability.* TLR simplifies correct multithreaded code development. Reasoning about granularity of locks is not required because serialization decisions are made at run-time based on actual data conflicts and independent of locking granularity. Thus, a critical problem in reasoning about writing multithreaded programs is solved. Cache blocks are the coherence unit and represent a fine granularity for sharing. TLR provide this granularity without programmer involvement.

2. *Stability.* Since locks are not written to and the “wait” on the lock variable is no longer required, properties of lock-free and wait-free execution are achieved transparently. This translates to improved system wide performance, no convoying or priority-inversion dangers, and robust execution in the presence of failing threads. TLR addresses the inherent limitations of the locking construct while maintaining the well-understood critical section abstraction for the programmer.

3. *Performance*. TLR enables high-performance multithreaded execution. Independent of lock granularity, because serialization decisions are made only in the presence of data conflicts and is not based on lock contention, performance of fine-granularity locking is achieved. Further, since a queue of requestors is constructed in the hardware by using the coherence protocol, the data transfers are efficient and low overhead. Programmers can focus on writing correct code while hardware automatically extracts performance.

TLR is the first proposal to address the trade-off among all the above three aspects and provide a robust solution to the synchronization problem. While TLR does tradeoff hardware for these properties, we believe the hardware cost is modest. Additionally, we address the inherent limitations of the locking construct automatically while maintaining the well understood critical section abstraction for the programmer. Subject to resource constraints, our scheme is the first to transparently provide a wait-free execution of a lock-based critical sections.

Software developers can use TLR in several ways. The size of transactions can be architecturally specified thus guaranteeing programmers a lock-free atomic execution of a sequence of memory operations. Such functionality can help programmers write simpler high-performance wait-free algorithms. Some of the hardware support, required for example in identifying critical sections, can be reduced by using appropriate compiler support. Operating systems can exploit the notion of transactional execution to provide strong guarantees and appropriate operating systems involvement can prevent software failures (that affect one thread) to interact negatively with other concurrent threads and allow other threads to continue execution.

Chapter 5

Performance Evaluation Methodology

In this chapter we describe our performance evaluation methodology. We use simulation techniques for evaluating performance. Two components of the simulation environment are the simulator and the benchmark binaries the simulator executes. We describe the simulator in Section 5.1 and the compiling infrastructure for generating the binaries in Section 5.2. In Section 5.3 we describe our target system for simulations and the configuration parameters. Section 5.4 describes the benchmarks we use for our performance evaluation.

5.1 SimpleMP simulation environment

We use SimpleMP, an execution-driven simulator for executing multithreaded binaries. The simulator is partly derived from the Simplescalar 3.0 toolset [22] but is completely rewritten. SimpleMP models accurately an out-of-order processor and a detailed memory hierarchy in a multiprocessor configuration. Values are copied and passed in the processor cores, physical registers, and architectural registers. Data values are also stored throughout the memory hierarchy (e.g., in caches, write buffers, and network packets).

To model coherency and memory consistency events accurately in a multiprocessor simulation, the processors must operate (read and write) on data in caches and write buffers (unlike *sim-outorder* in Simplescalar where the caches only store tags and the actual data is accessed directly from memory). The processor model in SimpleMP accesses data from these buffers directly instead of from a flat memory space, thus allowing accurate modeling of coherence and memory consistency. Contention is modeled at all levels in the memory system. A page table is implemented to translate virtual addresses to physical addresses. Coherency is maintained using physical addresses.

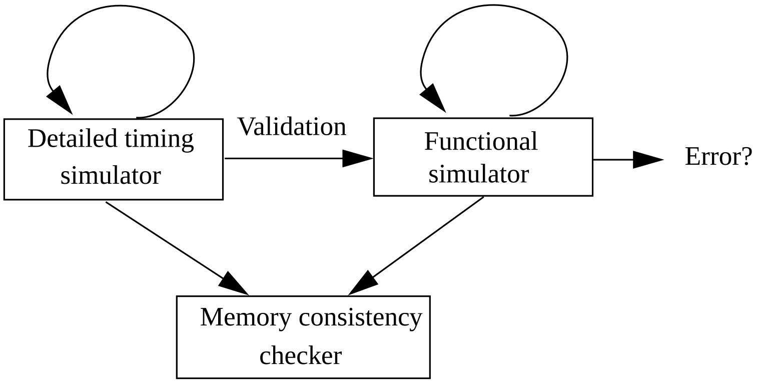


Figure 5-1: Simulation methodology. *The timing and functional simulators have their own register and memory space. They only communicate via a memory consistency checker to check for memory consistency violations. The detailed timing simulator informs the functional simulator when to execute based on the memory consistency model implemented by the timing simulator. Hence, this is a validation sequence and not a verification sequence since the timing and functional simulators cannot be totally decoupled in a multiprocessor environment.*

To ensure correct simulation, a functional checker simulator executes in parallel with the detailed simulator **only** for checking correctness. The functional simulator works in its own private memory and register space, and is robust enough to validate aggressive TSO implementations. The functional simulator has functional write buffers. When a memory consistency event is considered completed by the timing simulator, the functional simulator is asked to drain its functional write buffer entry. No values are exchanged between the functional and timing simulators. The interaction of the two simulators is shown in Figure 5-1.

Random perturbations are introduced in various segments of the simulator to ensure the simulation does not have an artifact of the way the simulator is written. For example, the main simulator loop goes through all processors in sequence in a cycle and executes the processor cores and event queues. The order in which the processors are sequenced through is selected at random. Undesirable artifacts may arise if this is not done. For example, if the simulation order is first processor 1, then processor 2, and so on every cycle, the first processor will generate a miss first and have access to system resources first and will provide biased behavior resulting in load imbalance issues since one processor unfairly gets priority in memory operations and may, for example, succeed in acquiring locks first. As a result, one processor starves the other processors. Such behavior may actually be true in real systems if clock skew forces one processor to always be faster in reaching the memory system and thus obtain resources before others. Since this is a system specific issue, we reduce the probability of such biases from occurring and introduce randomness in the order in which processor obtain resources. Similarly, small perturbations (on the order of a few cycles) are introduced throughout the memory system including the data network, network controllers, and coherence input queues. The ordering requirements, if any, of each network are maintained.

5.2 Compiling infrastructure

We use the PISA instruction set architecture [22] and the `gcc` compiler developed for that architecture. We wrote linker scripts to generate thread-safe binaries using the simplescalar compiler. Simplescalar `libc` libraries are not thread safe and we annotate segments in the binary at link-time to distinguish between system code and application code. Application code is thread-safe. In addition, the simulator supports both `FORK` and `SPROC` models for parallel computation. The PARMACS macros are used for compiling the parallel versions of the applications. All benchmarks are compiled with the -O3 option and all synchronization code is inlined. The compile infrastructure is shown in Figure 5-2. Only user-level instructions are modeled. A single process is assigned to each processor.

5.3 Target system and configuration

The processor configuration is common for all simulations and we discuss it first. We then discuss the chip multiprocessor (CMP), symmetric multiprocessor (SMP), and distributed shared-memory (DSM) configurations.

Processor configuration. Each processor is an aggressive out-of-order processor implementing total store ordering as its memory consistency model. The implementation of the memory consistency model is similar to that proposed by Gharachorloo et al. [45] where loads are aggressively issued and the load/store queue is snooped to check for any memory consistency violations. The processor configuration is shown in Table 5-1.

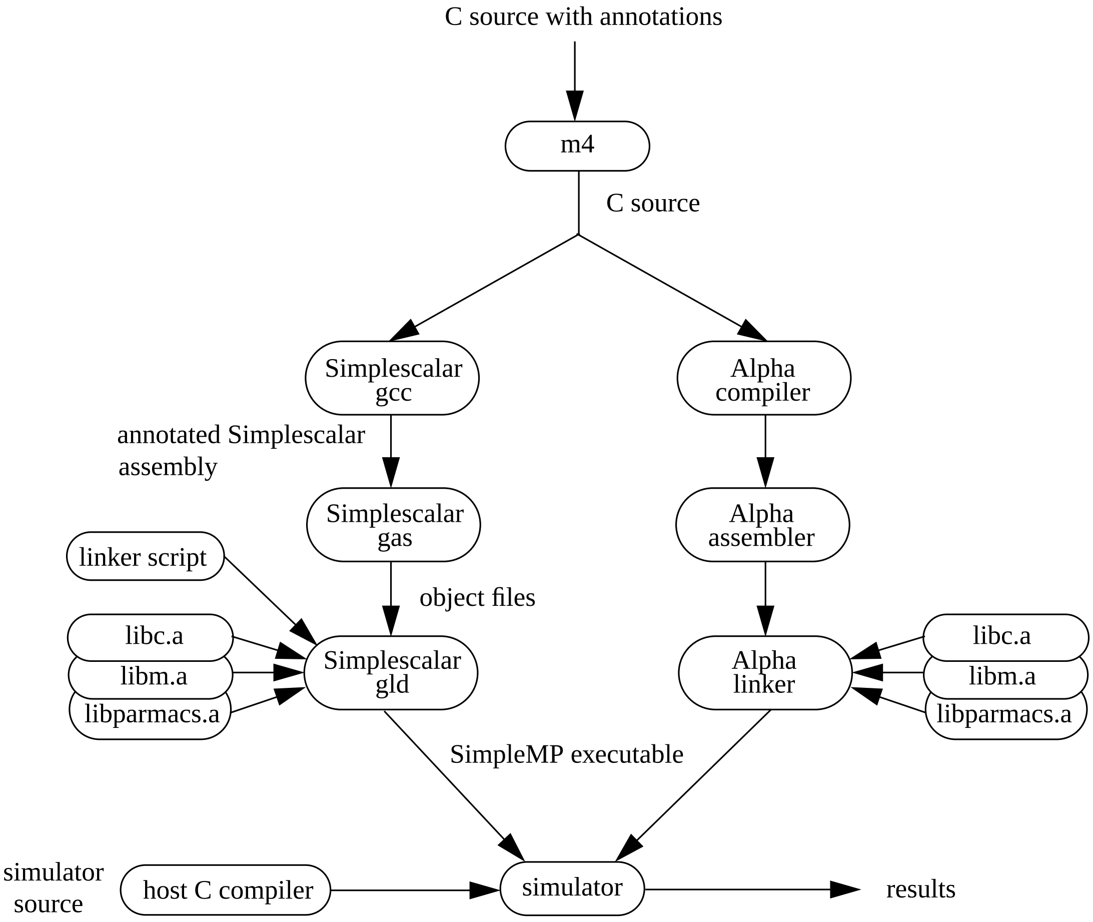


Figure 5-2: Compile infrastructure. *SimpleMP can simulate Alpha and PISA instructions.*

Chip multiprocessor (CMP) configuration. This system configuration has a single level cache (consisting of L1 caches) hierarchy with each L1 cache kept coherent using a broadcast network implementing a coherence protocol similar to the Sun Gigaplane. The data network is a point-to-point high-bandwidth direct network modeled after the Sun Gigaplane-XB. The level two cache and memory are off-chip. Figure 5-3 shows the CMP target system and Table 5-2 lists the CMP system parameters. SLE and TLR are evaluated for this configuration.

Symmetric multiprocessor (SMP) configuration. We add an additional level of caches (consisting of L2 caches) to the CMP configuration. Coherence is maintained across the L2 caches. Similar to the chip multiprocessor configuration, the broadcast network is high-bandwidth

Table 5-1: Processor configuration

Processor	Clock	1 GHz (1 ns cycle time).
	Fetch	16 entry instruction fetch queue.
	Branch prediction	8K entry combining predictor, 8K entry 4-way BTB, 64 entry return address stack, 3-cycle branch mispredict redirect penalty.
	Issue/execute/commit	out-of-order issue/execute 8 instructions per cycle, in-order commit of 8 instructions per cycle.
	Reorder buffer Load/store queue	128 entries. 64 entries.
	Functional units	pipelined, 8 alus, 2 multipliers, 4 floating point units, 3 memory ports.
	Write-buffer	64 entry, non-merging, each entry 64-byte wide.
	Load issue policy	issue loads to memory system as soon as address known.
	Memory consistency	total store ordering (TSO).
	Silent store-pair predictor[a]	64 entry silent store-pair predictor table, indexed by the store-conditional PC. support for up to 8 store-pair elisions at any time.
	Read-modify-write sequence predictor[b]	128 entry PC indexed predictor for collapsing read-modify-write sequences within critical sections into a single request.

[a] Used for SLE and TLR configurations in all experiments.

[b] Used for all experiments in Section 6.3.2 only.

and low-latency and the data network is a point-to-point direct network modeled after the Sun Gigaplane-XB. Figure 5-4 shows the SMP target system and Table 5-3 lists the system parameters. We evaluate SLE for this configuration.

Distributed shared-memory (DSM) configuration. The DSM configuration consists of two levels of caches and an SGI Origin 2000-type MESI protocol implemented among the L2 caches. The directory is full-mapped and stored with memory in DRAM. Silent evictions of clean cache blocks is supported. The SGI Origin 2000 uses two virtual channels for routing and relies on a complex high-level deadlock detector to resolve deadlock that may arise because of three-hop coherence transactions over two virtual channels, and resolves deadlocks by falling back on a slow

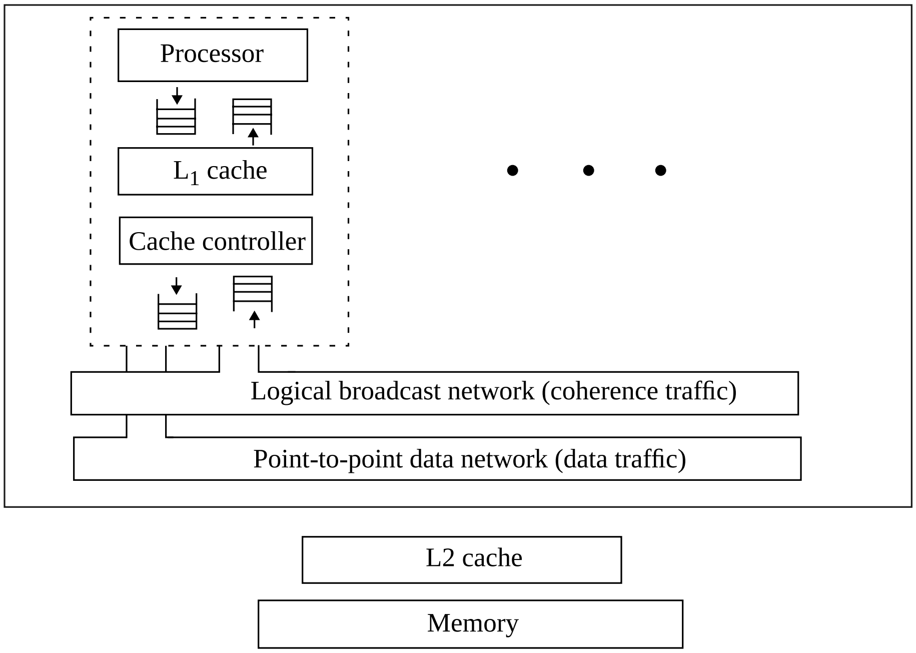

Processor
L_1 cache
Cache controller
Logical broadcast network (coherence traffic)
Point-to-point data network (data traffic)
L2 cache
Memory

Figure 5-3: Chip multiprocessor (CMP). *The L1 caches on the chip are kept coherent using a broadcast network as shown. Data transfer occurs through a high-bandwidth point-to-point network. The L2 cache and memory are off-chip as shown.*

Table 5-2: Memory system configuration: Chip multiprocessor

L1 caches	Data cache parameters	128KByte, 4-way associative, write-back, 1-cycle access, 16 pending misses. Block size: 64 bytes.
	Instruction cache parameters	64KByte, 2-way associative, 1-cycle access, 16 pending misses. Block size: 64 bytes.
	Protocol	Sun Gigaplane-type MOESI protocol between all L1s.
	Queue occupancy	1-cycle minimum in all queues in the system.
Network configuration	Snoop network	Split transaction. Address bus: broadcast network, snoop latency: 20 cycles, 120 outstanding transactions.
	Data network	Point-to-point, pipelined, transfer latency: 20 cycles.
L2 cache		4MByte, 6-way, 10-cycle access.
Page		4Kbyte size.

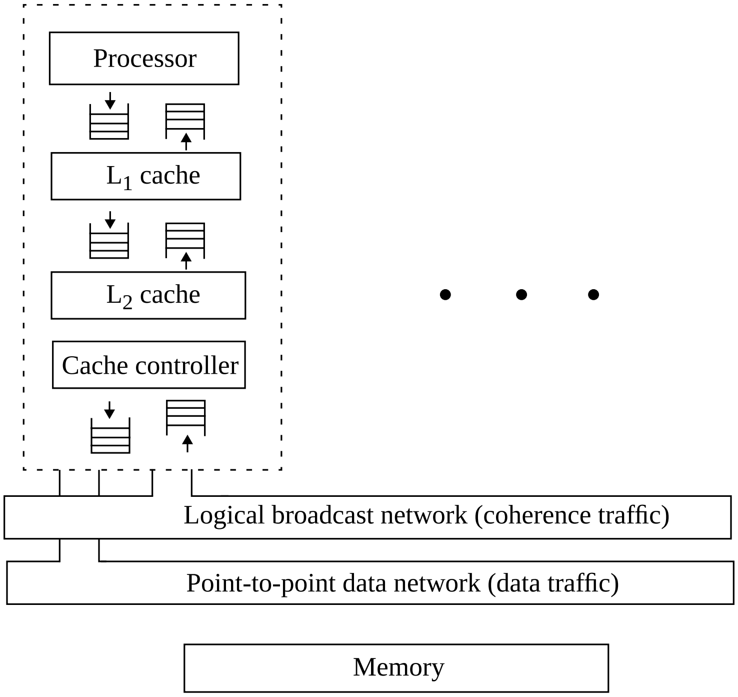

Processor
L1 cache
L2 cache
Cache controller
Logical broadcast network (coherence traffic)
Point-to-point data network (data traffic)
Memory

Figure 5-4: Symmetric multiprocessor (SMP). *The L2 caches are kept coherent using a broadcast network as shown. Data transfer occurs through a high-bandwidth point-to-point network. The queues between the L1 and L2 are also shown. Inclusion is maintained in the local hierarchy.*

Table 5-3: Memory system configuration: Symmetric multiprocessor

L1 caches	Data cache parameters	128KByte, 4-way associative, write-back, 1-cycle access, 16 pending misses. Block size: 64 bytes.
	Instruction cache parameters	64KByte, 2-way associative, 1-cycle access, 16 pending misses. Block size: 64 bytes.
	Protocol	MSI protocol.
	Queue occupancy	1 cycle minimum in all queues in the system.
L2 unified caches	Parameters	4MByte, 4-way, 12 cycle access, 16 pending misses.
	Protocol	Sun Gigaplane-type MOESI protocol between all L2s.
Network configuration	Snoop network	Split transaction. Address bus: broadcast network, snoop latency: 30 cycles, 120 outstanding transactions.
	Data network	Point-to-point, pipelined, 70 cycles transfer latency. Memory access: 70 cycles for 64 bytes.
Page		4KByte size.

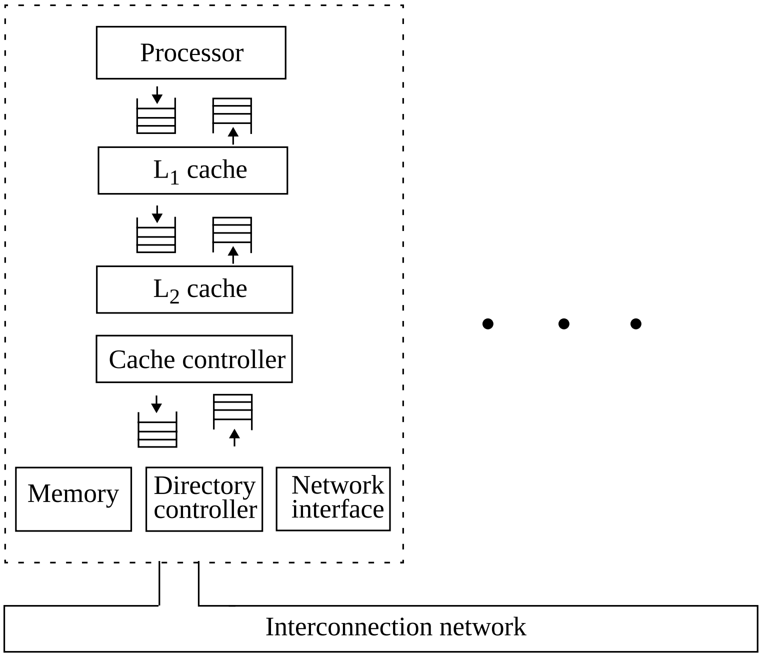


Figure 5-5: Distributed shared-memory system (DSM). *The directory and memory controller are integrated as shown. The interconnection network consists of three virtual channels for request, response, and interventions. The network is modeled as a point-to-point network.*

strict request-response sequence. To avoid the inefficiencies introduced by the use of only two virtual channels, our implementation employes three virtual channels. Figure 5-5 shows the distributed shared-memory multiprocessor target system and Table 5-4 lists the system parameters. We evaluate SLE for this configuration.

5.4 Benchmarks

Many microbenchmarks and benchmarks exist for evaluating synchronization performance. We select three microbenchmarks and seven benchmarks. The benchmarks are taken from the `SPLASH/SPLASH2` suites [150, 169].

Table 5-4: Memory system configuration: DSM multiprocessor

L1 caches	Data cache parameters	128KByte, 4-way associative, write-back, 1-cycle access, 16 pending misses. Block size of 64 bytes.
	Instruction cache parameters	64KByte, 2-way associative, 1-cycle access, 16 pending misses. Block size: 64 bytes.
	Protocol	MSI protocol
	Queue occupancy	1 cycle minimum in all queues in the system
L2 unified caches	Parameters	4MByte, 4-way, 12 cycle access, 16 pending misses.
	Coherence protocol	SGI Origin 2000 type MESI protocol, full mapped directory, 70 cycle access (overlapped with memory access).
Network configuration	Parameters	point-to-point network with three virtual channels
	Latencies	processor to local directory (70 ns), directory and remote route (50 ns). Some uncontended latencies: read miss to local memory: ~130 ns, read miss to remote memory: ~230 ns, read miss to remote dirty cache: ~360 ns
Page		4KByte size, round-robin allocation across nodes.

5.4.1 Microbenchmarks

We use three microbenchmarks—`multiple-counter`, `single-counter`, and `doubly-linked list`—specifically selected to evaluate three different behaviors of critical section data access representing points in the spectrum of data and lock contention. While these are not comprehensive, they are selected to provide insight into the behavior of SLE and TLR. The `multiple-counter` microbenchmark represents high, easily exploitable concurrency; the `single-counter` microbenchmark represents a case where no concurrency is exploitable and rate of data conflicts (and lock contention) is high. The `doubly-linked list` is a complex microbenchmark with difficult-to-exploit dynamic concurrency and high lock contention. The details of these microbenchmarks are described below.

1. `multiple-counter`. This microbenchmark represents an example of a coarse granularity lock and no data conflicts among critical sections. The `multiple-counter` microbenchmark consists of *n* counters protected by a single lock. Each processor uniquely updates only

one of *n* counters $2^{24}/n$ times. While a single lock protects the counters, there is no dependence across the various critical sections for the data itself and hence no conflicts.

2. `single-counter`. This microbenchmark represents an example of a fine granularity lock and high data conflicts. The `single-counter` microbenchmark corresponds to critical sections operating on a single cache block. One counter is protected by a lock and *n* processors increment the counter $2^{16}/n$ times. No inherent exploitable concurrency exists as all processors operate upon the same data (and cache block).

3. `doubly-linked list`. This microbenchmark represents an example of a fine granularity lock and a dynamically varying data conflict rate. The `doubly-linked list` microbenchmark consists of a doubly-linked list with `Head` and `Tail` pointers protected by one lock. Each processor dequeues an item by removing the item pointed to by `Head`, and then enqueues it by adding it to `Tail`. A processor that removes the last item sets both `Head` and `Tail` to `NULL`, and a processor that inserts an item into an empty list sets both `Head` and `Tail` to point to the new item. The benchmark finishes when $2^{16}/n$ enqueue and dequeue operations have completed. A non-empty queue can support concurrent enqueue and dequeue operations. When the queue is non-empty, each process modifies `Head` or `Tail`, but not both, so enqueuers can execute without interference from dequeuers, and vice versa. Processors must modify both pointers for an empty queue. This concurrency is difficult to exploit in any simple way using locks, since an enqueuer does not know if it must lock the tail pointer until after it has locked the head pointer, and vice-versa for dequeuers [66, 149]. The critical sections are non-trivial involving pointer manipulations and multiple cache block accesses. Figure 5-6 shows the C code for the `enqueue`() and `dequeue`() functions.

In the microbenchmarks, processors execute critical sections in a loop for a fixed number of iterations. Special care was taken in designing these microbenchmarks. We use a methodology similar to that used by Kumar et al. [89]. To ensure fairness, we introduce delay after a lock release operations. After releasing the lock, the processor waits a minimum random interval before proceeding to ensure another processor has an opportunity to acquire the lock before a successive local lock reacquire, thus reducing unfairness. The wait outside the critical section has to be larger than the inter-processor lock transfer time to ensure that the local processor will not succeed in reacquiring the lock.

```
void enqueue(entry *new)
{
  entry *tail;
  new->prev = NULL;
  new->next = NULL;

  LOCK (lock)
  tail = Tail;
  new->next = tail;
  if (tail == NULL)
    Head = new;
  else
    tail->prev = new;
  Tail = new;
  UNLOCK (lock)
}
```

```
entry *dequeue()
{
  entry *head;



  LOCK (lock)
  head = Head;
  if (head != NULL)
    {
      prev = head->prev;
      if (prev != NULL)
        prev->next = NULL;
      else
        Tail = NULL;
      Head = prev;
    }
  UNLOCK (lock)
  return head;
}
```

Figure 5-6: Doubly-linked list microbenchmark code. *The left side shows the enqueue function and the right side shows the dequeue function.*

5.4.2 Benchmarks

The benchmarks we use are `ocean-cont`, `cholesky`, `mp3d`, `barnes`, `radiosity`, `water-nsq`, and `raytrace`. `Barnes`, `cholesky`, and `mp3d` are drawn from the `SPLASH` [150] and `ocean-cont`, `radiosity`, `water-nsq` and `raytrace` are drawn from the `SPLASH2` suites [169]. All benchmark data structures are padded appropriately to eliminate false sharing. The benchmarks were run to completion. We use modified versions of `barnes` and `mp3d` from Alain Kägi’s experiments [80]. The modifications are described below. Table 5-5 lists the various benchmarks used, and their input sets. These specific benchmarks have been chosen because they represent noticeable synchronization delays and employ lock-based synchronization. These benchmarks have been optimized for sharing and thus have little communication in most cases. We are interested in determining the robustness and potential of our proposal even for these well tuned benchmarks. We briefly describe these benchmarks below. Detailed description of the benchmarks can be found elsewhere [150, 169].

Table 5-5: Benchmarks

Application	Suite	Type of simulation	Input data set	Nesting
Barnes[a]	`SPLASH`	N-Body	4K bodies	yes
Cholesky[b]	`SPLASH`	Matrix factorization	tk14.0/tk15.0	no
Mp3d[c]	`SPLASH`	Rarefied field flow	24,000 mols, 25 iter.	no
Radiosity	`SPLASH2`	3-D rendering	room	yes
Water-nsq	`SPLASH2`	Water molecules	512 mols, 3 iter.	no
Ocean-cont[d]	`SPLASH2`	Hydrodynamics	128x128, 2 days	no
Raytrace[e]	`SPLASH2`	Image rendering	car	no

[a] In this version of `barnes`, locks are stored directly in the cells instead of in a separate array as in the original version. The benefit of this new layout is primarily to remove unnecessary contention introduced by the fixed size lock array [80].

[b] tk14.0 input set used for experiments in Section 6.3.1 and tk15.0 input set used for experiments in Section 6.3.2.

[c] Locking version of `mp3d` used [80].

[d] Array storage is increased from 128 elements to 131 elements in each dimension to create arrays of prime size, thus reducing cache conflicts among elements in the arrays [93].

[e] Used only for experiments in Section 6.3.2.

`Barnes.` `Barnes` simulates the evolution of a system of bodies under the influence of gravitational forces. Every body is modeled as a point mass and exerts forces on all other bodies in the system. For each discrete time step (an iteration), `barnes` computes new positions of the bodies in the system. To avoid computing all $O(N^2)$ interactions among the bodies, `barnes` approximates the force exerted by a sufficiently distant cluster of bodies by the force resulting from the cluster's center of mass thus reducing the number of computed interactions to $O(N\mathit{log}N)$ or $O(N)$ depending on the distribution of bodies in the system. `Barnes` is based on a hierarchical octree representation of space in three dimensions. The root of this tree represents a space cell containing all bodies in the system. The tree is built by adding particles to the initially empty root cell and subdividing a cell into its eight children as soon as it contains more than a single body. Leaf nodes in the tree represent the actual bodies while the other nodes, the cells, represent a portion of the three-dimensional space holding the cells' children. A cell bisects the parent cell in all three dimensions. When computing the forces exerted by other bodies, `barnes` walks down the tree in breadth-first-search fashion and stops whenever it reaches a leaf or the considered node's center of mass is sufficiently far away, whichever comes first.

Nearly all of `barnes`' execution time is spent in two phases. The first phase loads bodies in the tree and the second phase computes the interactions. Each process is responsible for a fraction of the bodies in the system. In the tree-building phase, each process loads its bodies in the octree using locks to ensure atomic updates of the cell nodes. In the interaction computation phase, each process computes the forces exerted by other bodies for each body that they own. This phase does not require mutual exclusion because a process updates only the bodies it owns.

The original version of `barnes` stores the locks associated with the cells in a separate array. We instead store the locks in the cells directly. The benefits of this new layout is primarily to remove some unnecessary contention introduced by the fixed size lock array used in the original version. In the original version, since the number of elements in the array is less than the number of bodies in a typical simulation input, multiple bodies will map to the same lock in the array creating artificial contention. The new version does not suffer from artificial contention.

`Cholesky.` `Cholesky` performs Cholesky factorization of a sparse positive definite matrix. This program focuses on the most time-consuming components of factorization. The three steps in Cholesky factorization are: ordering, symbolic factorization, and numerical factorization. The program assumes an ordered input matrix. The second step accounts for a small fraction of the overall factorization runtime and thus is performed on a single process. The third step, numerical factorization, determines the actual numerical values of the non-zero entries in L (corresponding to the LU matrix). This is typically the most time-consuming phase and is parallelized in the program. Locks are used to protect task queues and matrix columns.

`Mp3d.` `Mp3d` is a Monte Carlo simulation of rarefied fluid flow simulating the hypersonic flow of particles at extremely low densities. `Mp3d` simulates the trajectories of particles through an active space and adjusts the velocities of the particles based on collisions with the boundaries (such as the wind tunnel walls) and other particles. After the system reaches steady state, statistical analysis of the trajectory data produces an estimated flow field for the studied configuration. The algorithm implemented in `mp3d` reduces the N^2 problem of finding collision partners to order N by representing the active space as an array of three-dimensional unit-sized cells. Only particles present in the same cell at the same time are eligible for collision consideration. If the application finds an eligible pair, it uses a probabilistic test to decide whether a collision actually occurs.

Work is allocated to each process through a static assignment of the simulated particles. Each simulated step consists of a move phase and a collision phase for each particle that the process owns. The move phase computes the particle's new position based both on its current position and velocity, and its interaction with boundaries. The collision phase determines if the particle just moved collides with another particle, and if so, adjusts the velocities of both particles. Data sharing occurs during collisions and through accesses to the unit-sized space cells. During a collision, a process may have to update the position and velocity of a particle owned by another process. Also, each space cell maintains a count of the particle population currently present in that cell. Therefore, each time a process moves a particle, it may have to update the population count of some space cells if that particle passes from one cell to another. These data accesses to particles and space cells may lead to race conditions that optional locks will eliminate at some performance cost. Locks associated with each space cell may be used to eliminate race conditions while accessing particles and space cells. Since processes update particle information owned by other processes only during a collision and a collision can only occur if two particles are present in the same cell, the space cell locks ensure mutual exclusion for both particle and space cell accesses. We study `mp3d` compiled with these locks.

`Radiosity.` `Radiosity` computes the equilibrium distribution of light in a scene using the iterative hierarchical diffuse radiosity method. A scene is initially modeled as a number of large input polygons. Light transport interactions are computed among these polygons, and polygons are hierarchically subdivided into patches as necessary to improve accuracy. The main data structures represent patches, interactions, interaction lists, the quadtree structures, and a BSP tree which facilitates efficient visibility computation between pairs of polygons. The structure of computation and the access patterns to the data structures are highly irregular. Parallelism is managed by distributed task queues, one per process, with task stealing for load balancing. Locks are used to protect access to task queues, interaction lists, and other shared structures.

`Water-nsq.` `Water-nsq` is an N-body molecular dynamics application for evaluating forces and potentials that occur over time in a system of water molecules. The forces and potentials are computing using an $O(N^2)$ algorithm, and a predictor-corrector method is used to integrate the motion of water molecules over time. For a user-specified number of time-steps, this program estimates the forces each molecule exerts on all others according to the Newtonian equations of motion. `Water-nsq` avoids computing all N^2 interactions by eliminating from consideration molecules outside of a sphere centered at the examined molecule and of a radius corresponding to half of the box length.

After initialization and one-time computations, each time-step consists of five phases: calculating the predicted values of atomic variables; computing intra-molecular forces for all atoms; computing the inter-molecular forces; calculating the corrected values of variables from the predicted values and computed forces; and computing the total kinetic energy of the system. The third task (computing the inter-molecular forces) accounts for the most of the execution time: its time complexity is $O(N^2)$ while all the other tasks have a time complexity of O(N).

`Water-nsq` exploits mostly the parallelism available within a phase; it exploits the inter-phase parallelism to a limited extent to avoid some synchronization between phases. To exploit locality, `water-nsq` both assigns statically to each process an even fraction of the molecules and stores the molecules assigned to the same process next to each other. Communication among processes occurs during the second (intra-molecular computation) and third (inter-molecular computation) phases. Communication in the second phase consists only of adding scalars into a global sum; locks ensure that the processes correctly update that sum. Communication also occurs in the inter-molecular computation, where processes read positions of the interacting molecules, compute the forces, and update the forces of both molecules. A lock per molecule ensures the atomicity of the force updates.

`Ocean-cont`. `Ocean-cont` simulates eddy and boundary currents in an ocean basin. The simulation is performed over multiple time-steps until the eddies and mean ocean flow attain a mutual balance. The work done every time-step involves setting up and solving a set of spatial partial differential equations. The continuous functions are transformed into discrete counterparts by second order finite-differencing and the resulting difference equations set up and solved on two-dimensional fixed-size grids representing horizontal cross-sections of the ocean basin. The grids are represented conceptually as 4-D arrays with all subgrids allocated contiguously and locally in the nodes that own them. A red-black Gauss-Seidel multigrid equation solver is used. The memory access behavior of `ocean-cont` is regular and input independent. Grid tasks are permanently assigned to processes and each task performs the computational steps on the section of the grids that it owns, regularly communicating with other processes. Communication among processes involves barrier synchronization to preserve dependences between certain computations, near-neighbor communication while computing the Jacobians and Laplacians, and updates of a counter by all processes for every SOR iteration to determine convergence.

The program uses two locks. The first lock ensures that each process updates a global sum correctly in order to compute a matrix integral. The second lock helps determine when the SOR iterations have converged. In both cases the algorithm uses a simple lock rather than a tree of locks to perform the reduction.

In our experiments, the array storage is increased slightly (from 128 elements to 131 elements in each dimension) to create arrays of prime size, thus reducing cache conflicts among elements in the arrays [93].

`Raytrace`. `Raytrace` renders a three-dimensional scene using ray tracing. A hierarchical uniform grid is used to represent the scene. A ray is traced through each pixel in the image plane and reflects in unpredictable ways off the objects it strikes. Each contact generates multiple rays, and the recursion results in a ray tree per pixel. The image plane is partitioned among processes in contiguous blocks of pixel groups, and distributed tasks queues are used with task stealing. The memory access patterns of the application are highly unpredictable. The program provides efficient access to the scene description through a hierarchical uniform grid to traverse the read-only scene data quickly, round-robin distribution of the scene data across the nodes in the system to balance load on the network and the memory modules, and replication of data in the caches.

The program uses locks to protect access to a counter and to ensure correct operation on distributed task queues. The counter is used to assign a unique identifier to each newly spawned ray and the critical section to access it consists only of fetching the counter, adding one to it, and storing it back to memory. Contention to that lock is very high. A set of locks protect the task queues (one queue per process and one lock per queue). Contention to these locks is typically fairly low, unless the number of participating processes approaches the number of rays created.

5.4.3 Synchronization primitives

In this section, we discuss the two synchronization primitives we use in our experiments. The test&test&set locks, discussed below in Section 5.4.3.1 form our base synchronization primitive and is also used for SLE and TLR. We compare TLR to MCS software queued locks and we discuss MCS locks below in Section 5.4.3.2.

5.4.3.1 Test&test&set locks

Test&test&set [145] is an extension of test&set [7]. Test&test&set performs a read of the lock before attempting a test&set operation. Waiting requesters spin on shared, read-only copies of the lock and wait for the holder to release the lock. When the lock holder issues the release, the read-only copies are invalidated. The holder, having obtained a writable copy of the lock, releases it. Subsequently, all waiting requesters issue a request to load a read-only copy of the lock, and finding it released, all attempt a test&set. Only one of the requestors succeeds in the test&set.

The contention when the lock is freed can be substantial because all requesters attempt to acquire the lock at that point and then all attempt to upgrade the lock to a writable state.

5.4.3.2 MCS locks

The MCS scheme [120, 121] inserts requesters for a held lock into a software queue at the time of the request. Atomic operations such as swap and compare&swap are used to update the list correctly. With queue-based locking, arbitration for the eventual recipient of the lock is therefore performed in advance, first-come, first-served.

Maintaining the requester queue in software results in large overhead, especially under contentionless conditions. When a lock is released, however, communication occurs only between the releaser and the requester at the head of the queue. Network traffic is thus reduced to a constant number of network traversals per synchronization access. In addition, each processor waiting for the lock spins locally on distinct memory addresses (instead of a single address as with test&test&set), which further reduces the load on the network. Each processor in the queue maintains a pointer to the address on which the next processor in the queue spins. When the current lock holder leaves the critical section, it clears the value pointed to by the address that it maintains.

Chapter 6

Performance Evaluation

Speculative Lock Elision (SLE) and Transactional Lock Removal (TLR) provide improved programmability and stability of multithreaded programs. In this chapter, we study the impact of both SLE and TLR on the execution time of programs.

We begin the chapter by qualitatively understanding the sources of performance in Section 6.1. We consider different critical section behaviors, namely varying data conflict and lock contention, and provide intuition behind why one would expect improved performance. We also discuss conditions under which performance may be degraded. In Section 6.2 we use microbenchmarks to quantitatively study SLE and TLR and in Section 6.3 we evaluate SLE and TLR using benchmarks chosen from the `SPLASH` and `SPLASH2` suite. The experimental methodology, system configuration, microbenchmarks, and benchmarks were discussed in Chapter 5. In the discussion below, we refer to the base system without SLE or TLR as BASE. The base system and SLE together form BASE+SLE, or SLE for short; and the base system and SLE and TLR together form BASE+SLE+TLR, or TLR for short. We also compare TLR to MCS locks in Section 6.3.2.

6.1 Qualitatively understanding performance

Lock contention occurs when a thread attempts to acquire a lock owned by some other thread. Data conflicts occur when multiple threads access protected data when executing in a lock-free mode and at least one thread is writing the protected data. Lock contention prevents data conflicts from being exposed. Lock contention determines performance of the base system without SLE or TLR. Data conflicts, on the other hand, determine performance in a system with SLE or TLR.

Four cases for lock contention and data conflicts are:

1. No lock contention and no data conflicts

2. No lock contention and data conflicts
3. Lock contention and no data conflicts
4. Lock contention and data conflicts

Since lock contention masks data conflicts, we do not consider case 2 above. This case occurs when a protected data structure is either accessed from outside a critical section, or the data structure is protected by different locks. These cases are examples of data races in a program. Since the intent of locks is to prevent data races, we do not consider programs that have data races for discussion in this section. Both SLE and TLR however maintain the semantics of the program independent of whether a data race exists and always provide a correct execution in the presence of data races. Handling data races under TLR correctly was discussed in Chapter 4.

6.1.1 No lock contention

In this case, multiple threads, though executing concurrently, do not request the same lock. Thus, if the lock is not held by another thread, the data protected by the lock is not being accessed by another thread. SLE and TLR behave identically because of the absence of any data conflicts. Performance benefits may accrue because of the following reasons:

1. *Reduced observed remote memory latencies*. Since the lock variable is kept in shared state locally, all accesses to the lock variable result in a cache hit and do not experience a long latency miss. The benefit arises if the reorder buffer of the processor is unable to completely tolerate a remote miss to another processors cache. The benefit is reduced for local lock acquires where the lock being acquired is already cached locally in an exclusive state.
2. *Reduced memory traffic.* Lock acquire and release operations often result in memory system traffic, even in the absence of contention. By eliding lock operations when possible, lock induced memory traffic in the form of upgrades, data transfers, and read-for-exclusive-ownership requests in the memory system is eliminated. The benefit is not present for local lock acquires where the lock being acquired is already cached locally in an exclusive state.

Since the lock is not contended, SLE and TLR do not achieve any concurrent execution benefits over BASE.

6.1.2 Lock contention and no data conflicts

In this case, lock contention is present but the various threads access non-conflicting data sets. This commonly occurs when coarse-grain locks are used or due to conditional control flows within the critical section. In the absence of data conflicts, SLE and TLR both behave identically because TLR is never invoked in the absence of data conflicts. This case provides maximum benefits for SLE and TLR over BASE because the locks are truly unnecessary for correctness of the dynamic execution of the program. In addition to reduced memory latencies, and reduced memory traffic, this case also benefits from concurrent critical section execution and completion—BASE unnecessarily serializes execution of concurrent threads. Reasons for performance benefits due to SLE (and TLR) include:

1. *Concurrent critical sections.* This occurs as a result of coarse-grain locking but sometimes may also occur due to varying control-flow within a critical section resulting in non-conflicting data sets being accessed by the multiple threads.
2. *Reduced observed remote memory latencies.* Since the lock variable is kept in shared state locally, all accesses to the lock variable result in a cache hit and do not experience a long latency miss. The benefit arises if the reorder buffer of the processor is unable to completely tolerate a remote miss to another processors cache. The benefit is reduced for local lock acquires where the lock being acquired is already cached locally in an exclusive state.
3. *Reduced memory traffic.* Lock acquire and release operations often result in memory system traffic, even in the absence of contention. By eliding lock operations when possible, lock induced memory traffic in the form of upgrades, data transfers, and read-for-exclusive-ownership requests in the memory system is eliminated. The benefit is not present for local lock acquires where the lock being acquired is already cached locally in an exclusive state.

6.1.3 Lock contention and data conflicts

The final case we consider is when lock contention occurs and the threads access common data sets in a conflicting manner and occurs mostly with fine-grain lock use. With data conflicts, TLR behaves differently from SLE because while SLE may need to fall back on the BASE mechanisms, TLR provides explicit support for achieving a successful lock-free execution even in the presence of data conflicts. BASE behavior is still limited by lock contention[1] and behaves similar to the earlier case in Section 6.1.2 of lock contention and no data conflicts. Since SLE and TLR behave differently, we discuss the two separately.

6.1.3.1 SLE

In the presence of conflicts, SLE forces a restart due to misspeculation. The restart process itself is not expected to result in a performance loss because the thread would otherwise have merely spun waiting on a location (often the lock location for test&test&set-based locks). However, the restart results in memory requests being reissued to the memory system and may result in increased coherence protocol interference thus degrading performance. This observation is not new and even holds true for BASE where if, under lock contention, multiple threads issue requests into the critical section speculatively, the interference in the memory system increases and unnecessary latencies are added to the critical path of access to the data block by the thread owning the lock. This is also referred to as wasted parallelism where while the multiple threads appear to be doing work, they are interfering with the thread holding the lock because the data block is repeatedly stolen away from the lock owner by competing threads.

Under SLE, if all threads restart, the additional requests and memory traffic may result in performance loss for SLE over the BASE. However, if not all threads restart, then performance may be gained.

1. *Performance loss if all threads restart.* In the presence of data conflicts, while traffic due to the lock variable may be reduced, additional latencies introduced due to coherence protocol interference may degrade performance if all threads restart. This is because when multiple processors compete for a cache block simultaneously, the coherence permissions for the block transfer and the block moves around the system from processor to processor. If the processor restarts, the additional latency due to the movement of the cache block and the traffic introduced may degrade performance.
2. *Performance gains possible if at least one thread succeeds in elision.* Often, threads may be separated sufficiently apart in time such that one thread succeeds even in the presence of data

[1] In the presence of lock contention, the conflicting threads rather than interfere with the thread owning the lock, wait for the lock to be released by spinning locally on a location.

conflicts. This may happen because even though for two threads conflicting on data, only one thread observes the conflict and restarts and the other thread proceeds and completes its critical section without restarting. The benefits also occur because on a restart, a smaller number of threads compete for the data, and the traffic due to lock operations is reduced.

6.1.3.2 TLR

In the presence of data conflicts, TLR presents benefits over SLE. TLR uses an explicit concurrency control mechanism using timestamps for fairness, and request deferrals to provide a serializable lock-free execution in the presence of data conflicts and for reducing the negative impact of coherence protocol interference. The coherence protocol is used to construct a chain of conflicting processors and thus enabling coordinated and efficient data transfer. TLR, as described in this thesis, does not change the coherence protocol state transitions but coordinates data transfers using the coherence protocol and timestamps. The order in which processors execute *conflicting* critical sections is determined by timestamps—called *timestamp-order.* The order in which data blocks move around the system is determined by the coherence protocol order—the order in which requests were received by the coherence protocol—called *coherence-order.* This was studied in detail in Chapter 4. TLR allows non-conflicting threads to complete in parallel and without any serialization while conflicting threads are ordered based on timestamps.

Performance gains occur when either the timestamp order is similar to the coherence order or when the cost of misspeculation (due to a mismatch between timestamp-order and coherence-order) is lesser than the cost of actually acquiring and releasing locks.

Reasons for performance benefit/loss include:

1. *Reduced observed remote memory latencies.* Since the lock variable is kept in shared state locally, all accesses to the lock variable result in a cache hit and do not experience a long latency miss. The benefit arises if the reorder buffer of the processor is unable to completely tolerate a remote miss to another processors cache. The benefit is reduced for local lock acquires where the lock being acquired is already cached locally in an exclusive state.

2. *Reduced memory traffic.* Lock acquire and release operations often result in memory system traffic, even in the absence of contention. By eliding lock operations when possible, lock induced memory traffic in the form of upgrades, data transfers, and read-for-exclusive-owner-

ship requests in the memory system is eliminated. The benefit is not present for local lock acquires where the lock being acquired is already cached locally in an exclusive state.

3. *No locking overhead in presence of data conflicts*. With TLR, no locks are requested even in the presence of data conflicts (SLE by itself would not provide much benefit in the presence of data conflicts). Thus, memory traffic and observed memory latencies are reduced.

4. *Coordinated data transfer.* Since data is requested directly and the coherence protocol is used to construct chains for fast data transfer, the latency is optimized. Additionally, hardware is used to coordinate the transfer thus minimizing latency.

5. *Coherence-order/timestamp-order mismatch.* Since the coherence protocol is unchanged, the order of the processor requests in the chain may be different from the order of timestamps (and thus priority) of the processors in the chains. A performance degradation may occur if the two chains are out of order often enough that the delays in transferring data hurt performance. This is the primary determiner for performance under TLR. As we will see, a sub-optimal ordering (due to a mismatch between coherence-order and timestamp-order) results in sub-optimal performance while an optimal ordering gives optimal performance.

Resource constraints and performance degradation. When a resource constraint is encountered and speculation can not continue, the elided store is allowed to be exposed to the memory system, thus essentially performing a lock acquire operation without actually requiring a restart. The speculative work is not wasted and is committed if the write successfully completes.

Cost of maintaining fairness. TLR enforces fairness by providing starvation freedom. If the benchmark would benefit from unfairness, then TLR would perform worse than BASE. An example is if a processor executes a critical section multiple times in succession before allowing any other processor to execute the critical section and this results in better performance than if all processors took turns in a fair manner (for example, in a first-come first-served manner). In this case, TLR would pay the price for enforcing fairness. We discuss this more later in the chapter.

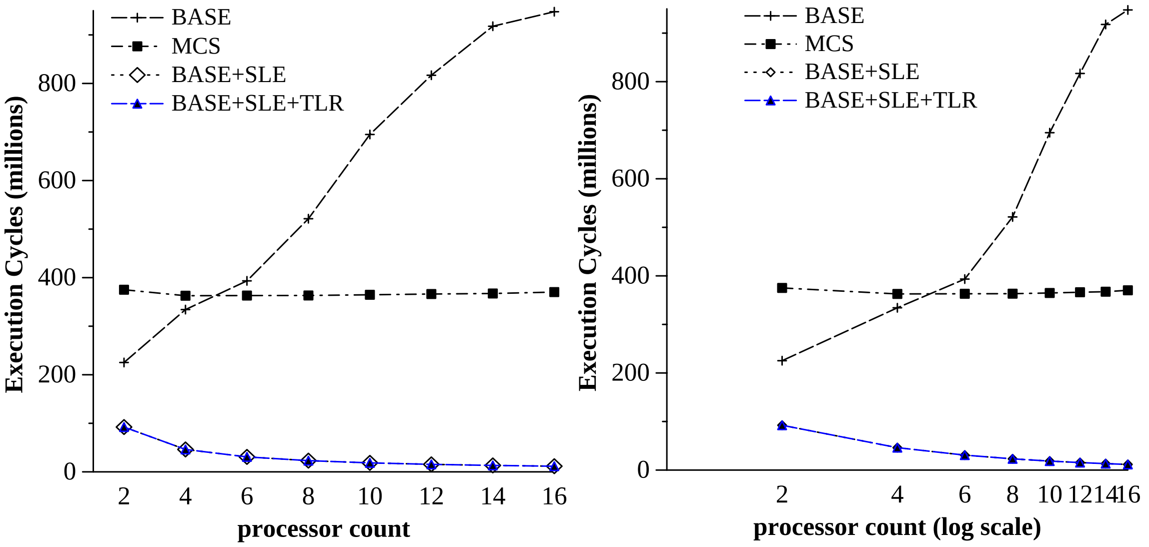


Figure 6-1: Multiple-counter microbenchmark results. *The left graph has a linear x-axis and the right graph has a log scale x-axis. The y-axis represents parallel cycle time in millions. The benchmark performs 2^{24} /n increments of a unique local counter for an n-processor system. As expected, SLE and TLR suffer from no locking overhead and as shown by the right graph, represent perfect scalability with increasing processor counts The SLE and TLR plots cannot be distinguished because they perform identically.*

6.2 Microbenchmark evaluation

Figure 6-1 shows results for `multiple-counters`. The BASE scheme degrades performance as more threads are run concurrently because of severe contention for the lock while using test&test&set locks. MCS, as expected is scalable under high contention and experiences a fixed software overhead independent of the number of threads competing for the lock. SLE (BASE+SLE) and TLR (BASE+SLE+TLR) behave identically because there are no data conflicts. Both schemes outperform BASE and MCS because in the absence of any data conflicts, both SLE and TLR experience no locking overhead or serialization, and true concurrency in the application is exploited. As is seen by the log-scale graph on the right side in Figure 6-1, perfect scalability is achieved using SLE and TLR.

Figure 6-2 shows results for `single-counter`. As is the case with the `multiple-counter`, BASE performance degrades with increasing threads because of severe conten-

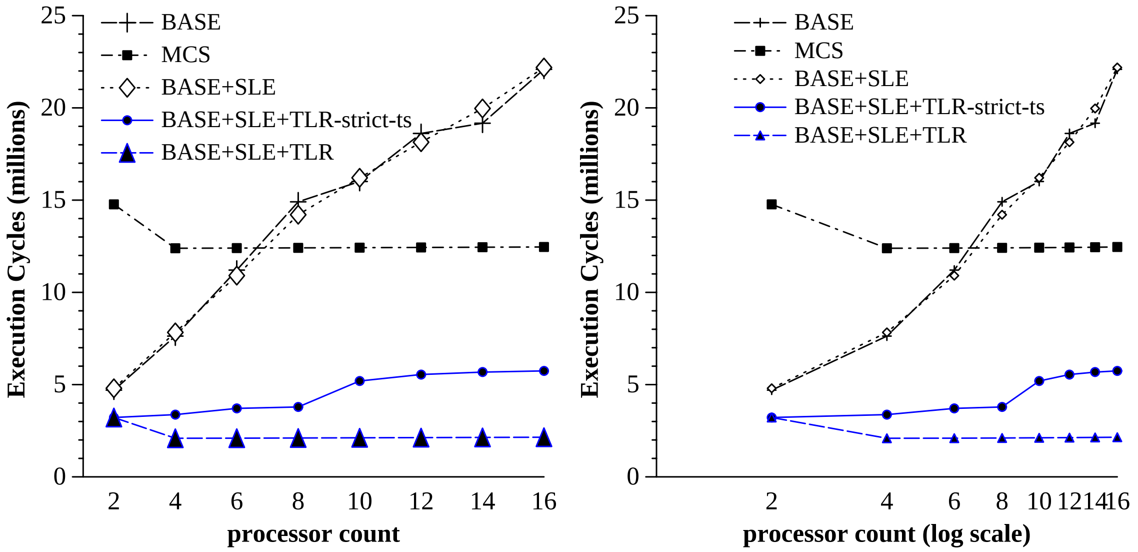


Figure 6-2: Single-counter microbenchmark results. *The left graph has a linear x-axis whereas the right graph has a log scale x-axis. The y-axis represents parallel cycle time in millions. The benchmark performs $2^{16}/n$ increments of a shared counter for an n-processor system. BASE and SLE perform similar because SLE experiences frequent data conflicts and falls back on the BASE scheme. Two schemes for TLR are shown. TLR-strict-ts corresponds to the case where timestamp order is enforced even if deadlock dangers did not exist and fairness was not compromised (see Section 4.4.5). Performance gap between TLR and TLR-strict-ts exists because the mismatch between timestamp order and coherence order results in a sub-optimal performance. MCS achieves scalable performance but experiences a fixed software overhead.*

tion for the lock. SLE behaves similar to BASE because SLE detects frequent data conflicts, turns off speculation, and falls back to the BASE scheme. MCS again is scalable but experiences a fixed software overhead. We show two cases for TLR: TLR and TLR-*strict-ts*. Under TLR, as discussed in Section 4.4.5, timestamp order can be selectively relaxed if there is no danger of deadlock—a case that occurs when only one cache block is contended for. TLR-strict-ts corresponds to the implementation where timestamp order is always enforced, independent of whether deadlock could occur or not. As can be seen, both TLR and TLR-strict-ts outperform BASE, SLE, and MCS. MCS performs a constant factor worse than TLR because MCS has the additional software overhead of lock acquisitions and queue maintenance while TLR uses the existing coherence protocol to construct an ordered queue in hardware. The performance gap between TLR and TLR-strict-ts exists because sometimes the order in which requests reach the coherence point (coherence-order) is different from the order of the respective timestamps (timestamp order) resulting in misspeculation if the timestamp order is strictly enforced. This mismatch of protocol order and timestamp order results in a sub-optimal[2] ordering and additional latencies (see Section 4.4.4 for a detailed discussion).

When timestamps can be selectively relaxed, as is the case for TLR in Figure 6-2, ideal TLR behavior can be achieved. The average number of cycles between any two releases in the system is on the order of 30 cycles (broadcast latencies 20 cycles and data transfer latency is 20 cycles). This essentially is as close as one can get to the ideal behavior for a single cache block benchmark because in this case, the timestamp order can be considered identical to coherence order. The TLR execution suffers no misspeculation and no processor ever restarts in that execution.

The considerable performance gap between TLR and TLR-strict-ts suggests future work to minimize the effects of coherence-order and timestamp-order mismatch.

Figure 6-3 shows results for `doubly-linked list`. Performance of BASE degrades similar to the other microbenchmarks because of severe lock contention. SLE does not perform well either (and performs similar to BASE) because determining when to apply speculation is difficult due to the dynamic concurrency of the benchmark. More often than not, SLE falls back to the base case of lock acquisitions using BASE because of detected data conflicts. Any concurrency SLE exploits is offset by locking overhead when SLE needs to acquire the lock. MCS is scalable but experiences a fixed software overhead. TLR performs well and can exploit enqueue/dequeue concurrency. The `doubly-linked list` microbenchmark consists of two critical sections protected by the same lock and the data accesses within these critical sections is in reverse order. Thus, in a sense, this microbenchmark represents an extreme case for TLR. In this microbenchmark, multiple cache blocks are contended for (the `head` pointer, `tail` pointer, and the data element) and multiple chains are formed for these blocks. Two main effects occur for TLR: on one hand, performance gains occur due to exploited concurrency, and on the other hand, sub-optimal performance is achieved due to a mismatch of timestamp-order and coherence-order. Since these two effects do not occur evenly, the plot is not flat as for the other microbenchmarks. Nevertheless, TLR still outperforms the BASE, SLE, and MCS schemes.

[2] We use the term sub-optimal to imply the performance is not as good as it could be. The performance may still be better than BASE or MCS and thus using a term such as *performance degradation* would be inaccurate and misleading.

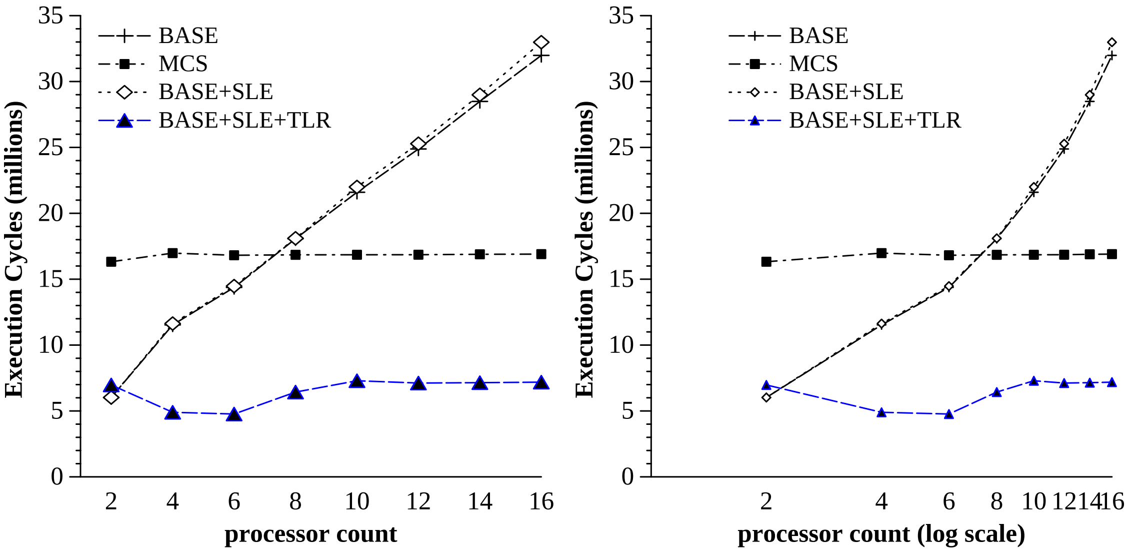


Figure 6-3: Doubly-linked list microbenchmark results. *. The left graph has a linear x-axis whereas the right graph has a log scale x-axis. The y-axis represents parallel cycle time in millions. The benchmark performs 2^{16}/n enqueue/dequeue pairs in an n processor system. SLE experiences frequent conflicts and turns off speculation thus behaving similar to the BASE scheme. MCS performs with a fixed software overhead and TLR outperforms both BASE and MCS.*

In summary, TLR outperforms both BASE and MCS for the microbenchmarks we use. TLR exploits dynamic concurrency while both BASE and MCS are limited by synchronization performance. MCS performs a constant factor worse than TLR while BASE performance degrades quite substantially with increasing contention. Poor behavior of BASE under lock contention occurs because of repeated access to the lock variable by multiple processors racing for the lock and data thus introducing a large amount of traffic into the network. MCS is scalable because processors form an orderly queue of lock requestors in software rather than repeatedly compete for the lock variable and data.

We now briefly look at the effect of unfairness of the primitive on performance. As discussed in Chapter 5, we have designed our microbenchmarks to ensure fairness. A random and minimum delay is added after a lock release so that a remote processor can succeed in acquiring the lock before the previous lock holder reacquires the lock. In an experiment, we set this delay to 0—in other words, the processor releasing the lock does not wait before reattempting to acquire the lock. TLR is fair but does not strictly enforce first-come first-served order. In other words, the use of

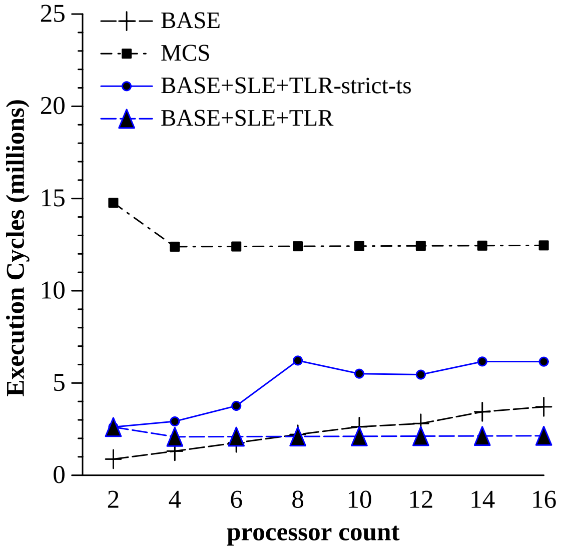


Figure 6-4: Impact of unfairness on microbenchmark performance. *We use the single-counter microbenchmark. MCS enforces strict first-come first-served ordering and thus pays a fixed overhead. TLR and TLR-strict-ts provide a sense of fairness—while it is not first-come first-served, it is based on timestamp resolution for conflicting requests. Thus, all processors keep up with each other over time even though the ordering is not strictly first-come first-served. BASE greatly benefits from the unfairness in the benchmark because a processor can acquire the lock multiple times (in 10s) before another processor is able to intervene and acquire the lock.*

timestamps ensures all processors get to execute their critical sections within a bound and keep up with each other. The results are shown for the single counter example in Figure 6-4. BASE performs better than TLR because in BASE, a processor performs a series of successive local lock acquires and releases before another processor can acquire the lock. This is not the intent of the microbenchmark since the microbenchmark is intended to study the performance of synchronizing multiple processors when multiple processors compete for the lock; not the synchronization performance for a single processor.

6.3 Benchmark performance

We now study the performance of SLE and TLR using some benchmarks chosen from the `SPLASH` and `SPLASH2` suites. The system configuration was discussed in Section 5.3 and the benchmarks and their input sets were discussed in Section 5.4. We first evaluate SLE performance and then separately evaluate TLR.

6.3.1 SLE performance

For SLE, we chose a restart threshold of 1. This means, on a conflict-induced misspeculation, execution was restarted and SLE retried once more. If a subsequent conflict occurred, the lock was acquired. Support for up to 8 silent store-pair elisions is assumed (Section 3.10) implying that up to 8 properly nested locks can be elided. The experiments in this section did not suffer from any resource induced misspeculation. In other words, the critical section data fit in the local cache hierarchy.

SLE is evaluated for three different system configurations: CMP (chip multiprocessor), SMP (symmetric multiprocessor), and DSM (distributed shared-memory multiprocessor). We present results for two thread configurations for each: 8 threads and 16 threads. The SMP configuration has larger latencies than the CMP configuration and the DSM configuration uses a different coherence protocol.

In all figures in this section, the y-axis is normalized parallel execution time—cycles taken to execute the parallel portion of the benchmark. The first bar of each pair corresponds to the BASE case and the second bar of each pair corresponds to SLE (BASE+SLE). Each bar is divided into two parts: contributions due to lock variable accesses (loads and stores of the test, test&set, and release) and the remaining contributions. The lock portion only includes memory references to lock variables and does not necessarily include the total time spent in the synchronization algorithm itself (e.g., the branch instructions in the test&test&set algorithm are not counted in the lock portion).

The stall accounting for the bars is performed at instruction commit—the instruction that stalls commit is charged the stall. The breakup is approximate since accounting for stall cycles due to individual operations is difficult and often inaccurate in out-of-order processors. In addition, a lock acquire operation involves an atomic read-modify-write instruction. This instruction cannot retire from the reorder buffer until it has been ordered by the memory system. In other words, such an operation acts like a memory fence[3] and remains at the head of the reorder buffer until any writes in the write buffer prior to the lock acquire are also drained and exposed to the memory system. Thus, the lock portion also accounts for the time it takes to flush the write buffer while a lock

[3] This behavior will occur independent of the memory consistency model implemented (including in release consistent systems). Any load operations past the lock acquire can still freely be issued because we support an aggressive implementation of total store ordering [45].

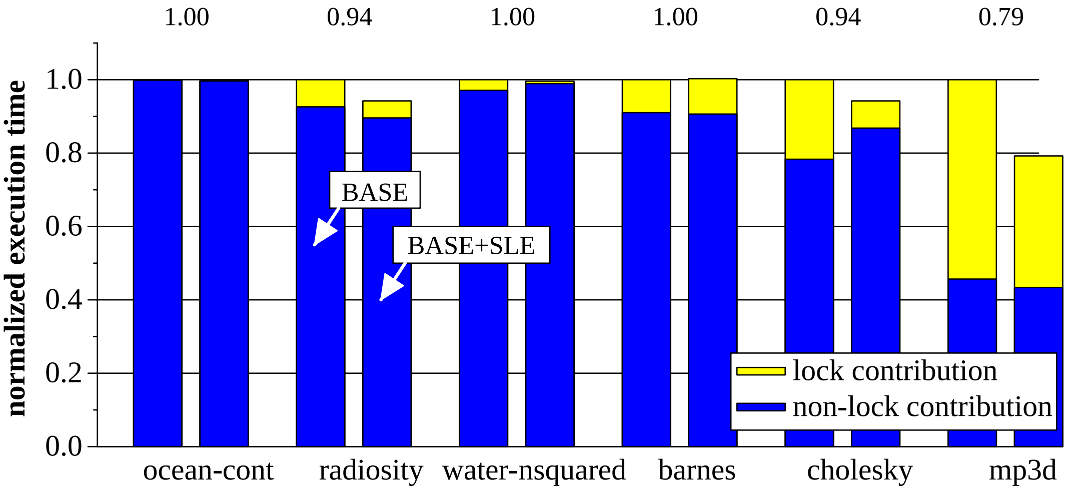


Figure 6-5: SLE performance for an 8-way CMP. *The y-axis is normalized parallel execution time.The first bar of each pair corresponds to the base case. The second bar of each pair corresponds to the SLE case. The fractions on top of the bar-pairs are normalized execution times for the SLE case. All normalizations are with respect to the base case.*

acquire is pending stalled at the head of the reorder buffer. For some benchmarks, the non-lock portion for the optimized case is larger than the non-lock portion for the base case. This is because sometimes removing locks puts other memory operations on the critical path. Speculative loads issued for data within critical sections that were earlier overlapped with the lock acquire operation are now exposed and stall the processor. A normalized execution time implies a lower bar is better.

Figure 6-5 and Figure 6-6 show performance of SLE for a 8- and 16-way chip multiprocessor configuration. As expected, the locking overhead is higher for a 16-way configuration than for a 8-way configuration.

While `ocean-cont` has contention, the contribution of synchronization operations to performance loss is small and thus the performance improvement is slight.

`Water-nsq` has low contention. The bars for `water-nsq` indicate performance can be improved by eliding lock operations (about 3% for 8 threads and 10% for 16 threads). However, once SLE is applied, the net performance gains remain small. This is a result of the inaccuracy of our stall accounting methodology. We use the retire stage to determine stalls. The memory operations within the critical section, that were earlier overlapped with the lock latencies, get exposed

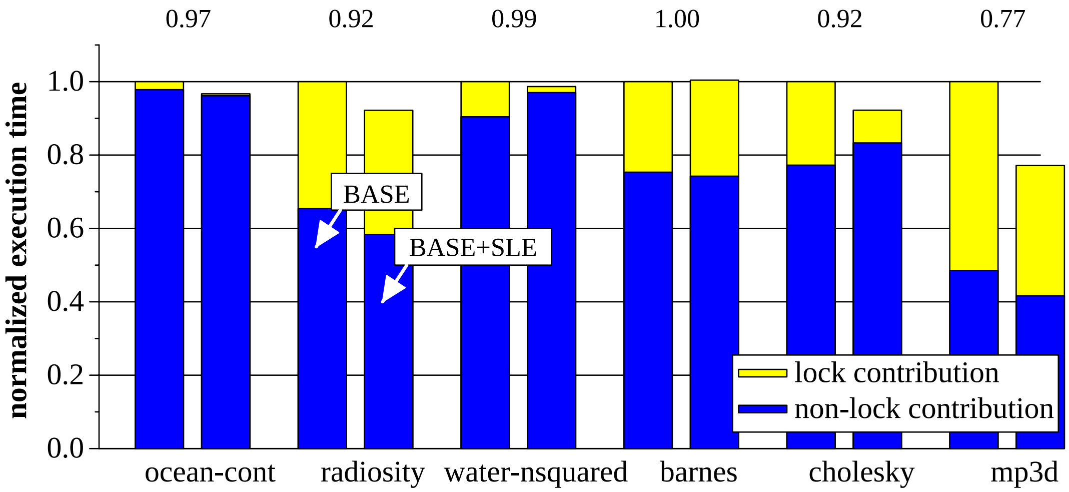


Figure 6-6: SLE performance for a 16-way CMP. *The y-axis is normalized parallel execution time. The first bar of each pair corresponds to the base case. The second bar of each pair corresponds to the SLE case. The fractions on top of the bar-pairs are normalized execution times for the SLE case. All normalizations are with respect to the base case.*

and account for the stalls. These operations result in cache misses and stall the processor core. The lock latencies earlier were merely hiding other latencies that were also on the critical path.

`Radiosity`, `barnes`, and `cholesky` (tk14.O input was used for the experiments in this section) have lock contention and even with SLE, a substantial lock contribution remains. This is because these benchmarks also have true sharing within critical sections resulting in data conflicts. Data conflicts under SLE result in a misspeculation and after a certain number of restarts, the execution falls back on the lock-based mechanism. With SLE, `Barnes` experiences a slight performance loss (< 1%) for a 16 thread configuration.

`Mp3d` has largely uncontended locks. While SLE helps `mp3d` quite noticeably, a large portion of the lock contribution still remains. `Mp3d` performs frequent lock operations by locking a cell and operating on it. More than a million lock acquires are performed for the run and the locks are largely migratory in nature. Thus, often a lock when requested is present in a remote cache. By eliminating lock acquire and release operations, substantial memory traffic is removed (in the form of upgrades and read requests to remote caches). Every cell has a lock and thus the cache footprint is large. Two reasons for the remaining lock contributions are: 1) the 128Kbyte data cache cannot hold all locks in shared state and these locks are frequently evicted. Thus, the processor core experiences long latency misses to memory for locks that, with a larger cache, otherwise would have been only a local cache hierarchy hit. 2) mp3d performs frequent writes and thus the lock acquire stalls the head of the reorder buffer while the write buffer gets flushed. While loads still issue past the pending lock acquire, the reorder buffer cannot completely hide the latency and load-data dependent instructions may follow.

6.3.1.1 Varying system configurations

We now evaluate SLE performance with two different system configurations—an SMP configuration and a DSM configuration.

Figure 6-7 and Figure 6-8 present performance for an SMP configuration. The SMP configuration uses the same processor model as the CMP configuration and the same coherence protocol. However, the SMP configuration has a large level-two cache and longer coherence and data latencies. The performance trends are similar to the CMP configuration. The gains are slightly higher for the SMP version because the increased latencies in the system result in lock operations contributing to a larger portion of execution time. The performance to be gained in the event of a successful SLE execution is higher.

The benchmark where the trend with the SMP configuration is markedly different from the CMP configuration is mp3d. This is because now a large 4MB level two cache backs the level one caches and thus most locks that are elided remain locally cached in a shared state over the execution of the program and the out-of-order processor core does not experience long latency misses to memory or remote caches.

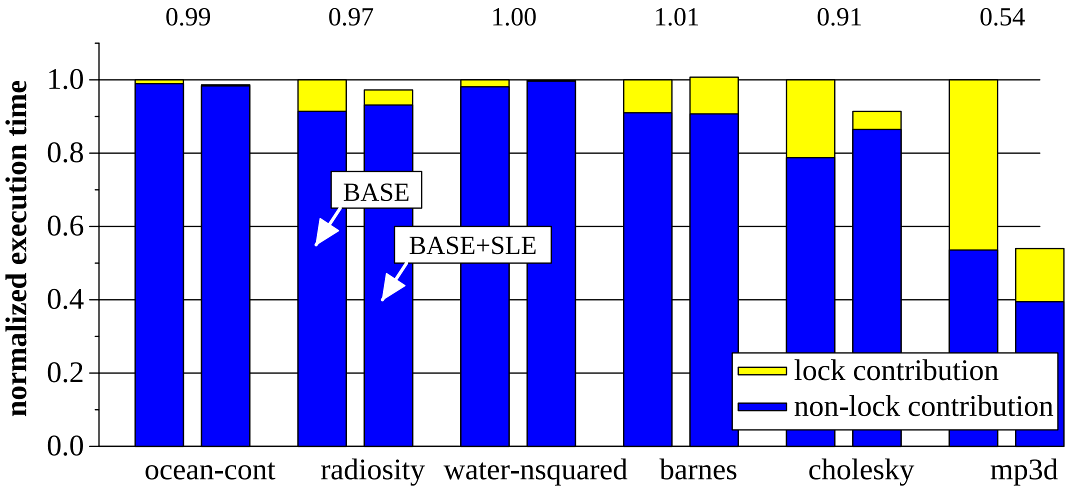


Figure 6-7: SLE performance for an 8-way SMP. *The y-axis is normalized parallel execution time. The first bar of each pair corresponds to the base case. The second bar of each pair corresponds to the SLE case.The fractions on top of the bar-pairs are normalized execution times for the SLE case. All normalizations are with respect to the base case.*

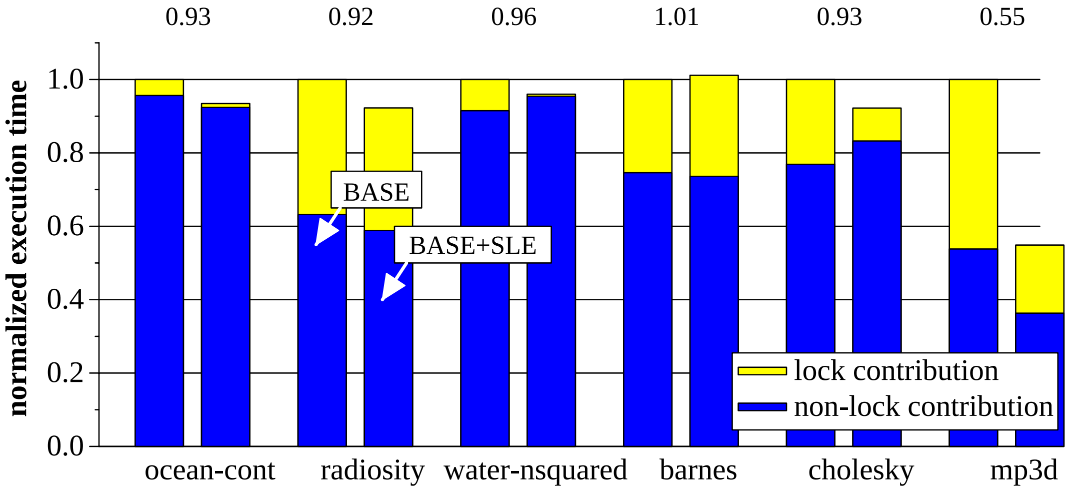


Figure 6-8: SLE performance for an 16-way SMP. *The y-axis is normalized parallel execution time. The first bar of each pair corresponds to the base case. The second bar of each pair corresponds to the SLE case. The fractions on top of the bar-pairs are normalized execution times for the SLE case. All normalizations are with respect to the base case.*

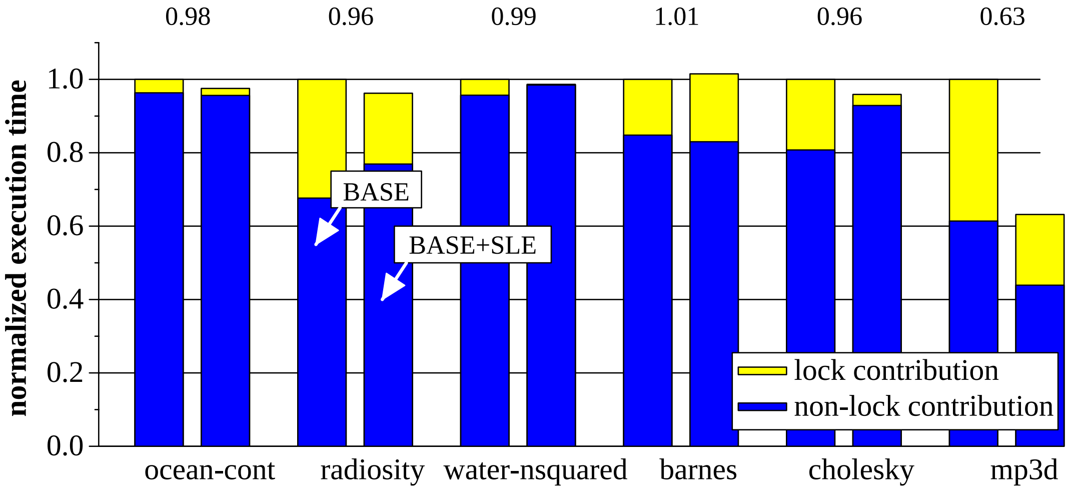


Figure 6-9: SLE performance for a 8-way DSM. *The y-axis is normalized parallel execution time. The first bar of each pair corresponds to the base case. The second bar of each pair corresponds to the SLE case. The fractions on top of the bar-pairs are normalized execution times for the SLE case. All normalizations are with respect to the base case.*

The distribute shared memory configuration (DSM) we use implements a very different coherence protocol from the CMP and SMP configurations. The DSM protocol is based on the SGI Origin 2000 protocol and is a NACK-based protocol whereas the CMP and SMP protocols are based on the Sun Gigaplane protocol and are non-NACK protocols.

Since the DSM configuration uses a NACK-based protocol, the performance impact of lock operations in a distributed shared-memory multiprocessor can be severe especially under contention. Thus, while the performance potential for SLE is high, the danger of performance degradation exists because the cost of misspeculation is also now higher—longer latencies result because of coherence protocol interference.

Figure 6-9 and Figure 6-10 show performance for the 8 and 16 thread configurations for DSM. `Ocean-cont` greatly benefits from SLE for a 16 thread DSM configuration because lock contention now contributed substantially to performance loss for this configuration. While `radiosity` and `cholesky` benefit from SLE, substantial lock contribution still remains because of data conflicts. `Barnes` experiences a performance loss for a 16 thread configuration because the cost of misspeculations are higher. `Mp3d` behaves similar to the SMP configuration because, similar to the SMP configuration, the level-two cache can accommodate the working set of the locks.

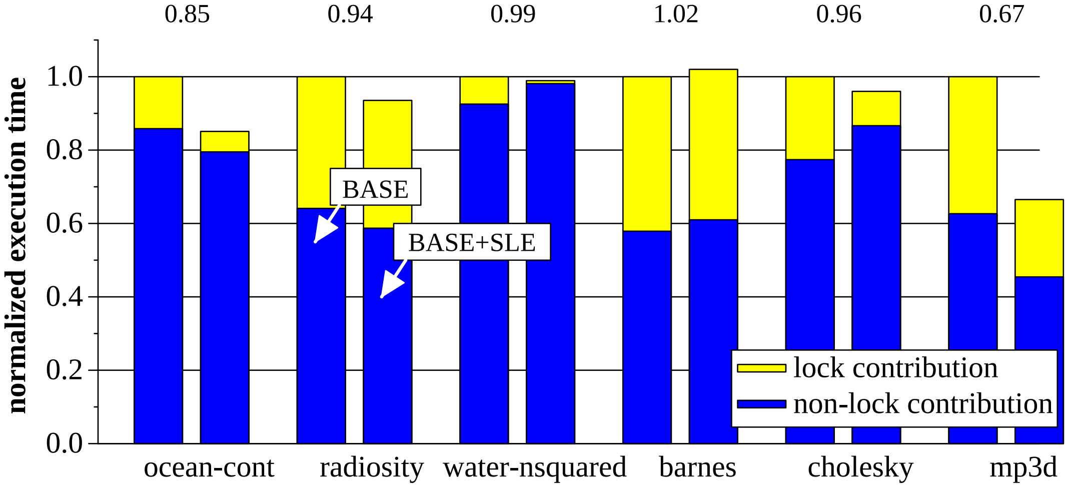


Figure 6-10: SLE performance for a 16-way DSM. *The y-axis is normalized parallel execution time. The first bar of each pair corresponds to the base case. The second bar of each pair corresponds to the SLE case. The fractions on top of the bar-pairs are normalized execution times for the SLE case. All normalizations are with respect to the base case.*

6.3.1.2 Restart thresholds

We look at SLE sensitivity to the restart threshold. The only benchmarks that were sensitive to the restart threshold are `radiosity`, `barnes`, and `cholesky`. These are the benchmarks that also suffer from lock contention and data conflicts.

For the CMP configuration, when we increases the restart threshold to 10, `barnes`, `radiosity`, and `cholesky` suffer from performance degradation (~5 to 10% performance loss)—processors repeatedly execute, misspeculate and restart thus consuming system resources, introducing coherence protocol interference. The SMP configuration is also sensitive to the restart threshold and follow similar trends. The DSM configuration is more sensitive to the restart threshold than the CMP or SMP configurations. A performance loss of up to 15% is observed across `barnes`, `radiosity`, and `cholesky`, when the restart threshold is kept at 10.

In summary, SLE performance is quite sensitive to the restart threshold and we conservatively use a threshold of 1 to mitigate performance loss due to coherence protocol interference. Exponential backoff in reissuing requests and Transactional Memory-type techniques may be employed to reduce the effects of protocol interference.

6.3.2 TLR performance

Substantial lock overhead remains for `radiosity`, `barnes`, and `cholesky` for SLE. In this section, we discuss the effectiveness of TLR in removing the remaining lock overhead. We add `raytrace` as an additional benchmark for evaluating TLR. `Raytrace` has a highly contended lock (and high data conflicts) and `raytrace` would not benefit from SLE. Since TLR targets conflicts, we also compare TLR performance to that of software queue-based MCS locks. We did not compare SLE performance in the earlier section to MCS because SLE, in the presence of data conflicts, would fall back on the base locking mechanism implemented using test&test&set. Thus, the performance comparison would end up being between MCS locks and test&test&set locks and prior work has shown MCS locks outperform test&test&set locks under contention [81].

We evaluate four configurations—1) BASE: base system, 2) BASE+SLE: base system with SLE optimization, 3) BASE+SLE+TLR: base system with SLE and TLR optimizations, and 4) MCS: system with MCS locks [120]. For convenience we will refer to these four schemes in text as BASE, SLE, TLR, and MCS respectively. BASE, SLE, and TLR use the same benchmark executable employing the test&test&set lock.

We focus discussion in this section on a 16 thread configuration for TLR and its performance is shown in Figure 6-11. The y-axis is normalized execution time. All bars are normalized to BASE. Each benchmark has three bars: the first bar is BASE. The second bar is SLE and the third bar is TLR. Each bar is divided into two parts: contributions due to lock variable accesses (loads and stores) and the remaining contributions. The lock portion only includes memory references to lock variables and does not necessarily include the total time spent in the synchronization algorithm itself (e.g., the branch instructions in the test&test&set algorithm are not counted here).

The stall accounting for the bars is similar to that used for SLE earlier and is performed at instruction commit—the instruction that stalls commit is charged the stall. The breakup is approximate since accounting for stall cycles due to individual operations is difficult and often inaccurate in out-of-order processors. In addition, a lock acquire operation involves an atomic read-modify-write instruction. This instruction cannot retire from the reorder buffer until it has been ordered by the memory system. In other words, such an operation acts like a memory fence[4] and remains at

[4] This behavior will occur independent of the memory consistency model implemented (including in release consistent systems). Any load operations past the lock acquire can still freely be issued because we support an aggressive implementation of total store ordering [45].

the head of the reorder buffer until any writes in the write buffer prior to the lock acquire are also drained and exposed to the memory system. Thus, the lock portion also accounts for the time it takes to flush the write buffer while a lock acquire is pending stalled at the head of the reorder buffer. For some benchmarks, the non-lock portion for the optimized case is larger than the non-lock portion for the base case. This is because sometimes removing locks puts other memory operations on the critical path. Speculative loads issued for data within critical sections that were earlier overlapped with the lock acquire operation are now exposed and stall the processor. A normalized execution time implies a lower bar is better.

All experiments employ the instruction-based predictor for collapsing read-modify-write sequences in critical sections into a single write operation thus reducing latencies within critical sections (Section 4.4.3). This results in a highly optimized base system execution and the performance numbers for TLR are thus conservative. Later, we discuss the effect this predictor has on the base system and present performance numbers to give an idea of how much better TLR would do against a more conventional base case. The speedup for technique *X* over technique *Y* is the ratio of the benchmark parallel cycle count with technique *Y* to that of the benchmark parallel cycle count with technique *X*. A speedup value greater than 1 is better.

`Ocean-cont` and `water-nsq` do not show much performance benefits. While `ocean-cont` has lock contention and opportunities for concurrent critical section execution, the performance impact on our target system is not much because lock accesses do not contribute much to performance loss. `Water-nsq` has frequent uncontended lock acquires. While the bars for BASE show potential for performance, removing locks does not result in a corresponding performance gain because the data cache misses within the critical section, that were earlier overlapped with lock access misses, are now exposed and account for the stalls. For, TLR speedup over BASE for `water-nsq` is 1.01 and for `ocean-cont` is 1.02. MCS performs the same as BASE for `ocean-cont`, and has a speedup of 0.96 (i.e., actually a performance loss) over BASE for `water-nsq`. The performance loss for MCS for `water-nsq` is due to the software overhead for uncontended locks.

For `radiosity`, speedup of TLR over BASE is 1.47 and nearly all locking overhead disappears. Speedup of MCS over BASE is 1.35. The task queue critical section was most contended for in `radiosity` and accounted for most conflict-induced restarts under TLR.

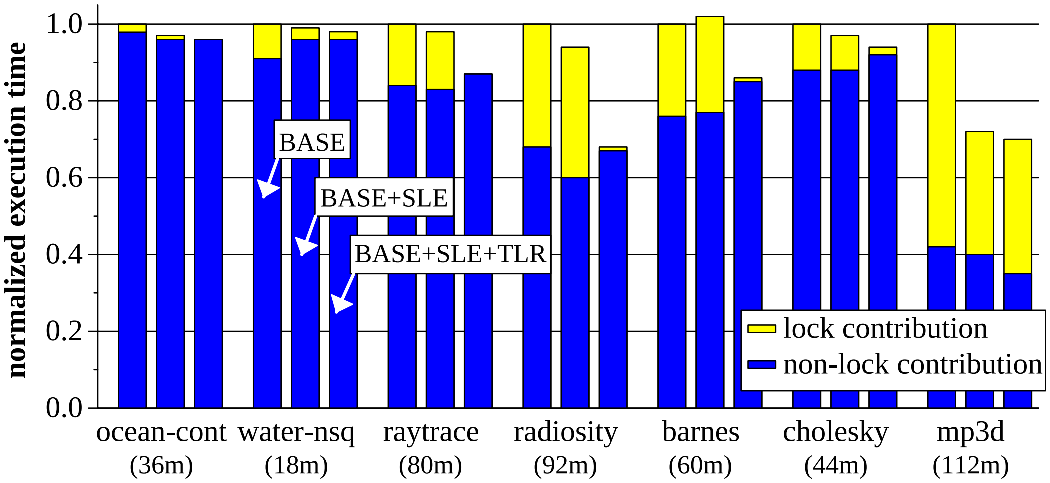


Figure 6-11: TLR performance for a 16-way CMP. *The y-axis is normalized execution time. All bars are normalized to the performance of BASE. Benchmarks are on the x-axis. Each benchmark has three bars: first bar is BASE, second bar is BASE+SLE and third bar is BASE+SLE+TLR. Each bar is divided into two parts: contributions due to lock variables (load and store instructions) and the remaining contributions. The number in parentheses below the benchmark name is the parallel execution cycle count, in millions, for the BASE shown as the first of three bars for each benchmark.*

For `raytrace`, the speedup of TLR over BASE is 1.17. MCS performance is similar to TLR. For `raytrace` (`car` input) on our system, lock contribution to execution time is 16%—much less than those reported earlier on systems with larger latencies, slower memory systems and different cache coherence protocols [81, 89].

For `barnes` TLR speedup over BASE is 1.16. However, MCS speedup over BASE is 1.21. MCS performs 4% better than TLR—the only application where MCS performs better than TLR. `Barnes` is based on a hierarchical octree representation of space in three dimensions and each node in the tree has its own lock. The root of this tree represents a space cell containing all bodies in the system. The tree is built by adding particles to the initially empty root cell and subdividing a cell into its eight children as soon as it contains more than single body. Most locking occurs in the tree building phase. Each process loads its bodies in the octree using locks to ensure atomic updates of the cell nodes. These locks tend to be contended and have data conflicts resulting in

Table 6-1: TLR-execution data conflict characteristics for 16 threads

	ocean-cont	water-nsq	raytrace	radiosity	barnes	cholesky	mp3d
no-conflict	62.4%	42.4%	91.9%	70.7%	79.2%	74.8%	91.6%
conflict	37.6%	57.6%	8.1%	29.3%	20.8%	21.5%	8.4%
resource	0%	0%	0%	0%	0%	3.7%	0%

TLR restarting frequently. TLR's restarts are due to sub-optimal ordering discussed earlier in Section 4.5. MCS constructs an ordered software queue and thus performs better than TLR.

`Cholesky`, with the `tk15.0` input set, is the only benchmark that cannot fit one critical section's data within the local cache. About 3.7% of dynamic critical section executions resulted in resource limitations for local buffering (write buffer limitations). This occurs at three functions (`ScatterUpdate`, `CompleteSuperNode`, and `ModifyColumn)` where a column in the matrix is locked and the algorithm then writes to the column entries resulting in buffer limitations (80% due to write buffer and 20% due to cache). TLR nevertheless achieves a speedup of 1.05 over BASE. MCS performs slightly worse than BASE (0.97).

`Mp3d` has frequent lock accesses but these locks are largely uncontended. The 128K data cache is unable to hold all locks and hence the processor suffers miss latency to locks. With TLR, significant lock contribution still remains. TLR achieves a speedup of 1.40 over BASE. BASE performs better than MCS (speedup over MCS: 1.47) because MCS pays a software overhead even for uncontended locks. This overhead adds up significantly if locking is frequent. TLR outperforms MCS by achieving a speedup of 2.06 because TLR pays no software overhead.

The performance gap between MCS and TLR for `barnes` and the TLR restarts in the applications suggests more optimizations are possible for TLR where coherence protocol support can be used. A similar gap (between TLR and an ideal TLR execution) was also observed in Figure 6-2 in Section 6.2.

6.3.2.1 TLR data conflict characteristics

Table 6-1 shows data conflict characteristics for a TLR execution for a 16 processor system. The rows correspond to the number of dynamic critical section executions that either had no conflict, a conflict, or a resource constraint. This may not correspond strictly to the number of lock acquires in the system because for nested critical sections are counted as a single instance. These do not include restart counts. For example, for raytrace 87,742 dynamic critical sections did not have any detected conflict-induced restarts and 7,754 had detected conflict-induced restarts for the given system. A conflict that is masked due to timestamp order induced deferral is counted under the no-conflict category. The accounting is performed only once per dynamic execution—repeated restarts are not counted here.

6.3.2.2 Impact of TLR on network traffic

Figure 6-12 shows the network traffic for various configurations. The y-axis is normalized message count. The first bar is the BASE. The second bar corresponds to MCS and is normalized to BASE. The third bar is TLR and is also normalized to BASE. The lower portion of the bar is the

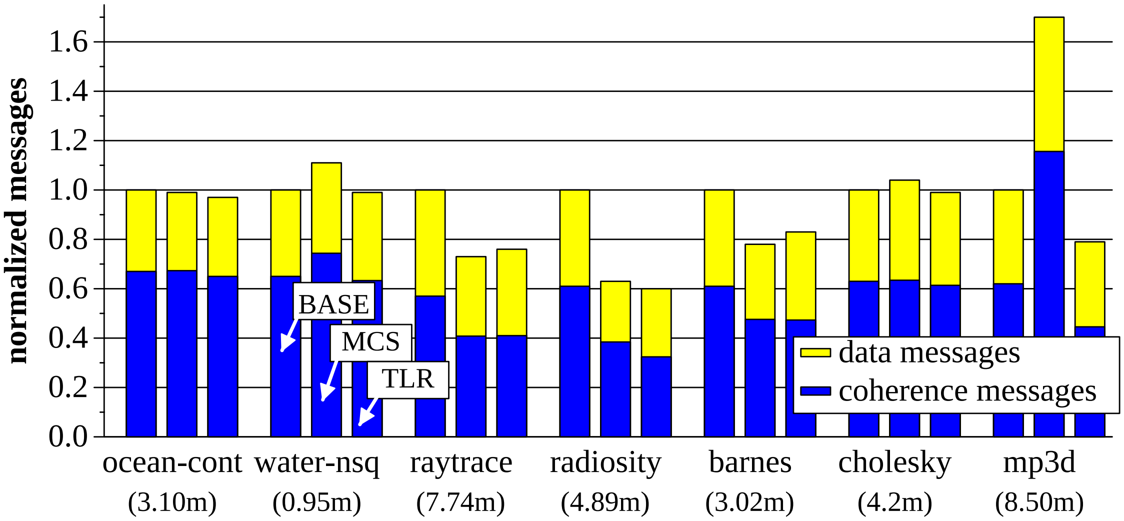


Figure 6-12: Impact of TLR on network traffic. *The y-axis is normalized message count. The first bar is the BASE. The second bar corresponds to MCS and is normalized to BASE. The third bar is TLR and is also normalized to BASE. The lower portion of the bar is the coherence traffic contribution (read for shared and exclusives, upgrades, write-backs, and instruction fetches) and the upper portion is the data traffic. The numbers below the benchmarks are the total number of messages, in millions, sent in the network for BASE. For TLR, two additional categories exist: marker messages and probe messages. These are not shown because their numbers are very small: ocean: 0.09% 0.05%, water: 0.6% ~0%, raytrace: 0.4% 0.2%, radiosity: 0.24% 0.06%, barnes: 1.66% 0.40%, cholesky: 0.6% 0.01%, and mp3d: 0.3% and 0.03%. TLR introduces minimal probe and marker messages as compared to the total number of messages in the system.*

coherence traffic contribution (read for shared and exclusives, upgrades, write-backs, and instruction fetches) and the upper portion is the data traffic. The numbers below the benchmarks are the total number of messages, in millions, sent in the network for BASE. For TLR, two additional categories exist: marker messages and probe messages. These are not shown because their numbers are very small: ocean: 0.09% 0.05%, water: 0.6% ~0%, raytrace: 0.4% 0.2%, radiosity: 0.24% 0.06%, barnes: 1.66% 0.40%, cholesky: 0.6% 0.01%, and mp3d: 0.3% and 0.03%. TLR introduces minimal probe and marker messages as compared to the total number of messages in the system

6.3.2.3 Coarse-grain vs. fine-grain experiment

With `mp3d`, a noticeable locking overhead remained and we investigated it further. We conjectured replacing the per-cell fine-grain locks in `mp3d` by one single coarse-grain lock should provide better performance because the data foot-print reduces and the memory system behavior should improve substantially. We replaced the individual cell locks in `mp3d` with a single lock. This is bad for BASE (and MCS) because now the benchmark has severe lock contention. As expected, TLR with one lock for all cells in `mp3d` outperforms BASE with fine-grain per-cell locks by 58% (speedup 2.40) and outperforms TLR with fine-grain per-cell locks by 41% (speedup 1.70). Thus, using coarse-grain locks can improve performance significantly over fine-grain locks.

6.3.2.4 Read-modify-write prediction effects

The performance we report for the BASE case uses the instruction-based predictor for collapsing read-modify-write sequences within predicted critical sections into a single write operation. We give speedups of BASE with the predictor (the results in Figure 6-11) with respect to BASE without the predictor (BASE-no-opt: a more conventional base case). The speedup is calculated as the ratio of the parallel cycle count for BASE and parallel cycle count for BASE-no-opt. A speedup value greater than 1 is better. The speedups are—`ocean-cont:`1.00, `water-nsq`: 1.04, `raytrace`: 1.28, `radiosity:` 1.05, `barnes`: 1.04, `cholesky`: 1.33, and `mp3d`: 1.13. With the optimization, the time spent waiting for lock operations increases because critical section data latencies are reduced. Thus, our speedups in Figure 6-11 would be much larger if we assumed a more conventional base case without the predictor. For all benchmarks, a 128 entry PC-indexed predictor was sufficient (only `radiosity` used more than 30 entries—using just under 100) and most of the remaining benchmarks used less than 20 entries). The above results suggest supporting such a predictor even in systems without TLR.

6.4 Chapter summary

We have demonstrated with a set of microbenchmarks and applications that SLE and TLR have the potential to outperform common locking algorithms transparently. We now summarize the results for microbenchmarks and benchmarks.

6.4.1 Microbenchmark summary

In the absence of data conflicts, SLE can remove all dependence on locks and locking overhead as was demonstrated by the `multiple-counter` microbenchmark. TLR behaves identical to SLE as demonstrated by the `multiple-counter` microbenchmark.

In the presence of data conflicts, TLR can remove locking overhead and perform coordinated and low latency data transfer among conflicting processors as was demonstrated by the `single-counter` microbenchmark. For the `single-counter` and `doubly-linked list` microbenchmarks, SLE degrades to BASE performance because speculation is not performed. SLE performs a little poorer than BASE because the cost of misspeculation cannot be recovered. TLR consistently outperforms BASE.

TLR can extract and exploit dynamic concurrency in programs—situations where data conflict varies along an execution. For example, in the `doubly-linked list` TLR identified dynamically circumstances where `enqueue`() and `dequeue`() operations could occur concurrently.

TLR performance is sensitive to the ordering of the chains constructed for deadlock-free concurrency control. This was observed in the `single-counter` microbenchmark. Selectively relaxing timestamp order gave consistently better and stable performance as compared to a timestamp-enforced ordering. This suggests to future work for further improving performance of TLR.

Except for situations where using unfair primitives gets better performance than when using fair algorithms, TLR outperforms test&test&set without requiring any changes to the software.

TLR consistently outperforms MCS for all microbenchmarks. MCS experiences a fixed software overhead independent of the concurrency possible in the benchmark. TLR does not suffer software overhead and performs coordinated and efficient data transfer between any conflicting processors.

6.4.2 Benchmark summary

We now discuss performance for the benchmarks we use. SLE performance is sensitive to true data sharing. If frequent data conflicts occur, SLE should not be applied to those critical sections else performance loss may occur.

SLE performance is sensitive to restart threshold. In `barnes` and `radiosity` performance loss of up to 15% was observed for some configurations if the restart threshold is not chosen carefully.

For critical sections in our benchmarks, local buffering resources were sufficient to handle critical section data accesses for all but one configuration. The only configuration where the write buffer was not sufficiently large was `cholesky` for the input set tk15.0 where about 3.7% critical sections experienced limited resources.

Using TLR removes sensitivity of performance on the restart threshold. This is because TLR provides a lock-free execution even in the presence of high data conflicts. TLR never performs worse than BASE or SLE for the applications and system configurations chosen.

TLR performs better than MCS for all benchmarks except `barnes`. In the presence of contention, TLR performs better than MCS while in the absence of contention, TLR outperforms MCS. MCS performs better than TLR for one benchmark because MCS constructs an ordered queue of requestors while TLR undergoes restarts due to a mismatch between coherence-order and timestamp-order.

TLR with coarse-grain locks can outperform the BASE system with fine-grain locks. This was seen for `mp3d` where replacing all cell locks by a single lock resulted in TLR achieving a speedup of 2.4 over BASE and 1.7 over TLR with fine-grain locks.

If the locks can be cached locally, the overhead of locking can be nearly eliminated. Only `mp3d` was an example where a small cache size resulted in frequent evictions. Replacing fine-grain locks by coarse-grain locks (as described above) addresses this issue quite effectively.

The read-modify-write sequence predictor for collapsing such sequences within critical sections is quite effective even without TLR with large performance gains being obtained (speedup of 1.00 up to 1.33).

TLR adds minimal extra messages (probes and marker messages) for concurrency control as can be seen from Figure 6-12. Further, the total number of messages in the system with TLR is consistently lesser than systems without TLR and MCS-based systems.

Chapter 7

Conclusion

This dissertation provides the first solution that bridges the long-standing gap between writing correct and stable multithreaded code and writing high-performance multithreaded code. We first summarize the contributions of the thesis in Section 7.1 and discuss some future research directions in Section 7.2.

7.1 Contributions

This thesis makes two core contributions—Speculative Lock Elision and Transactional Lock Removal—as a step towards achieving transparent high-performance lock-free and reliable execution of multithreaded programs.

7.1.1 Speculative Lock Elision

Speculative Lock Elision (SLE) is a microarchitectural technique to remove unnecessary serialization from a dynamic instruction stream. The key idea behind SLE involves using the cache coherence protocol to obtain appropriate permissions on the necessary cache blocks, accessing and modifying data speculatively if needed, and then providing the appearance of instantly committing the critical section by making updates visible to other processors at a single commit point.

Three key features of SLE are:

1. *Enables highly concurrent multithreaded execution*. Multiple threads can concurrently execute critical sections guarded by the same lock. Correctness is determined without acquiring (or modifying) the lock. No write permissions are required on the lock variable in the event of a successful speculation.

2. *Simplifies correct multithreaded code development*. Programmers can use conservative synchronization to write correct multithreaded programs without significant performance impact. If the synchronization is not required for a correct execution, the execution will behave as if the synchronization were not present.

3. *Can be implemented easily.* SLE can be implemented entirely in the microarchitecture, without instruction set support and without system-level modifications (e.g., no coherence protocol changes are required) and is transparent to programmers. Existing synchronization instructions are identified dynamically. Programmers do not have to learn a new programming methodology and can continue to use well understood synchronization routines. The technique can be incorporated into modern processor designs, independent of the system and the cache coherence protocol.

SLE is a step towards enabling high-performance multithreaded programming. SLE permits programmers to use frequent and conservative synchronization to write *correct* multithreaded code easily; SLE automatically and dynamically removes unnecessary instances of synchronization in the absence of data conflicts.

7.1.2 Transactional Lock Removal

Transactional Lock Removal (TLR) is a hardware mechanism to convert lock-based critical sections transparently and optimistically into lock-free optimistic transactions and a timestamp-based fair conflict resolution scheme to provide transactional semantics and starvation freedom, if the data accessed by the transaction can be locally cached and subject to some implementation specific constraints.

While SLE provides benefits in the absence of data conflicts, TLR provides benefits even in the presence of conflicts. TLR provides both serializability and failure atomicity. TLR transparently and cleanly addresses the trade-off among programmability, performance, and stability discussed earlier in Section 1.2. We have presented one deferral-based implementation of TLR that does not require changes to the coherence protocol state transitions.

Three contributions of TLR are:

1. *Programmability.* TLR simplifies correct multithreaded code development. Reasoning about granularity of locks is not required because serialization decisions are made at run time based

on actual data conflicts and independent of lock granularity. Cache blocks are the coherence unit and represent a fine granularity for sharing. TLR provides this granularity without programmer involvement.

2. *Stability.* Since locks are not written to and the “wait” on the lock variable is no longer required, properties of lock-free and wait-free execution are achieved transparently. This translates to improved system wide performance, no convoying or priority-inversion dangers, and robust execution in the presence of failing threads. TLR addresses the inherent limitations of the locking construct while maintaining the well-understood critical section abstraction for the programmer.

3. *Performance*. TLR enables high-performance multithreaded execution. Independent of lock granularity, because serialization decisions are made only in the presence of data conflicts and is not based on lock contention, performance of fine-granularity locking is achieved. Further, since a queue of requestors is constructed in the hardware by using the coherence protocol, the data transfers are efficient and low overhead. Programmers can focus on writing correct code while hardware automatically extracts performance.

TLR is the first proposal to address the trade-off among all the above three aspects and provide a robust solution to the synchronization problem. While TLR does trade off hardware for these properties, we believe the hardware cost is modest. Subject to resource constraints, our scheme is the first to transparently provide a wait-free execution of a lock-based critical section.

We showed hardware with TLR outperforms hardware without TLR and performs better than MCS locks for a range of microbenchmarks and benchmarks representing both high and low data conflict conditions. Importantly, TLR provides sustained high performance even for fine-grain sharing by providing efficient and coordinated data transfer among conflicting threads. Further, TLR provides better load balancing because of the use of timestamps for fairness.

7.2 Future directions

We have introduced the concepts of SLE and TLR and given an implementation of each of the techniques. In this section we discuss future research directions. We first discuss directions for improving the core mechanisms of SLE and TLR themselves. Then we discuss the ways operating systems can be involved to improve the TLR guarantees and ways to exploit the programmability benefits of TLR.

7.2.1 SLE mechanisms

SLE performance is sensitive to the restart threshold. While the experiments in this thesis have used a static restart threshold, dynamically selecting such a threshold may provide better performance characteristics. Future work remains in selecting dynamic restart thresholds depending upon the application phase.

Further work remains in confidence predictors for deciding which locks to apply SLE to. If data-conflict-induced restarts repeatedly occur for a given lock, applying SLE in those situations may degrade performance. Sophisticated mechanisms for determining when to apply SLE should be investigated.

7.2.2 TLR mechanisms

TLR removes sensitivity of performance to the restart threshold because TLR provides a lock-free execution even in the presence of data conflicts. The design space for TLR is large and this dissertation has investigated only part of the design space—namely the use of wound-wait-type algorithms using request deferrals. While this design often provides high performance, sometimes performance was sub-optimal.[1] An example is where MCS performs better than TLR for the benchmark `barnes`. Further, the `single-counter` microbenchmark emphasized the performance difference that may arise if coherence-order is different from timestamp-order.

Future work remains in studying new mechanisms for keeping the queues due to coherence-order and timestamp-order matched. Section 4.4.4 discussed in detail the performance interactions of timestamp-ordered queues and coherence-ordered queues. Performance was sub-optimal when these queues were out of order with respect to each other.

Selectively relaxing timestamp-order for performance was discussed in Section 4.4.5 and modified TLR algorithms were briefly discussed in Section 4.4.2.3 where a wounded processor

[1] Performance is often still better than without TLR even though it may not be optimal.

may potentially continue executing in TLR mode for a while under certain conditions. These techniques present potential for performance improvement of TLR and should be investigated further.

The wait-die mechanism should be investigated along with hybrid mechanisms of wound-wait and wait-die. Selective use of NACKs may also help performance in certain implementations.

7.2.3 Stability and programmability interactions

While TLR provides the mechanisms for providing a wait-free execution of a critical section subject to certain implementation-specific constraints (see Section 4.6), the conditional aspect must be investigated in more detail. Often this aspect is a function of the specific microarchitecture and system. Events such as certain interrupts[2] and the scheduling quantum may make it difficult or impossible to guarantee TLR. While a misspeculation can always be triggered and the lock acquired thus guaranteeing correct execution even under these conditions, transactional properties may not be maintained. While these limitations can be specified to the programmer so that the programmer can write programs while taking these into consideration, operating system support can be used to improve upon these guarantees.

Subject to resource constraints, TLR can provide a wait-free execution of a synchronization primitive such as critical sections. Wait freedom is a more broadly applicable property than merely for critical sections and is a function of the actual program itself. Critical sections are only one aspect but a key aspect because they make reasoning about correctness of sharing easy. The next step is to investigate how programmers can exploit the understanding of conditional wait-free execution and write algorithms and programs accordingly. TLR provides the programmer with a high-performance implementation of conditional wait-free synchronization and the performance limitations of wait-free algorithms can potentially be addressed. It would be useful for the hardware to inform the program whether TLR was not successful for a given instance to maintain certain properties so that the program can proceed to a more heavy-weight slow algorithm that meets the guarantees without TLR. This way, in the common case TLR provides guarantees with high performance and in the uncommon case (such as resource-induced failures and other unexpected

[2] Interrupts that can be delayed for a finite time do not pose a problem.

conditions) the more heavy-weight algorithm can be invoked to achieve the TLR guarantees of wait freedom and starvation freedom.

Reducing hardware requirements by adding some support in software should also be investigated. These include adding hints in software for identifying transactions. These hints need not always be correct because TLR does not rely on them for correctness.

Bibliography

[1] Uri Abraham. Self-Stabilizing Timestamps. Unpublished Manuscript, April 2002.

[2] Sarita V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin, Madison, WI, 1993. Also appears as Technical Report CS-TR-93-1198, Computer Sciences Department, University of Wisconsin, Madison, WI, Dec 1993.

[3] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[4] Sarita V. Adve and Mark D. Hill. Weak Ordering—A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[5] Juan Allemany and Ed Felten. Performance Issues in Non-blocking Synchronization on Shared-Memory Multiprocessors. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 125–134, August 1992.

[6] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[7] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr. Architecture of the IBM System/360. *IBM Journal of Research and Development*, 8(2):87–101, April 1964.

[8] James H. Anderson and Mark Moir. Universal Constructions for Large Objects. In *9th International Workshop on Distributed Algorithms*, pages 168–182, September 1995.

[9] Thomas E. Anderson. The Performance Implications of Spin-Waiting Alternatives for Shared-Memory Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II (software), pages 170–174, August 1989.

[10] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[11] Greg Barnes. Method for Implementing Lock-Free Shared Data Structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, June 1993.

[12] R. Bayer and E.M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1:173–189, 1972.

[13] Michael Beck, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Harold Bohme. *Linux Kernels Internals*. Addison-Wesley, Reading, MA, second edition, 1998.

[14] Philip A. Bernstein and Nathan Goodman. Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems. In *Sixth International Conference on Very Large Data Bases*, pages 285–300, October 1980.

[15] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, first edition, 1987.

[16] Phillip A. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[17] Brian N. Bershad. Practical Considerations for Lock-Free Concurrent Objects. Technical Report CMU-CS-91-183, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1991.

[18] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast Mutual Exclusion for Uniprocessors. In *Proceedings of the Fifth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 223–233, October 1992.

[19] Philip Bitar and Alvin M. Despain. Multiprocessor Cache Synchronization: Issues, Innovations, Evolution. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 424–433, June 1986.

[20] Philip Bohannon, Daniel Lieuwen, Avi Silberschatz, S. Sudarshan, and Jacques Gava. Recoverable User-Level Mutual Exclusion. In *Proceedings of the seventh IEEE Symposium on Parallel and Distributed Processing*, pages 293–301, October 1995.

[21] Tony Brewer and Greg Astfalk. The Evolution of the HP/Convex Exemplar. In *Proceedings of the 42nd IEEE Computer Society International Conference (COMPCON)*, pages 81–86, February 1997.

[22] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, Computer Sciences Department, University of Wisconsin, Madison, WI, June 1997.

[23] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C.K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on the Management of Data*, pages 383–394, May 1994.

[24] Richard P. Case and Andris Padegs. Architecture of the IBM System/370. *Communications of the ACM*, 21(1):73–96, January 1978.

[25] Donald D. Chamberlin, Raymond F. Boyce, and Irving L. Traiger. A Deadlock-Free Scheme for Resource Locking in a Data-Base Environment. *Proceedings of IFIP Congress 74*, pages 340–343, August 1974.

[26] Alan Charlesworth, Alan Phelps, Ricki Williams, and Gary Gilbert. Gigaplane-XB: Extending the Ultra Enterprise Family. In *Proceedings of the Symposium on High Performance Interconnects V*, pages 97–112, August 1997.

[27] Sung-Eun Choi and E. Christopher Lewis. A Study of Common Pitfalls in Simple Multi-Threaded Programs. In *Proceedings of the Thirty-First ACM SIGCSE Technical Symposium on Computer Science Education*, pages 325–329, March 2000.

[28] *Alpha Architecture Handbook Version 4*, October 1998.

[29] IBM Corporation. The RS/6000 Enterprise Server Model S80. IBM Whitepaper available from http://www.rs6000.ibm.com, 1999.

[30] Intel Corporation. Hyper-Threading Technology. http://developer.intel.com/technology/hyperthread/, 2001.

[31] International Business Machines Corporation. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufman, San Francisco, CA, 1998.

[32] Alan L. Cox and Robert J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[33] Travis S. Craig. Building FIFO and Priority-Queueing Spin Locks from Atomic Swap. Technical Report UW-CSE-93-02-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA, February 1993.

[34] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman, San Francisco, CA, 1999.

[35] Keith Diefendorff. Compaq Chooses SMT for Alpha. *Microprocessor Report*, 13(16), December 1999.

[36] Edsger W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, September 1965.

[37] Danny Dolev and Nir Shavit. Bounded Concurrent Time-Stamping. *SIAM Journal on Computing*, 26(2):418–455, 1997.

[38] Cynthia Dwork and Orli Waarts. Simple and Efficient Bounded Concurrent Timestamping or Bounded Concurrent Timestamp Systems are Comprehensible! In *24th Annual ACM Symposium on the Theory of Computing*, pages 655–666, May 1992.

[39] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[40] Manoj Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin, Madison, WI, 1993. Also appears as Technical Report CS-TR-93-1196, Computer Sciences Department, University of Wisconsin, Madison, WI, Nov 1993.

[41] Manoj Franklin and Gurindar S. Sohi. The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67, May 1992.

[42] Manoj Franklin and Gurindar S. Sohi. A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[43] Volker Gaede and Oliver Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[44] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, Stanford, CA, 1995. Also appears as Technical Report CSL-TR-95-685, Computer Systems Laboratory, Stanford University, Stanford, CA, Dec 1995.

[45] Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, August 1991.

[46] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta,

and John L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[47] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Van Doren. Architecture and Design of AlphaServer GS320. In *Proceedings of the Ninth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, November 2000.

[48] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, May 1999.

[49] James R. Goodman. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the Second Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 72–81, October 1987.

[50] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Shared-Memory Multiprocessors. In *Proceedings of the Third Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, April 1989.

[51] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, May 1988.

[52] Sridhar Gopal, T.N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative Versioning Cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.

[53] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[54] Silicon Graphics. *MIPS R10000 Microprocessor User's Manual Version 2.0*. Silicon Graphics Inc., Mountain View, CA, 1996.

[55] Gary Graunke and Shreekant Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[56] Jim Gray. Notes on Database Operating Systems. In R. Bayer, R. M. Graham, J. H. Saltzer, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer Verlag, Berlin, 1978.

[57] Jim Gray. The Transaction Concept: Virtues and Limitations. In *Seventh International Conference on Very Large Data Bases*, pages 144–154, September 1981.

[58] Michael Greenwald. *Non-blocking Synchronization and System Design*. PhD thesis, Stanford University, Stanford, CA, 1999. Also appears as Stanford University Technical Report STAN-CS- TR-99-1624, Stanford University, Stanford, CA, 1999.

[59] Linley Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, 12(14):12–15, October 1998.

[60] Theo Haerder. Observations on Optimistic Concurrency Control Schemes. *Information Sys-*

tems, 9(2):111–120, 1984.

[61] Lance Hammond, Mark Willey, and Kunle Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the Eighth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[62] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, San Francisco, CA, second edition, 1995.

[63] Maurice Herlihy. Impossibility and Universality Results for Wait-Free Synchronization. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 276–290, August 1988.

[64] Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[65] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[66] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[67] Maurice Herlihy and Jeannette M. Wing. Axioms for Concurrent Objects. In *Proceedings of the 14th Annual Symposium on Principles of Programming Languages*, pages 13–26, January 1987.

[68] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[69] Mark D. Hill. Multiprocessors Should Support Simple Memory Consistency Models. *IEEE Computer*, 31(8):28–34, August 1998.

[70] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, February 2001.

[71] Richard C. Holt. Some Deadlock Properties of Computer Systems. *ACM Computing Surveys*, 4(3):179–195, December 1972.

[72] Institute of Electrical and Electronics Engineers, New York, NY. *IEEE Standard for the Scalable Coherent Interface (SCI)*, August 1993. ANSI/IEEE Std. 1596-1992.

[73] Intel Corporation. *Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Manual*, January 1996.

[74] Amos Israeli and Ming Li. Bounded Time Stamps. *Distributed Computing*, 6(4):205–209, 1993.

[75] Amos Israeli and Lihu Rappoport. Efficient Wait-Free Implementation of a Concurrent Priority Queue. In *7th International Workshop on Distributed Algorithms*, pages 27–29, September 1993.

[76] Amos Israeli and Lihu Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, pages 151–160, August 1994.

[77] Van Jacobson, Robert Braden, and David Borman. TCP Extensions for High Performance–RFC 1323, May 1993.

[78] Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, CA, November 1987.

[79] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[80] Alain Kägi. *Mechanisms for Efficient Shared-Memory, Lock-Based Synchronization*. PhD thesis, University of Wisconsin, Madison, WI, May 1999.

[81] Alain Kägi, Doug Burger, and James R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 170–180, June 1997.

[82] Jim Kahle. A Dual-CPU Processor Chip. In *Proceedings of the 1999 International Microprocessor Forum*, October 1999.

[83] Tareef S. Kawaf, D. John Shakshober, and David C. Stanley. Performance Analysis Using Very Large Memory on the 64-bit AlphaServer System. *Digital Technical Journal*, 8(3), 1996.

[84] Stefanos Kaxiras and James R. Goodman. Improving CC-NUMA Performance Using Instruction-Based Prediction. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 161–170, January 1999.

[85] Richard E. Kessler and James L. Schwarzmeier. Cray T3D: A New Dimension for Cray Research. In *Proceedings of the 38th IEEE Computer Society International Conference (COMPCON)*, pages 176–182, February 1993.

[86] Thomas F. Knight. An Architecture for Mostly Functional Languages. In *Proceedings of ACM Lisp and Functional Programming Conference*, pages 500–519, August 1986.

[87] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-Conscious Synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, February 1997.

[88] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the Eighth Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[89] Sanjeev Kumar, Dongming Jiang, Rohit Chandra, and Jaswinder P. Singh. Evaluating Synchronization on Shared Address Space Multiprocessors: Methodology and Performance. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 23–34, May 1999.

[90] H.T. Kung and John T. Robinson. On Optimistic Methods of Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[91] Steven R. Kunkel, Bill Armstrong, and Philip Vitale. System Optimization for OLTP Workloads. *IEEE Micro*, 19(3):55–64, May/June 1999.

[92] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach Featur-*

ing the Internet. Addison-Wesley, Reading, MA, first edition, 2001.

[93] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[94] Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[95] Leslie Lamport. Concurrent Reading and Writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[96] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[97] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(6):558–565, July 1978.

[98] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[99] Leslie Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[100] Leslie Lamport. The Mutual Exclusion Problem: Part I — The Theory of Interprocess Communication. *Journal of the Association for Computing Machinery*, 33(2):313–326, April 1986.

[101] Leslie Lamport. The Mutual Exclusion Problem: Part II — Statement and Solutions. *Journal of the Association for Computing Machinery*, 33(2):327–348, April 1986.

[102] Leslie Lamport. On Interprocess Communication–Part I: Basic Formalism, Part II: Algorithms. *Distributed Computing*, 1(2):77–85, 1986.

[103] Leslie Lamport, Sharon E. Perl, and William E. Weihl. When Does a Correct Mutual Exclusion Algorithm Guarantee Mutual Exclusion. *Information Processing Letters*, 76(3):131–134, December 2000.

[104] Butler W. Lampson and David D. Redell. Experiences with Processes and Monitors in Mesa. *Communications of the ACM*, 22(2):105–117, February 1980.

[105] James Laudon and Daniel E. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[106] D.D. Lee and Randy H. Katz. Using Cache Mechanisms to Exploit Nonrefreshing DRAM's for On-Chip Memories. *IEEE Journal of Solid-State Circuits*, 26(4):657–666, April 1991.

[107] Joonwon Lee and Umakishore Ramachandran. Synchronization with Multiprocessor Caches. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 27–37, May 1990.

[108] Daniel Leibholz and Rahul Razdan. The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor. In *Proceedings of the 42nd IEEE Computer Society International Conference (COMPCON)*, pages 28–36, February 1997.

[109] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

[110] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John L. Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[111] Kevin M. Lepak and Mikko H. Lipasti. On the Value Locality of Store Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 182–191, June 2000.

[112] Kevin M. Lepak and Mikko H. Lipasti. Temporally Silent Stores. In *Proceedings of the Tenth Symposium on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[113] Beng-Hong Lim and Anant Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 25–35, October 1994.

[114] Mikko H. Lipasti and John P. Shen. Exceeding the Dataflow Limit Via Value Prediction. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 226–237, December 1996.

[115] Tom Lovett and Russel Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, May 1996.

[116] Peter Magnusson, Anders Landin, and Erik Hagersten. Efficient Software Synchronization on Large Cache Coherent Multiprocessors. Technical Report T94:07, Swedish Institute of Computer Science, Kista, Sweden, February 1994.

[117] Jose F. Martínez and Josep Torrellas. Speculative Locks for Concurrent Execution of Critical Sections in Shared-Memory Multiprocessors. In *Workshop on Memory Performance Issues*, June 2001.

[118] Jose F. Martínez and Josep Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *Proceedings of the Tenth Symposium on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[119] Henry Massalin and Calton Pu. A Lock-Free Multiprocessor OS Kernel. Technical Report CUCS-005-91, Computer Science Department, Columbia University, New York, NY, May 1991.

[120] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[121] John M. Mellor-Crummey and Michael L. Scott. Synchronization Without Contention. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, April 1991.

[122] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-blocking and

Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 267–275, May 1996.

[123] Maged M. Michael and Michael L. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[124] C. Mohan. Less Optimism About Optimistic Concurrency Control. In *Proceedings of the Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, pages 199–204, February 1992.

[125] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[126] Mark Moir. *Efficient Object Sharing in Shared-Memory Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, NC, 1996.

[127] Mark Moir. Transparent Support for Wait-Free Transactions. In *11th International Workshop on Distributed Algorithms*, pages 305–319, September 1997.

[128] Mark Moir. Laziness Pays! Using Lazy Synchronization Mechanisms to Improve Nonblocking Construction. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 61–70, July 2000.

[129] Andreas Moshovos, Scott Breach, T.N. Vijaykumar, and Gurindar S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193, June 1997.

[130] Jeffrey Naughton. Personal Communication. August 2001.

[131] N. Nishi, T. Inoue, M. Nomura, S. Matsushita, S. Torii, A. Shibayama, J. Sakai, T. Ohsawa, Y. Nakamura, S. Shimada, Y. Ito, M. Edahiro, K. i. Minami, O. Matsuo, H. Inoue, T. Manabe, T. Horiuchi, M. Motomura, M. Yamashina, and M. Fukuma. A 1GIPS 1W Single-Chip Tightly-Coupled Four-Way Multiprocessor with Architecture Support for Multiple Control Flow Execution. In *Proceedings of the 2000 IEEE International Solid-State Circuits Conference*, February 2000.

[132] R. R. Oehler and Randy D. Groves. IBM RISC System/6000 processor architecture. *IBM Journal of Research and Development*, 34(1):23–36, January 1990.

[133] Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, October 1996.

[134] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the Association for Computing Machinery*, 26(4):631–653, October 1979.

[135] Christos H. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, MD, first edition, 1986.

[136] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *Journal of the Association for Computing Machinery*, 27(2):228–234, April

1980.

[137] Gregory F. Pfister, William C. Brantley, David A. George, Steve L. Harvey, Wally J. Kleinfelder, Kevin P. McAuliffe, Evelin A. Melton, V. Alan Norton, and Jodi Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, August 1985.

[138] Serge A. Plotkin. Sticky Bits and Universality of Consensus. In *Proceedings of the Eighth ACM Symposium on Principles of Distributed Computing*, pages 159–175, August 1989.

[139] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 294–305, December 2001.

[140] Ravi Rajwar and James R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the Tenth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, October 2002.

[141] Ravi Rajwar, Alain Kägi, and James R. Goodman. Improving the Throughput of Synchronization by Insertion of Delays. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 168–179, January 2000.

[142] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw Hill, second edition, 2000.

[143] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 199–210, June 1997.

[144] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems*, 3(2):178–198, June 1978.

[145] Larry Rudolph and Zary Segall. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[146] Takayuki Sato, Kazuhiko Ohno, and Hiroshi Nakashima. A Mechanism for Speculative Memory Accesses Following Synchronizing Operations. In *14th International Parallel and Distributed Processing Symposium*, May 2000.

[147] Christoph Scheurich and Michel Dubois. Correct Memory Operation of Cache-Based Multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 234–243, June 1987.

[148] Steven L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, October 1996.

[149] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.

[150] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[151] Ashok Singhal, David Broniarczyk, Frederick M. Cerauskis, Jeff Price, L. Yuan, Gerald Cheng, D. Doblar, S. Fosth, Nalini Agarwal, K. Harvey, and Erik Hagersten. Gigaplane: A High Performance Bus for Large SMPs. In *Proceedings of the Symposium on High Performance Interconnects IV*, pages 41–52, August 1996.

[152] Ashok Singhal, Bjorn Liencres, Jeff Price, Frederick M. Cerauskis, David Broniarczyk, Gerald Cheung, Erik Hagersten, and Nalini Agarwal. Implementing Snooping on a Split-Transaction Computer System Bus. US Patent 5,978,874, November 1999.

[153] James E. Smith and Andrew R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, June 1985.

[154] Gurindar S. Sohi, Scott E. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[155] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, May 2002.

[156] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, June 2000.

[157] Per Stenström, Mats Brorsson, and Lars Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[158] Janice M. Stone, Harold S. Stone, Phil Heidelberger, and John Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel & Distributed Technology*, 1(4):58–71, November 1993.

[159] Salvatore Storino and John Borkenhagen. A Multi-Threaded 64-bit PowerPC Commercial RISC Processor design. In *Proceedings of the 11th Annual International Symposium on High-Performance Chips*, August 1999.

[160] Paul Sweazey and Alan J. Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, June 1986.

[161] Alexander Thomasian. Concurrency Control: Methods, Performance, and Analysis. *ACM Computing Surveys*, 30(1):70–119, March 1998.

[162] Marc Tremblay. An Architecture for the New Millenium. In *Proceedings of the 11th Annual International Symposium on High-Performance Chips*, August 1999.

[163] John Turek, Dennis Shasha, and Sundeep Prakash. Locking Without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 212–222, August 1992.

[164] John D. Valois. *Lock-Free Data Structures*. PhD thesis, Rochester Institute of Technology, Rochester, NY, 1995.

[165] John D. Valois. Lock-Free Linked Lists Using Compare-and-Swap. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 214–222, August 1995.

[166] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual: Version 9/ SPARC International*. Prentice Hall, Englewood Cliffs, NJ, 1994.

[167] Jeanette M. Wing and Chun Gong. Testing and Verifying Concurrent Objects. *Journal of Parallel and Distributed Computing*, 17(2):164–182, January/February 1993.

[168] Philip J. Woest and James R. Goodman. An Analysis of Synchronization Mechanisms in Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 152–165, April 1991.

[169] Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[170] Kenneth Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

Appendix A

Correctness Constructions

Here we show that a successful SLE and TLR execution is serializable—the execution corresponds to some legal serial execution. We define a legal execution as some serial execution and show that SLE (and thus TLR) will only commit serializable executions. Any non-serializable executions will be detected and rejected.

A.1 Maintaining serializability

Let CS = $\{CS_1,...,CS_n\}$ be a set of critical section executions[1] in the system. Further let $CSR_i = \{CSr_{i1},..., CSr_{ip}\}$ be the set of read operations in critical section CS_i and $CSW_i = \{CSw_{i1},...,CSw_{iq}\}$ be the set of write operations in critical section CS_i. Assume all operations, CSR_i and CSW_i, for a given critical section CS_i are ordered as per program order requirements. Let $R = \{r_1,...,r_j\}$ be normal individual read operations and $W = \{w_1,...,w_k\}$ be normal individual write operations interleaved in some order and belonging to regions outside critical sections (or in non-speculating critical sections).

If conflicting accesses occur, then the execution for these accesses is serialized by the coherence protocol—only one cache can have a writable copy of a cache block at any given time. Speculatively modified (and uncommitted) data by a thread in a critical section is never exposed to other threads. Further these values are only exposed at commit time. Since write serialization is maintained by our scheme, all values updated are exposed at the same time—the key is that only one processor can retire a store into the cache for a given address at any one time.

[1] These can be any sequence of memory operations identified to be critical sections. We do not specify protected access to these operations via the use of any lock—no restrictions are placed on it.

All CS are serializable. As discussed in Section 2.3.1.1, three conflict situations must be avoided to guarantee a serializable execution. Assume both CS_i and CS_j have committed and CS_i committed before CS_j. The three situations are:

1. *Write-read conflict*. Occurs if CS_i reads something CS_j wrote. This can never occur because uncommitted modified data is never exposed to other threads. Under SLE and TLR, this condition would be detected and CS_j would restart.

2. *Read-write conflict*. Occurs if CS_i overwrites what CS_j read and then CS_j reads it again. This cannot occur because when CS_i performs a conflicting write with CS_j, CS_j would misspeculate and restart and thus would re-execute the transaction. Thus the read-again problem will not occur.

3. *Write-write conflict*. Occurs if CS_i overwrites what CS_j wrote. This cannot occur because a conflict would have been detected earlier and one of the critical sections would have re-executed.

Thus, if CS_i commits before CS_j and they have accessed conflicting data sets, then all dependence arcs flow from CS_i to CS_j—there cannot be a cycle.

Each individual memory operation from R and W can be viewed as a critical section of unit size. Thus, they are a degenerate case of CS_i and the same conflict resolution schemes can be applied as above to guarantee a serializable execution.

Further, the implementation must guarantee that for a critical section CS_i, all dependences from memory operations $\{R, W\}_{pre\text{-}cs}$ prior to CS_i to $\{CSR_i, CSW_i\}$ are maintained and all dependences from $\{CSR_i, CSW_i\}$ to $\{R,W\}_{post\text{-}cs}$ operations after CS_i are maintained as per the underlying memory consistency model.

In other words,

$$\mathbf{\{R,W\}_{pre\text{-}cs} \rightarrow_{\mathit{po}} \{CSR_i, CSW_i\} \rightarrow_{\mathit{po}} \{R,W\}_{post\text{-}cs}}$$

where $\rightarrow_{po}$ is the program order chain.

Thus, $\{CSR_i, CSW_i\}$ is ordered atomically—it is serializable with respect to all other thread executions, and it is ordered with respect to operations before and after $\{CSR_i, CSW_i\}$.

Assume LR_i is a read (say, of the lock variable) and LW_{i1} and LW_{i2} are writes (say of the lock variable). Thus the ordering above can be represented as:

$$\mathbf{\{\{R,W\}_{pre\text{-}cs} \rightarrow_{\textit{po}} LR_i \rightarrow_{\textit{po}} LW_{i1}\} \rightarrow_{\textit{po}} \{CSR_i, CSW_i\} \rightarrow_{\textit{po}} \{LW_{i2} \rightarrow_{\textit{po}} \{R,W\}_{post\text{-}cs}\}}$$

We know {CSR$_i$, CSW$_i$} is serializable by SLE. Since CSR$_i$ and CSW$_i$ are read and write operations, we can rewrite the above as follows:

$$\mathbf{\{R,W\}_{pre\text{-}cs} \rightarrow_{po} \{LR_i \rightarrow_{p\ o}\ LW_{i1} \rightarrow_{po} \{CSR_i, CSW_i\} \rightarrow_{po} LW_{i2}\ \} \rightarrow_{po} \{R,W\}_{post\text{-}cs}}$$

Since {CSR$_i$, CSW$_i$} is serializable, we can rewrite the above again as follows:

$$\mathbf{\{R,W\}_{pre\text{-}cs} \rightarrow_{po} \{LR_i \rightarrow_{po} \{LW_{i1} \rightarrow_{po} CSR_i, CSW_i \rightarrow_{po} LW_{i2}\ \}\} \rightarrow_{po} \{R,W\}_{post\text{-}cs}}$$

Since LW$_{i1}$ immediately precedes {CSR$_i$, CSW$_i$}, and {CSR$_i$, CSW$_i$} immediately precedes LW$_{i2}$, conceptually the region of serializable execution now includes the outmost curly braces shown above. The above program ordering chain is maintained as per the single thread. Note, LR$_i$, LW$_{i1}$, and LW$_{i2}$ are merely notations but no difference exists between them and R$_i$ and W$_i$.

As seen from the outside of this thread, the entire sequence {$\mathbf{LR_i \rightarrow_{\textit{po}} \{LW_{i1} \rightarrow_{\textit{po}} CSR_i, CSW_i \rightarrow_{\textit{po}} LW_{i2}}$ }} is ordered either before any other {CSR$_j$, CSW$_j$} or {R} or {W} because it is serializable. We also know that LW$_{i1}$ and LW$_{i2}$ are cancelling stores—LW$_{i2}$ undoes the effects of LW$_{i1}$ thus leaving the architectural state unchanged. We also know that CSW$_i$ does not contain LW$_i$. Thus, to guarantee serializability, LW$_{i1}$ and LW$_{i2}$ can be removed from the operations set (for conflict detection) because they are restoring values. Alternatively, one way to view it is the sequence shown is executed, all addresses are pre-requested. Then the operations are performed and then the new values checked for the old values. If the two values are the same, then the new value need not be written to the old value. This is also the case with LW$_{i1}$ and LW$_{i2}$. Since the operations ({CSR$_i$, CSW$_i$}) between these two stores are indivisible (atomic), all operations from other processors are either ordered before or after the sequence.

SLE guarantees that any committed execution is serializable. Thus, the lock variable itself (i.e., LR$_i$) is also part of the CSR$_i$ set for tracking any writes to it and detecting such conflicts. This is for serializability with respect to other threads. It is however not added to the CSW$_i$ set because the write is never exposed to other threads in speculative mode. Since the two elided stores are silent, the architectural state prior to and after the execution remains the same.

TLR builds off SLE and use SLE to commit executions. Therefore, the correctness constructions for SLE also hold for TLR. The TLR proposal in this thesis is based on the wound-wait algorithm by Rosenkrantz et al. [144] and they showed wound-wait to be deadlock-free.

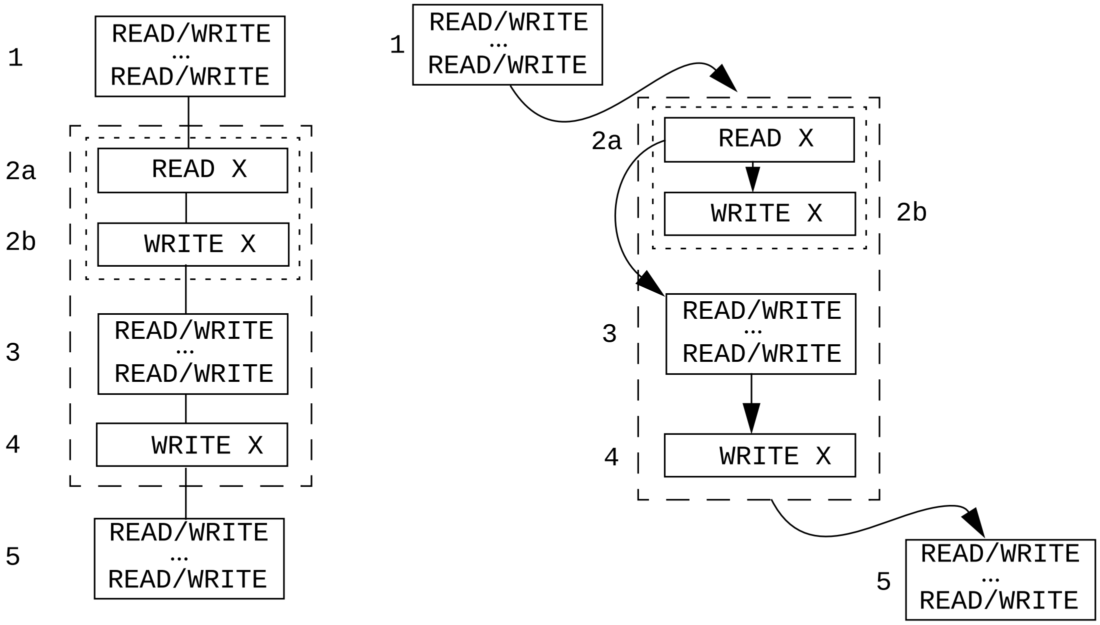


Figure A-1: Program order for memory operations from a single processor. *Memory operations are shown. The dashed box including blocks 2a, 2b, 3 and 4 signifies the atomicity of the region The dotted box around 2a and 2b signifies conceptually a read/write operation on the same address X. One legal ordering is shown on the right side.*

A.2 SLE and program order

This section revisits the above section and informally shows how program order is maintained with SLE. Figure A-1 shows the program order sequence for an instruction sequence. Blocks 2, 3, and 4 constitute the atomic region we are interested in. 2a and 2b correspond to the first store we wish to elide. Note, the operation is conceptually split in two parts: 2a and 2b. We assume an implicit read before the store is elided to ensure temporal silence. The single processor program order shown is maintained. Block 1 precedes blocks 2, 3, and 4 in the figure. Thus, all operations of 1 are assumed to have completed before 2, 3, and 4 are executed. Similarly, Blocks 2, 3, and 4 are assumed to have completed before block 5 executes. Note, the notion of completion does not imply serialization in physical time but merely the appearance.

In the program ordering shown, the atomic block comprising of 2, 3, and 4 is assumed to have semantics of a memory barrier in keeping with most atomic read-modify-write operations. This will be true for strong models such as sequential consistency and total store ordering. However, for weaker models such as release consistency, the presence of fence operations is used to enforce ordering. Thus, an alternate ordering chain may be to relax even the block 1 to block 2 ordering unless the atomic block has a fence instruction in which case the ordering must be maintained.