Tel-Aviv University
Raymond and Beverly Sackler Faculty of Exact Sciences
School of Computer Science

Techniques for Building Highly Concurrent Data Structures

by

Ori Shalev

The work on this thesis was carried out under the supervision of
Prof. Nir Shavit
Prof. Sivan Toledo

A thesis submitted
for the degree "Doctor of Philosophy"

Submitted to the senate of Tel-Aviv University
October, 2007

Acknowledgments

First of all, I thank my adviser, Nir Shavit, who has come to be a friend at least as much he was a supervising professor. Nir was always truly committed to bringing my research to its full potential, and responsible for many of the breakthroughs we had. I thank Nir and his wife Shafi for hosting me during some of my very effective visits in Cambridge.

I thank Mark Moir, Steve Heller, and Miriam Kadansky from Sun Microsystems labs in Massachusetts for giving me the opportunity to take part in a unique research group as a summer intern and later as a partial-time remote intern. Sun Microsystems, as well, had a significant part in the financial support of my research. I thank Dave Dice for being an extraordinary co-author and colleague, for his patience, diligence, and great ideas. I thank Michal Geva, Tiki Cohen, and Susan Marks from Sun's development center in Israel, who hosted me warmly during my remote internship at Sun.

I thank Yehuda Afek, who cared and helped me at Tel-Aviv university during the entire period. I thank Sivan Toledo who has volunteered to be my official adviser during Nir's sabbatical and really cared for my progress. I thank Pnina Neria-Barzilay for being a great teaching coordinator, who gave me the opportunity to teach interesting courses, and was always flexible enough for my needs.

Finally, I thank my loving wife Jenny who kept encouraging me to pursue this degree and supported me, including times in which I was away from home.

Abstract

In this dissertation, we present several novel techniques for concurrently accessing shared data structures in shared memory multi-processor/multicore machines. These techniques improve parallelism and thus system throughput by avoiding the use of coarse-grained locking. We begin by presenting a new design for a concurrent hash table based on the novel concept of a *split-ordered list*. The hash table algorithm is the first non-blocking algorithm efficiently supporting growing data sets. We then proceed to introduce three paradigms for general implementation of concurrent data structures.

The first two general synchronization schemes share a principle we call *log-synchronization*. These schemes aim to improve throughput by enabling read-only operations to be executed concurrently with modifying operations. A low-overhead execution of the common type of operations is possible due to duplication of shared data into active and shadow copies.

Finally, we introduce *Transactional Locking* (TL), a practical design approach for software transactional memory (STM) systems. We introduce the TL2 fine-grained locking scheme based on a global clock, a scheme that deals with many important practical aspects of STM implementation: low overhead, involunerability to inconsistent states, and compatibility with non-custom memory management.

A common feature of all the proposed designs is the emphasis on making them practical. All algorithms were evaluated empirically and shown to provide high throughput in a significant set of benchmarks.

Contents

Chapter 1

Introduction

1.1 General Background

1.1.1 Overview

In recent years, shared memory multiprocessors have gained popularity due to the increasing need for high-throughput computing in the software industry. Despite the distributed nature of the world-wide-web, many applications depend on a centralized entity, such as a database, for which a shared memory multiprocessor is often the most suitable platform. Utilizing multiprocessors to achieve high throughput is a non-trivial task. One must overcome several sources of bottlenecks in order to provide a scalable solution: the machine hardware resources, such as memory and computational units, communication resources such as bus traffic, and application bottlenecks.

While the effects of hardware-related bottlenecks can generally be reduced by physically enhancing the hardware, the communication- and application-related bottlenecks are inherent. The large number of processors imposes higher memory access latency, and sequentially executing programs (or parts of them) do not utilize the available computational resources. A dominating trend in research of high throughput programming on shared memory multiprocessors is designing specialized algorithms for concurrent access to shared data. These algorithms are usually designed to allow higher parallelism while keeping communication resources usage as low as possible.

When programming in a concurrent environment, one must be aware of the constantly changing state of the shared data. Converting a sequential algorithm to a concurrent one can be an extremely difficult task, as scheduling and cache coherency related factors are mostly transparent to the programmer, but can be crucial to the program's correctness and efficiency. There is often a tradeoff between the complexity of a program and the level of parallelism it provides: coarse-grained locking is simple and relatively safe to use, but it usually entails longer critical paths, therefore decreasing parallelism. Proving concurrent algorithms correct can be tedious and time consuming due to the non-deterministic nature of parallel execution.

Coarse-grained locking is by definition, in many cases, a bad choice for the synchronization building-block of concurrent applications. The reason is that

locks provide separation in time, but not in space. Using a small set of locks for large and complex data structures, limits the availability of large portions of the shared data. Threads that operate on different parts of the data structure should not block each other from making progress merely because they are performing a similar operation concurrently. Coarse-grained locks hardly consider the locality of operations, only their timing. By decreasing the granularity of locks (and increasing their number), for some data structures is it possible to reduce the amount of contention. However, fine-grained locking algorithms can be quite complex.

While some data structures, such as stacks and queues, are fundamentally centralized having a small and contended set of hot spots, other structures, such as search trees and hash tables are accessed in a more distributed manner. Using fine-grained locks wisely on these complex structures significantly improves overall throughput.

The non-blocking programming paradigm has emerged as a "cure" for several drawbacks of the lock-based approach. By definition, it avoids locking, utilizing atomic primitives such as CAS (compare-and-swap) or LL/SC (load-linked/store-conditional) instead. Non-blocking programs are free from deadlocks and also avoid scenarios where threads wait for a resource to be released. During the past decade, there has been a gradual relaxation in the constraints considered as basic for non-blocking execution. The original *wait-freedom* [35] property, requiring progress by all threads, has been relaxed to *lock-freedom*, requiring progress by *some* thread. The *obstruction freedom* progress property has been proposed recently [38], allowing livelocks and depending on separate contention management mechanisms to guarantee progress. The research and practitioners communities have lowered the threshold of "accepted" level of non-blockingness for good reasons: it enabled simpler programs by reducing the overhead costs of progress management, improving overall performance.

One of the key properties of non-blocking synchronization is the ability to sustain long and even permanent failures without significant delays. However, to provide such ability, the process of applying complex modifications to data structures must be "documented" by the operating thread in case this thread is delayed or fails. Making the details of its operation visible to other threads adds to the overhead costs of non-blocking implementations. Recently, it has been claimed [20, 16] that the fine-grained locking approach is a decent alterna-

tive to the non-blocking one, as it lacks the kind of inherent overhead described above. Nevertheless, the possibility of deadlocks must be addressed, for example, by applying deadlock detection and recovery schemes. Fine-grained locking is also sensitive to permanent processor failure.

In the past, the research focus used to be on providing solutions for a rather small set of specific data structures – stacks, linked lists, queues, hash tables, and search trees. However, researchers are now understanding that a generic concurrent programming framework is essential. Parallelism can become ubiquitous only when programmers of ordinary skills are capable of writing efficient and safe concurrent applications. The *transactional memory* (TM) paradigm [39] has emerged as an attempt to provide easy-to-use and efficient generic framework for concurrent programming. Hardware [39, 77, 4, 27], software [20, 34, 42, 44, 62, 14, 82, 88, 93], and hardware/software hybrids [5, 14, 51] transactional memory schemes have been proposed. TM gives the programmer means to execute blocks of code transactionally, that is, with a guarantee that the effect of the execution is as if that code was processed in isolation. The concept of TM allows execution parallelism among threads that do not collide in time *and* space, i.e. as long as threads handle disjoint sets of memory locations they do not affect each other. Moreover, in most TM implementations, threads overlapping in their read-sets (the set of addresses they read from during the transaction) can also make progress in parallel. Most TM implementations allow easy transformation of existing coarse-grained locking code to transactional code, and by that, reduce the level of expertise required from programmers of concurrency-rich applications.

The interface provided by the transactional memory abstraction is fundamentally incompatible with the capabilities of conventional systems hardware. It is physically impossible to efficiently perform an unbounded number of atomic memory updates, especially given today's cache coherence protocols. Therefore, some TM schemes involve hardware enhancements (HTM) [39, 77, 4, 27], while others do software emulations (STM) [20, 34, 42, 44, 62, 14, 82, 88, 93]. To this time, hardware TMs have never been fully integrated in an actual product and are still far from being available for the developer community. In contrast, several STM implementations have been publicly released [43, 32, 42]. Generally, hardware implementations struggle with supporting transactions that are unbounded in time and size given limited hardware resources, and STMs must

cope with the unavoidable overhead of execution in software.

Until recently, the majority of proposed STM solutions [44, 21, 33, 62] took the non-blocking approach, attempting to achieve the wide set of goals consisting of both freedom from deadlocks and long delays, and general applicability to any custom data structure. Trying to achieve this set of goals came at the cost of performance: the overhead of the proposed mechanisms was significant in respect to the analogous sequential algorithms. Therefore, some recent papers [20, 82, 16, 15] chose to take the fine-grained locking approach to implementing software transactions. This approach is theoretically more sensitive to scheduling misbehaviors but it has the potential to have smaller overhead costs due to the simplicity of using locks.

1.1.2 Hardware Support for Synchronization

A basic building-block of the non-blocking synchronization paradigm is a set of atomic primitives, which is at least partially supported on most modern architectures. The vast majority of non-blocking algorithms make use of either the *Compare-And-Swap* (CAS) instruction or the pair *Load-Linked/Store-Conditional* (LL/SC). CAS receives an address, an expected present value, and a new value. The new value is atomically set to the address if the actual present value is equal to the expected one. The LL/SC primitive instruction works in a fundamentally different way: the SC instruction fails to store a value into a memory location if that location has been modified since the last time the operating thread executed LL on it.

Unlike other primitives such as *Fetch-And-Increment*, CAS and LL/SC are *optimistic*: they are not guaranteed to succeed in concurrent environments because other threads can modify the target address. However, these primitives can guarantee sufficient progress for some algorithms where failure only occurs as a result of another thread's success. This type of progress is acceptable as *lock-free*, since at least one thread is guaranteed to make progress in any given time. Both CAS and LL/SC have weak variations that allow spurious failures, where in the case of LL/SC, the weaker type called *Restricted LL/SC* is more common in existing architectures supporting LL/SC instructions. The reason for that is the state-oriented characteristics of LL/SC: it is non-trivial to maintain state information for an unbounded number of LL-ed memory locations.

On the other hand, CAS has a stateless implementation in hardware, and because of that, it is sensitive to execution scenarios collectively referred to by the term ABA. In these scenarios, a memory word changes from value A to B and then back to A, while a thread attempting to set that word using CAS is not aware of these two changes. The ABA problem is critical especially when handling pointers to dynamically allocated objects that can be recycled into addresses being referenced by other threads.

The commonly available atomic primitives are all capable of handling single words. However, many data structures are complex, consisting of multiple indirection levels. Performing the series of memory accesses of a single high-level operation cannot be performed by executing accesses one after another, as intermediate states are publicly exposed, potentially leading to inconsistencies when the executions are interleaved. Therefore, the task of designing non-blocking algorithms for accessing complex data structures must be carried out with extreme care. Some data structures, such as red-black trees, have no non-blocking implementation with reasonably low overhead.

One way to reduce the algorithmic complexity of non-trivial data structure access is by enhancing the hardware capabilities so that they support multi-word atomic synchronization primitives. Unfortunately, memory systems have historically evolved into a form that does not allow multi-word primitives to be integrated efficiently. The presence of cache and cache-coherency protocols complicates the hardware requirements for supporting such primitives. Nevertheless, a *Double-Compare-And-Swap* (DCAS) operation, performing a CAS on two locations atomically, has actually been implemented in the Motorola 68K series processors, whose production has terminated in year 2000. Although research has been conducted on the strength of DCAS [24, 25, 2, 7], it has never been accepted as a viable basis for wide enough range of problems. On the thoretical side, it has been proven by Attiya and Dagan [7] that there exists a problem solvable in $O(1)$ steps using DCAS equivalent primitives, which requires $\Omega(\log\log * n)$ steps using CAS equivalent ones.

In 1993, Herlihy and Moss [39] proposed the idea of a hardware transactional memory (HTM) and a specific implementation based on the cache coherency infrastructure in bus based shared memory multiprocessors. The proposed HTM architecture allowed optimistic atomic execution of bounded size transactions, however, was never implemented, mainly due to the transaction size

limitation. Since then, several hardware-based solutions [4, 77, 9] have been suggested as a way to cope with the transaction size limits and other drawbacks of TM systems such as support for nesting and I/O operations within transactions. Additionally, two approaches for hybrid software/hardware TM have become public [14, 51].

All hardware-implemented transactional memory schemes involve expanding the machine instruction set with new instructions for transaction manipulation. These instructions are either special loads and stores used within transactions or mode changing instructions, from non-transactional to transactional mode and vice versa. The hardware constructs for supporting transactions are usually additional per-processor associative memory units used for buffering the transaction's temporary data. The transaction size limit in HTMs originates from the bounded size of that additional cache memory, while the limit on transaction duration is in many cases a result of the weak persistence of the buffered data over context switches and I/O operations.

In the next three sections we will survey the specific research background of the results in this thesis. We start with concurrent hash tables.

1.1.3 Concurrent Hash Tables

There are several lock-based concurrent hash table implementations in the literature. In the early eighties, Ellis proposed an extendible concurrent hash table for distributed data based on a two level locking scheme, first locking a table directory and then the individual buckets [18, 19]. Michael [67] has recently shown that on shared memory multiprocessors, simple algorithms using a reader-writer lock [66] per bucket have reasonable performance for non-extendible tables. However, to resize one would have to hold the locks on all buckets simultaneously, leading to significant overheads. A recent algorithm by Lea [54], proposed for *java.util.concurrent*, the Java™ Concurrency Package, is probably the most efficient known concurrent extensible hash algorithm. It is based on a more sophisticated locking scheme that involves a small number of high level locks rather than a lock per bucket, and allows concurrent searches while resizing the table, but not concurrent inserts or deletes. In general, lock-based hash-table algorithms are expected to suffer from the typical drawbacks of blocking synchronization: deadlocks, long delays, and priority inversions

[24]. These drawbacks become more acute when performing a *resize* operation, an elaborate "global" process of redistributing the elements in all the hash table's buckets among newly added buckets. Designing a lock-free extendible hash table is thus a matter of both practical and theoretical interest.

Michael, in [67], builds on the work of Harris [30] to provide an effective compare-and-swap (CAS) based lock-free linked-list algorithm (which we will elaborate upon in the following section). He then uses this algorithm to design a lock-free hash structure: a fixed size array of hash buckets with lock-free insertion and deletion into each. He presents empirical evidence that shows a significant advantage of this hash structure over lock-based implementations in multiprogrammed environments. However, this structure is not extendible: if the number of elements grows beyond the predetermined size, the time complexity of operations will no longer be constant.

As part of his "two-handed emulation" approach, Greenwald [25] provides a lock-free hash table that can be resized based on a double-compare-and-swap (DCAS) operation. However, DCAS, an operation that performs a CAS atomically on two non-adjacent memory locations, is not available on current architectures. Moreover, although Greenwald's hash table is extendible, it is not a true extendible hash table. The average number of steps per operation is not constant: it involves a helping scheme where that under certain scheduling scenario would lead to a time complexity linearly dependent on the number of processes.

In [76], Purcell and Harris presented the first non-blocking open-addressing hash table scheme. By taking the open-addressing approach, the obstruction-free implementation is not dependent on memory management. However, their table cannot be dynamically resized.

Concurrently and independently of our work on hash tables, Gao et al. [23] have developed an extendible and "almost wait-free" hashing algorithm based on an open addressing hashing scheme and using only CAS operations. Their algorithm maintains the dynamic size by periodically switching to a global resize state in which multiple processes collectively perform the migration of items to new buckets. They suggest performing migration using a write-all algorithm [48]. Theoretically, each operation in their algorithm requires more than constant time on average because of the complexity of performing the

write-all [48], and so it is not a true extendible hash-table. However, the non-constant factor is small, and the performance of their algorithm in practice will depend on the real-world performance of algorithms for the write-all problem [48, 50].

1.1.4 Programming with Transactions

The *transactional memory* programming paradigm [46] is gaining momentum as an approach of choice for replacing locks in concurrent programming. Combining sequences of concurrent operations into atomic transactions seems to promise a great reduction in the complexity of both programming and verification, by making parts of the code appear to be sequential without the need to use locks. The "transactional manifesto" is that transactions will remove from the programmer the burden of figuring out the interaction among concurrent operations that happen to overlap or modify the same locations in memory. Transactions that do not overlap will run uninterrupted in parallel, and those that do will be aborted and retried without the programmer having to worry about issues such as deadlocks. The current implementations of transactional memory are purely software based (software transactional memories (STM)), but down the road, combinations of hardware and software will hopefully offer the simplified reasoning of transactions with a low computational overhead.

However, the transactional approach is not a panacea, and like locks, suffers from several notable drawbacks. Transactions simplify but do not eliminate the programmer's need to reason about concurrency, and the decision of which instructions to include in a transaction are explicitly in the hands of the programmer. This leaves the programmer with a tradeoff similar to that of locks, between the size of the transactions used and the program's performance: any reduction of transaction size implies added complexity in code design and verification. Concurrently executing transactions also introduce numerous programming language issues such as nesting, I/O, exceptions [31], and early release [44], that offer to complicate transactional programming even if it is made efficient. Finally, though great progress is being made in this area, transactions implemented in software (i.e. STMs) are at this point less efficient, in some cases they can be even slower than monitor locks, because of the overhead of

the conflict detection and copy-back mechanisms used in implementing them [29, 44, 61, 62, 14, 88, 93].

1.1.5 Software Transactional Memory

Implementing transactional memory in software has two major advantages: one is the availability of software libraries in contrast to architectural modifications that are also extremely expensive to manufacture even as prototypes, and the other is the virtually unlimited resources available, namely memory. However, in all STMs, there is a price in latency of load and store instructions. While memory access instructions used in HTMs are in most cases identical in latency to classic loads and stores, when emulated in software, transactional reads and writes involve additional checks and address calculation for reaching the actual data. When reading from a memory location, it is essential to identify whether that location is currently participating in an active transaction, and in some cases, follow pointers to reach the actual value. An additional source of overhead is the need to traverse the read-set and/or write-set of a transaction on commit time, a task performed in parallel and as a part of the cache protocol in some hardware alternatives.

Shavit and Touitou proposed the first STM implementation in [87], which supported static transactions, i.e. ones where the addresses accessed by a transaction are known in advance. Their implementation is lock-free, as aborting threads help the transaction that caused them to abort.

In [6], Anderson and Moir implemented lock-free multi-word CAS (MWCAS), a variation of which Moir used in [70] to build a lock-free dynamic STM. In this construction, the memory is divided into B blocks which threads lock during the execution of transactions. However, since each transaction involves applying a B-word MWCAS, there is an inconvenient tradeoff between efficiency and parallelism. Moir also describes a wait-free variation by applying a different helping scheme.

Herlihy et al. introduced a dynamic obstruction-free STM (DSTM) [44]. Its main principle is that transactional objects point to transactional records, that in turn, point to the object's versions before and after the transaction. Transactions are aborted and committed by setting a status word pointed from each

transactional record. These levels of indirection enabled an elegant scheme that supports parallelism of transactions that do not overlap in memory access, however, the overhead of accessing memory and validating and committing transactions is relatively high. Concurrently, Harris and Fraser [33] developed an STM with a different approach, WSTM, in which memory words are associated with transactions using a map from heap addresses to transactional descriptors. The set of addresses participating in a transaction is acquired only at commit-time, minimizing the chances for collisions with other transactions. This approach requires reading-only transactions to be visible to possibly colliding modifying transactions by writing to shared memory even when the actual data is unchanged. In his Ph.D. dissertation [21] Fraser described OSTM, an STM implementation similar to WSTM, where transactions handle objects rather than individual memory words.

Marathe et al. [62] proposed Adaptive STM (ASTM), a combination of DSTM and OSTM, where an object's association with transactions alternates between DSTM-like to OSTM-like depending on the application. Recently, Marathe et al. [63] introduced the Rochester STM (RSTM), an improved STM implementation in which the data is co-located with transactional meta-data, reducing the overhead cost of transactional operations.

In [20], Ennals claimed that obstruction-freedom is not an essential property of STMs, and by dropping this requirement STMs can be significantly faster. Ennals proposed a lock-based STM and demonstrated performance improvements. Both Ennals' implementation and the one Saha et al. [82] developed for their Multi-Core RunTime (McRT) STM-supporting environment, use versioned locks for transactional objects, committing and validating transactions by comparing the versions of read-set addresses while write-set ones are locked.

The integration of a global clock in STM implementation as an aid for conflict detection was developed by Riegel et al. [80] independently of our work on Transactional Locking II [15]. In their work, the global clock is used as a part of a tight system for bookkeeping transactional operations, which minimizes transaction aborts.

1.2 The Contributions of this Thesis

The main goal of this thesis is to present a collection of approaches that will help with the adoption of advanced programming techniques such as lock-free data structures and transactional programming. In order for these advanced tools to be accepted by the developer communities, they must (1) offer competitive performance in the sequential case as well as in the parallel one, (2) not require uncommon programming skills, and (3) be compatible to existing software and hardware platforms. These goals are essentially the motivation of the work we present in this dissertation.

Both the non-blocking synchronization and transactional memory paradigms started off quite far from the above goals, and since the time they were initially proposed, have made some progress towards them. Researchers taking the non-blocking approach for concurrency have been struggling with data structures such as hash tables and binary trees, which are substantially more complex to access than simple stacks and queues. This was among the reasons that led the research community into more general solutions in the form of transactional memory.

In this dissertation, we begin by presenting an efficient implementation of a non-blocking extendible hash table – an *ad-hoc* concurrent algorithm based on the novel concept of *split-ordering*, making it competitive to lock-based alternatives.

Split-ordering was conceived in order to overcome the difficulty of atomically moving items from old to new buckets when extending hash tables. Metaphorically, with split-ordering we distribute hash table buckets among the elements instead of the other way around. Unlike moving an item, the operation of directing a bucket pointer can be done in a single CAS operation, and since items are not moved, they can never be lost when threads are preempted. However, to make this approach work, one must be able to keep the items in the list sorted in such a way that any bucket's sublist can be "split" by directing a new bucket pointer within it. This operation must be recursively repeatable, as every split bucket may be split again and again as the hash table grows. To achieve this goal we introduce *recursive split-ordering*, a new ordering on keys that keeps items in a given bucket adjacent in the list throughout the repeated

splitting process.

In Chapter 2 we describe the split-ordered hash table, which was initially presented in [84], and later published it in [86].

The experimental work we have done on high throughput hash tables led us to observations on how to design better software transactional memory schemes, which we then developed into several ideas for practical general concurrent access.

One approach, *log-synchronization*, separates modifying operations from reading-only ones, which are much more common in most applications. Log-synchronization is based on a global log of high-level *modifying* operations that serializes all concurrently executing operations according to the order in which they were initiated, and allows for almost overhead-free concurrent read-only operations. We present two variations of log-synchronization.

In the first, *Predictive Log-Synchronization* (PLS), the execution is non-blocking as threads are capable of predicting the result of a yet-to-complete operation using operation arguments stored in the log. In PLS, the shared data structure is duplicated, protected by a high level lock, and appended with a special *log* of high level operations. A thread owning the lock performs all data structure modifications logged by others on one copy, allowing all threads to concurrently read the other unmodified copy, and switches copies before releasing the lock. PLS is non-blocking since threads failing to acquire lock ownership make progress by inspecting the log and *predicting* the result of their own operation. Concurrently, all read-only operations can access the unmodified copy of the data structure in a non-blocking manner and with virtually no overhead.

In Chapter 3 we present the PLS technique, based on our conference paper [85].

The second log-synchronization scheme is brought in Chapter 4. In *address-based* log-synchronization, the log consists of the actual memory addresses written to and read from, and modifying operations (transactions) are executed sequentially by applying memory stores from the log. The address-based log-synchronizer detects conflicts on the address level by having modifying threads perform "virtual execution" on the data structure, during which they only record their memory load and store operations. All stores are then actually performed by a single lock-owning thread while reading-only operations

can make progress concurrently. This approach is highly optimized for workloads dominated by reading-only operations. To allow concurrent reads and writes, the shared-access fields of objects are individually duplicated in memory, unlike PLS, where the entire data structure is duplicated.

In an effort to improve the scalability of log-synchronization, while attempting to convert the log into a simple version counter, we came up with a fine-grained lock-based software transactional memory implementation, the *Transactional Locking II* (TL2). By utilizing a global clock, we managed to detect transaction conflicts and avoid inconsistent state views, providing an extremely low-overhead STM implementation. In TL2, transactions begin by sampling the global clock, and then performing a speculative execution, during which all accessed locations are verified against the sampled start time. To commit, threads lock the memory locations in their write-set, acquire a unique timestamp, revalidate their read set, and release the locks after writing the new values and timestamp.

Unlike all previously suggested STMs, TL2 avoids viewing inconsistent states and does it cleanly and without incurring significant overhead costs. TL2 enables very efficient reading-only transactions by not requiring the thread to post-validate its read-set. Reading-only transactions can simply complete successfully when the executing thread reaches the end of the transaction's code. Also unlike all other STMs, the TL2 algorithm allows allocated objects to be used in both transactional and non-transactional domains.

In Chapter 5 we bring a complete description of TL2, based on [15].

Chapter 2

Split-Ordered Lists: Non-Blocking Hash Tables

2.1 Introduction

Hash tables, and specifically extendible hash tables, serve as a key building block of many high performance systems. A typical extendible hash table is a continuously resized array of buckets, each holding an expected constant number of elements, and thus requiring an expected constant time for insert, delete and find operations [12]. The cost of resizing, the redistribution of items between old and new buckets, is amortized over all table operations, thus keeping the average complexity of any one operation constant. As this is an extendible hash table, "resizing" means extending the table. It is interesting to note, as argued elsewhere [49, 55], that many of the standard concurrent applications using hash tables require tables to only increase in size."

We are concerned in implementing the hash table data structure on multiprocessor machines, where efficient synchronization of concurrent access to data structures is essential. Lock-free algorithms have been proposed in the past as an appealing alternative to lock-based schemes, as they utilize strong primitives such as CAS (*compare-and-swap*) to achieve fine grained synchronization. However, lock-free algorithms typically require greater design efforts, being conceptually more complex.

In this chapter, we present the first lock-free extendible hash table that works on current architectures, that is, uses only loads, stores and CAS (or LL/SC [70]) operations. In a manner similar to sequential linear hashing [57] and fitting real-time[1] applications, resizing costs are split incrementally to achieve expected $O(1)$ operations per insert, delete and find. The proposed algorithm is simple to implement, leading us to hope it will be of interest to practitioners as well as researchers. As we explain shortly, it is based on a novel *recursively split-ordered* list structure. Our empirical testing shows that in a concurrent environment, even without multiprogramming, our lock-free algorithm performs as well as the most efficient known lock-based extendible hash-table algorithm due to Lea [54], and in high load cases it significantly outperforms it.

[1] In this chapter, by *real-time* we mean *soft real-time* [10], where some flexibility on the real-time requirements is allowed

2.1.1 The Lock-free Resizing Problem

What is it that makes lock-free extendible hashing hard to achieve? The core problem is that even if individual buckets are lock-free, when resizing the table, several items from each of the "old" buckets must be relocated to a bucket among "new" ones. However, in a single CAS operation, it is far from straightforward to atomically move even a single item, as this requires one to remove the item from one linked list and insert it in another. If this move is not done atomically, elements might be lost, or to prevent loss, will have to be replicated, introducing the overhead of "replication management". The lock-free techniques for providing the broader atomicity required to overcome these difficulties imply that processes will have to "help" others complete their operations. Unfortunately, "helping" requires processes to store state and repeatedly monitor other processes' progress, leading to redundancies and overheads that are unacceptable if one wants to maintain the constant time performance of hashing algorithms.

2.1.2 Split-Ordered Lists

To implement our algorithm, we thus had to overcome the difficulty of atomically moving items from old to new buckets when resizing. To do so, we decided to, metaphorically speaking, flip the linear hashing algorithm on its head: our algorithm *will not move the items among the buckets*, rather, it *will move the buckets among the items*. More specifically, as shown in Figure 2.1, the algorithm keeps all the items in one lock-free linked list, and gradually assigns the bucket pointers to the places in the list where a sublist of "correct" items can be found. A bucket is initialized upon first access by assigning it to a new "dummy" node (dashed contour) in the list, preceding all items that should be in that bucket. A newly created bucket splits an older bucket's chain, reducing the access cost to its items. Our table uses a modulo 2^i hash (there are known techniques for "pre-hashing" before a modulo 2^i hash to overcome possible binary correlations among values [54]). The table starts at size 2 and repeatedly doubles in size.

Unlike moving an item, the operation of directing a bucket pointer can be done in a single CAS operation, and since items are not moved, they are never "lost".

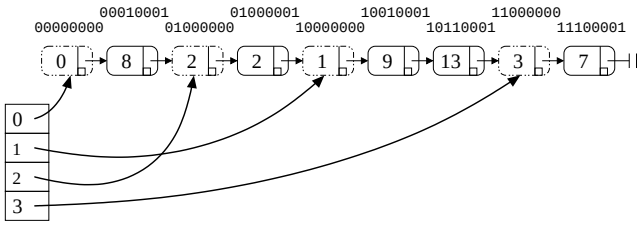


Figure 2.1: A Split-Ordered Hash Table

However, to make this approach work, one must be able to keep the items in the list sorted in such a way that any bucket's sublist can be "split" by directing a new bucket pointer within it. This operation must be recursively repeatable, as every split bucket may be split again and again as the hash table grows. To achieve this goal we introduced *recursive split-ordering*, a new ordering on keys that keeps items in a given bucket adjacent in the list throughout the repeated splitting process.

Apparently, recursive split-ordering can be achieved by simple *binary reversal*: reversing the bits of the hash key so that the new key's most significant bits (MSB) are those that were originally its least significant. As detailed below and in the next section, some additional bit-wise modifications must be made to make things work properly. In Figure 2.1 the split-order key values are written above the nodes (the reader should disregard the rightmost binary digit at this point). For instance, the split-order value of 3 is the bit-reverse of its binary representation, which is 11000000. The dashed-line nodes are the special dummy nodes corresponding to buckets with original keys that are 0,1,2, and 3 modulo 4. The split-order keys of regular (non-dashed) nodes are exactly the bit-reverse image of the original keys after turning on their MSB (in the example we used 8-bit words). For example, items 9 and 13 are in the "1 mod 4" bucket, which can be recursively split in two by inserting a new node between them.

To *insert* (respectively *delete* or *find*) an item in the hash table, hash its key to the appropriate bucket using recursive split-ordering, follow the pointer to the appropriate location in the sorted items list, and traverse the list until the key's proper location in the split-ordering (respectively until the key or a key indicating the item is not in the list) is found. The solution depends on the property

that the items' position is "encoded" in their binary representation, and therefore cannot be generalized to bases other than 2.

As we show, because of the combinatorial structure induced by the split-ordering, this will require traversal of no more than an expected constant number of items. A detailed proof appears in Section 2.3.

We note that our design is modular: to implement the ordered items list, one can use one of several non-blocking list-based set algorithms in the literature. Potential candidates are the lock-free algorithms of Harris [30] or Michael [67], or the obstruction-free algorithms of Valois[2][91] or Luchangco et al. [59]. We chose to base our presentation on the algorithm of Michael [67], an extension of the Harris algorithm [30] that fits well with memory management schemes [37, 68] and performs well in practice.

2.1.3 Complexity

When analyzing the complexity of concurrent hashing schemes, there are two adversaries to consider: one controlling the distribution of item keys, the other controlling the scheduling of thread operations. The former appears in all hash table algorithms, sequential or concurrent, while the latter is a direct result of the introduction of concurrency. We use the term *expected time* to refer to the expected number of machine instructions per operation in the worst case scheduling scenario, assuming (as is standard in the literature [12]) a hash function of uniform distribution. We use the term *average time* to refer to the number of machine instructions per operation averaged over all executions, also assuming a uniform hash function. It follows that constant expected time implies constant average time.

As we show in Section 2.3, if we make the standard assumption of a hash function with a uniform distribution, then under any scheduling adversary our new algorithm provides a lock-free extendible hash table with $O(1)$ average cost per operation.

The complexity improves to expected constant time if we assume a *constant extendibility rate*, meaning that the table is never extended (doubled in size)

[2]Valois' algorithm was labeled "lock-free" by mistake. It is livelock-prone.

a non-constant number of times while a thread is delayed by the scheduler. Constant expected time is an improvement over average expected time since it means that given a good hash function, the adversary cannot cause any single operation to take more than a constant number of steps.

One feature in which the new algorithm is similar in flavor to sequential linear hashing algorithms [57] (in contrast to all the above algorithms [23, 25, 54]) is that resizing is done incrementally and only bad distributions (ones that have very low probability given a uniform hash function) or extreme scheduling scenarios can cause the cost of an operation to exceed constant time. This possibly makes the algorithm a better fit for soft real-time applications [10] where relaxable timing deadlines need to be met.

2.1.4 Performance

We tested our new *split-ordered list* hash algorithm against the most-efficient known lock-based implementation due to Lea [54]. We created an optimized `C++` based version of the algorithm and compared it to split-ordered lists using a collection of tests executed on a 72-node shared memory machine. We present experiments in Section 2.4 which show that split-ordered lists perform as well as Lea's algorithms, even in non-multiprogrammed cases, although lock-free algorithms are expected to benefits systems mainly in multiprogrammed environments. Under high loads they significantly outperform Lea's algorithm, exhibiting up to four times higher throughput. They also exhibit greater robustness, for example in experiments where the hash function is biased to create non-uniform distributions.

The remainder of this chapter is organized as follows. In the next section we describe the background and the new algorithm in depth. In Section 2.3 we present the full correctness proof. In Section 2.4 the empirical results are presented and discussed.

2.2 The Algorithm In Detail

Our hash table data structure consists of two interconnected sub-structures (see Figure 2.1): a linked list of nodes containing the stored items and keys, and an expanding array of pointers into the list. The array entries are the logical "buckets" typical of most hash tables. Any item in the hash table can be reached by traversing down the list from its head, while the bucket pointers provide shortcuts into the list in order to minimize the search cost per item.

The main difficulty in maintaining this structure is in managing the continuous coverage of the full length of the list by bucket pointers as the number of items in the list grows. The distribution of bucket pointers among the list items must remain dense enough to allow constant time access to any item. Therefore, new buckets need to be created and assigned to sparsely covered regions in the list.

The bucket array initially has size 2, and is doubled every time the number of items in the table exceeds $size \cdot L$, where L is a small integer denoting the *load factor*, the maximum number of items one would expect to find in each logical bucket of the hash table. The initial state of all buckets is *uninitialized*, except for the bucket of index 0, which points to an empty list, and is effectively the head pointer of the main list structure. Each bucket goes through an initialization procedure when first accessed, after which it points to some node in the list.

When an item of key k is inserted, deleted, or searched for in the table, a hash function modulo the table size is used, i.e. the bucket chosen for item k is $k \ mod \ size$. The table size is always equal to some power 2^i, $i \geq 1$, so that the bucket index is exactly the integer represented by the key's i least significant bits (LSBs). The hash function's dependency on the table *size* makes it necessary to take special care as this size changes: an item that was inserted to the hash table's list before the resize must be accessible, after the resize, from both the buckets it already belonged to and from the new bucket it will logically belong to given the new hash function.

2.2.1 Recursive split-ordering

The combination of a modulo-size hash function and a 2^i table size is not new. It was the basis of the well known sequential extensible Linear Hashing scheme

proposed by Litwin [57], was the basis of the two-level locking hash scheme of Ellis [18], and was recently used by Lea in his concurrent extensible hashing scheme [54]. The novelty here is that we use it as a basis for a combinatorial structure that allows us to repeatedly "split" all the items among the buckets without actually changing their position in the main list.

When the table size is 2^i, a logical table bucket b contains items whose keys k maintain $k \ mod \ 2^i = b$. When the size becomes 2^{i+1}, the items of this bucket are split into two buckets: some remain in the bucket b, and others, for which $k \ mod \ 2^{i+1} = b + 2^i$, migrate to the bucket $b + 2^i$. If these two groups of items were to be positioned one after the other in the list, splitting the bucket b would be achieved by simply pointing bucket $b + 2^i$ after the first group of items and before the second. Such a manipulation would keep the items of the second group accessible from bucket b as desired.

Looking at their keys, the items in the two groups are differentiated by the i'th binary digit (counting from right, starting at 0) of their items' key: those with 0 belong to the first group, and those with 1 to the second. The next table doubling will cause each of these groups to split again into two groups differentiated by bit $i + 1$, and so on. For example, the elements 9 ($1001_{(2)}$) and 13 ($1101_{(2)}$) share the same two least significant bits (01). When the table size is 2^2, they are both in the same bucket, but when it grows to 2^3, having a different third bit will cause to to be separated. This process induces *recursive split-ordering*, a complete order on keys, capturing how they will be repeatedly split among logical buckets. Given a key, its order is completely defined by its bit-reversed value.

Let us now return to the main picture: an exponentially growing array of (possibly uninitialized) buckets maps to a linked list ordered by the split-order values of inserted items' keys, values that are derived by reversing the bits of the original keys. Buckets are initialized when they are accessed for the first time. List operations such as `insert`, `delete` or `find` are implemented via a linearizable lock-free linked list algorithm. However, having additional references to nodes from the bucket array introduces a new difficulty: it is non-trivial to manage deletion of nodes pointed to by bucket pointers. Our solution is to add an auxiliary dummy node per bucket, preceding the first item of the bucket, and to have the bucket pointer point to this dummy node. The dummy nodes are not deleted, which helps keep things simple.

```
so_key_t so_regularkey(key_t key) {
   return REVERSE(key OR 0x8000...0000);
}

so_key_t so_dummykey(key_t key) {
   return REVERSE(key);
}
```

Figure 2.2: **The Split-Ordering Transformation**. The function `so_regularkey` computes the split-order value for regular nodes, where the MSB is set before reversing the bits. The split-order value of dummy nodes is the exact bit reverse of the key.

In more detail, when the table size is 2^{i+1}, the first time bucket $b + 2^i$ is accessed, a dummy node is created, holding the key $b + 2^i$. This node is inserted to the list via bucket b, the *parent* bucket of $b + 2^i$. Under split-ordering, $b + 2^i$ precedes all keys of bucket $b + 2^i$, since those keys must end with $i + 1$ bits forming the value $b + 2^i$. This value also succeeds all the keys of bucket b that do not belong to $b + 2^i$: they have identical i LSBs, but their bit numbered i is "0". Therefore, the new dummy node is positioned in the exact location in the list that separates the items that belong to the new bucket from other items of bucket b. In the case where the parent bucket b is uninitialized, we apply the initialization procedure on it recursively before inserting the dummy node. In order to distinguish dummy keys from regular ones we set the most significant bit of regular keys to "1", and leave the dummy keys with "0" at the MSB. Figure 2.2 defines the complete split-ordering transformation using the functions `so_regularkey` and `so_dummykey`. The former, reverses the bits after turning on the MSB, and the latter simply performs the bit reversal[3].

Figure 2.3 describes a bucket initialization caused by an insertion of a new key to the set. The insertion of key 10 is invoked when the table size is 4 and buckets 0,1 and 3 are already initialized.

Since the bucket array is growing, it is not guaranteed that the parent bucket of an uninitialized bucket is initialized. In this case, the parent has to be initialized (recursively) before proceeding. Though the total complexity in such a series of recursive calls is potentially logarithmic, our algorithm still works. This is because given a uniform distribution of items, the chances of a logarithmic-size

[3]An efficient implementation of the `REVERSE` function utilizes a 2^8 or 2^{16} lookup table holding the bit-reversed values of $[0..2^8 - 1]$ or $[0..2^{16} - 1]$ respectively.

series of recursive initialization calls are low, and in fact, the expected length of such a bad sequence of parent initializations is constant.

2.2.2 The Continuously Growing Table

We can now complete the presentation of our algorithm. We use the lock-free ordered linked-list algorithm of Michael [67] to maintain the main linked list with items ordered based on the split-ordered keys. This algorithm is an improved variant, including improved memory management, of an algorithm by Harris [30]. Our presentation will not discuss the various memory reclamation options of such linked-list schemes, and we refer the interested reader to [30, 37, 67, 68]. To keep our presentation self contained, we provide the code of Michael's linked list algorithm below. This implementation is linearizable, implying that each of these operations can be viewed as happening atomically at some point within its execution interval.

Our algorithm decides to double the table size based on the average bucket load. This load is determined by maintaining a shared counter that tracks the number of items in the table. The final detail we need to deal with is how the array of buckets is repeatedly extended. To simplify the presentation, we keep the table of buckets in one continuous memory segment as depicted in Figure 2.4. This approach is somewhat impractical, since table doubling requires one process to reallocate a very large memory segment while other processes may be waiting. The practical version of this algorithm, which we used for performance testing, actually employs an additional level of indirection in accessing buckets: a main array points to segments of buckets, each of which is a bucket array. A segment is allocated only upon the first access to some bucket within it. The code for this dynamic allocation scheme appears in Section 2.2.4.

2.2.3 The Code

Lock-Free Linked Lists

Before we provide the code for our algorithm, we give here the code for the lock-free linked list implementation by Michael [67]. Our hash table code uses

the `list_insert`, `list_delete`, and `list_find` subroutines of Michael's linked list.

The difficulty in implementing a lock-free ordered linked list is in ensuring that during an insertion or deletion, the adjacent nodes are still valid, i.e. they are still in the list and are still adjacent. Both the implementation of Harris [30] and that of Michael [67] do so by "stealing" one bit from the pointer to mark a node as deleted, and performing the deletion in two steps: first marking the node, and then deleting it. This bit and the *next* pointer are set atomically by the same CAS operation[4]. The `list_find` operation is the most complicated: it traverses through the list, and stops when it reaches an item that is equal-to or greater-than the searched item. If a marked-for-deletion node is encountered, the deletion is completed and the traversal continues. The `list_find` in Michael's scheme thus improves on that of Harris since by completing the deletion immediately when a marked node is encountered it prevents other operations from traversing over marked nodes, that is, ones that have been logically deleted.

Figure 2.4 shows the various data types and thread-local variables used by Michael's linked list algorithm. These definitions will also apply to hash table code. In Figures 2.5 and 2.6, we bring the three subroutines of the linked list algorithm that are used in our hash table code.

Our Lock-Free Extendible Hash Table Code

We now provide the code of our algorithm. Figure 2.7 specifies several global variables used in the succeeding code listing. The accessible shared data structures are the array of buckets `T`, a variable `size` storing the current table size, and a counter `count` denoting the number of `regular` keys currently inside the structure [5]. The counter is initially 0, and the buckets are set as *uninitialized*, except the first one, which points to a node of key 0, whose `next` pointer is set to `NULL`. Each thread has three private variables `prev`, `cur`, and `next` (from Fig-

[4]Stealing one bit in a pointer in such a manner is straightforward assuming properly aligned memory, and can be achieved with indirection using a "dummy bit node" [2] in languages like Java™ where stealing a bit in a pointer is a problem. The Java™ Concurrency Package proposes to eliminate this drawback by offering "tagged" atomic variables.

[5]Though for the sake of brevity we do not mention it in the presented code, to reduce contention, we have threads accumulate updates locally and update the shared counter `count` only periodically. We included this optimization in the code used in our benchmarks.

ure 2.4), that point at a currently searched node in the list, its predecessor, and its successor. These variables have the same functionality as in Michael's algorithm [67]: they are set by `list_find` to point at the nodes around the searched key, and are subsequently used by the same thread to refer to these nodes in other functions.

In Figure 2.7 we show the implementation of the `insert`, `find` and `delete` operations. The `fetch-and-inc` operation can be implemented in a lock-free manner via a simple repeated loop of CAS operations.

The function `insert` creates a new node and assigns it a split-order key. Note that the keys are stored in the nodes in their split-order form. The bucket index is computed as `key` mod `size`. If the bucket has not been initialized yet, `initialize_bucket` is called. Then, the node is inserted to the bucket by using `list_insert`. If the insertion is successful, one can proceed to increment the item count using a `fetch-and-inc` operation. A check is then performed to test whether the load factor has been exceeded. If so, the table size is doubled, causing a new segment of uninitialized buckets to be appended.

The function `find` ensures that the appropriate bucket is initialized, and then calls `list_find` on `key` after marking it as regular and inverting its bits. `list_find` ceases to traverse the chain when it encounters a node containing a higher or equal (split-ordered) key. Notice that this node may also be a dummy node marking the beginning of a different bucket.

The function `delete` also makes sure that the `key`'s bucket is initialized. Then it calls `list_delete` to delete `key` from its bucket after it is translated to its split-order value. If the deletion succeeds, an atomic decrement of the total item count is performed.

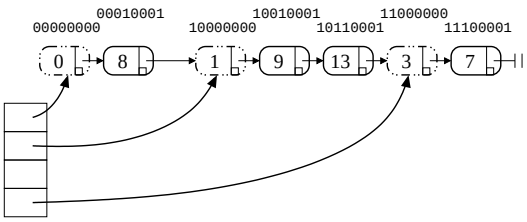
The role of `initialize_bucket` is to direct the pointer in the array cell of the index `bucket`. The value assigned is the address of a new dummy node containing the dummy key `bucket`. First, the dummy node is created and inserted to an existing bucket, `parent`. Then the cell is assigned the node's address. If the parent bucket is not initialized, the function is called recursively with `parent`. In order to control the recursion we maintain the invariant that `parent` $<$ `bucket`, where "$<$" is the regular order among keys. It is also wise to choose `parent` to be as close as possible to `bucket` in the list, but still preceding it. Formally, the following constraints define our the algorithm's choice

of parent uniquely, where "$<$" is the regular order and "$\prec$" is the split-order among keys:

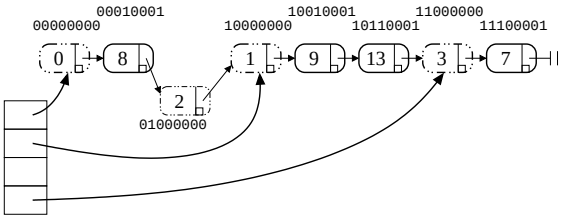
$$\forall k \prec \texttt{bucket}, (k = \texttt{parent} \vee k \prec \texttt{parent})$$
$$\texttt{parent} \prec \texttt{bucket}$$
$$\texttt{parent} < \texttt{bucket}$$

This value is achieved by calling the GET_PARENT macro, that unsets bucket's most significant turned-on bit. If the exact dummy key already exists in the list, it may be the case that some other process tried to initialize the same bucket, but for some reason has not completed the second step. In this case, list_insert will fail, but the private variable cur will point to the node holding the dummy key. The newly created dummy node can be freed and the value of cur used. Note that when line B8 is executed concurrently by multiple threads, the value of dummy is the same for all of them.

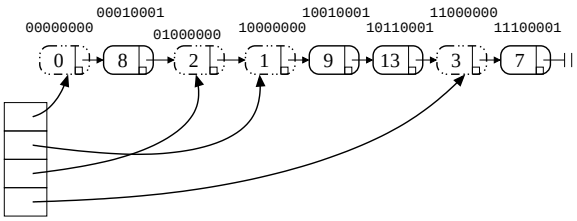
As we will show in the proof, traversing the list through the appropriate bucket and dummy node will guarantee the node matching a given key will be found, or declared not-found in an expected constant number of steps.



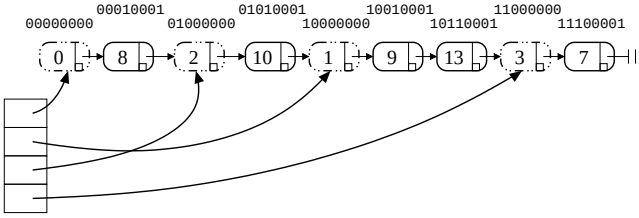
(a) Buckets 0,1 and 3 are initialized. Bucket 2 is uninitialized



(b) Insert(10) is invoked, requiring bucket 2's initialization.
A new dummy node is inserted, with split−order key of 2.



(c) Bucket 2 is assigned to the new dummy node



(d) The split−order regular key 10 is inserted to bucket 2

Figure 2.3: Insertion into the split-ordered list

```
struct MarkPtrType {     // Markable pointer type
  <mark, next>: <bool, NodeType* >;
};

struct NodeType {        // Node: contains key and next pointer
  so_key_t key;
  MarkPtrType <mark, next>;
};

/* thread-private variables */

MarkPtrType *prev;         /* prev */
MarkPtrType <pmark, cur>;  /* curr */
MarkPtrType <cmark, next>; /* next */
```

Figure 2.4: **Types and Structures**. The angular brackets notation denotes a single word type divided to the two fields `mark` and `next`. `mark` is a single bit, while the size of `next` is the rest.

```
int list_insert(MarkPtrType *head,
                NodeType *node) {
  key = node->key;
  while (1)  {
    if (list_find(head, key) return 0;
    node-><mark,next> = <0,cur>;
    if (CAS(prev, <0,cur>, <0,node>))
      return 1;
  }
}

int list_delete(MarkPtrType *head,
                so_key_t key) {
  while (1) {
    if (!list_find(head, key))
      return 0;
    if (!CAS(&(cur-><mark,next>), <0,next>,
        <1,next>))
      continue;
    if (CAS(prev, <0,cur>, <0,next>))
      delete_node(cur);
    else list_find(head, key);
    return 1;
  }
}
```

Figure 2.5: Michael's lock free list based sets

```
int list_find(NodeType **head, so_key_t key) {
try_again:
  prev = head;
  <pmark,cur> = *prev;
  while (1) {
    if (cur == NULL) return 0;
    <cmark,next> = cur-><mark,next>;
    ckey = cur->key;
    if (*prev != <0,cur>)
      goto try_again;
    if (!cmark) {
      if (ckey >= key)
        return ckey == key;
      prev = &(cur-><mark,next>);
    }
    else {
      if (CAS(prev, <0,cur>, <0,next>))
        delete_node(cur);
      else goto try_again;
    }
    <pmark,cur> = <cmark,next>;
  }
}
```

Figure 2.6: Michael's lock free list based sets – continued

```
/* shared variables */
MarkPtrType* T[ ];       // buckets
uint count;              // total item count
uint size;               // current table size

int insert(so_key_t key) {
I1:  node = new node(so_regularkey(key));
I2:  bucket = key % size;
I3:  if (T[bucket] == UNINITIALIZED)
I4:    initialize_bucket(bucket);
I5:  if (!list_insert(&(T[bucket]), node)) {
I6:    delete_node(node);
I7:    return 0;
     }
I8:  csize = size;
I9:  if (fetch-and-inc(&count) / csize > MAX_LOAD)
I10:   CAS(&size, csize, 2 * csize);
I11: return 1;
}

int find(so_key_t key) {
S1:  bucket = key % size;
S2:  if (T[bucket] == UNINITIALIZED)
S3:    initialize_bucket(bucket);
S4:  return list_find(&(T[bucket]),
                      so_regularkey(key));
}

int delete(so_key_t key) {
D1:  bucket = key % size;
D2:  if (T[bucket] == UNINITIALIZED)
D3:    initialize_bucket(bucket);
D4:  if (!list_delete(&(T[bucket]),
                so_regularkey(key)))
D5:    return 0;
D6:  fetch-and-dec(&count);
D7:  return 1;
}

void initialize_bucket(uint bucket) {
B1:  parent = GET_PARENT(bucket);
B2:  if (T[parent] == UNINITIALIZED)
B3:    initialize_bucket(parent);
B4:  dummy = new node(so_dummykey(bucket));
B5:  if (!list_insert(&(T[parent]), dummy)) {
B6:    delete dummy;
B7:    dummy = cur;
     }
B8:  T[bucket] = dummy;
}
```

Figure 2.7: Our split-order based hashing algorithm

2.2.4 Dynamic Sized Array

Our presentation so far simplified the algorithm by keeping the buckets in one continuous memory segment. This approach is somewhat impractical, since table doubling requires one process to reallocate a very large memory segment while other processes may be waiting. In practice, we avoid this problem by introducing an additional level of indirection for accessing buckets: a "main" array points to segments of buckets, each of which is a bucket array. A segment is allocated only on the first access to some bucket within it. The structure of the dynamic sized hash table is illustrated in Figure 2.8.

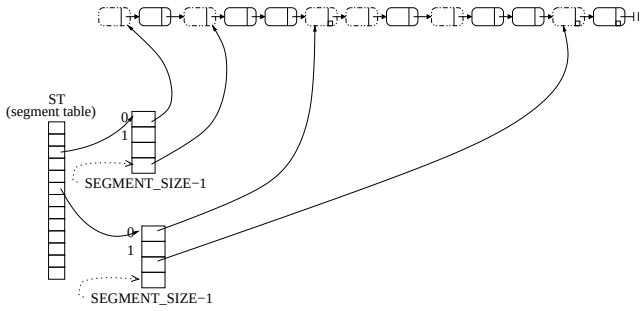


Figure 2.8: Structure of the Dynamic Sized Table

Applying this variation is done by replacing the array of buckets `T` by `ST`, an array of bucket segments, and accessing the table via calls to `get_bucket` and `set_bucket` as defined in Figure 2.9. Referring to the code of Figure 2.7, the lines I3, S2, D2, D4, B2, and B5 will use `get_bucket` to access the bucket, and in line B8 `set_bucket` will be called instead of the assignment. Accessing a bucket involves calculating the segment index and then the bucket index within the segment. In `get_bucket`, if the segment has not been allocated yet, it is guaranteed that the bucket was never accessed, and we can return `UNINITIALIZED`. When setting a bucket, in `set_bucket`, if the segment does not exist we have to allocate it and set its pointer in the segment table.

Asymptotically, introducing additional levels of indirection makes the cost of a single access $O(\log n)$. However, one should view the asymptotic in the context of overall memory size, which is bounded. In our case, each level extends the range exponentially with a very high constant, reaching the maximum integer value using a very shallow hierarchy. A level-4 hirarchiy can exhaust the mem-

ory of a 64-bit machine. Therefore, taking memory size into consideration, the overhead of our construction can be considered as constant.

```
typedef MarkPtrType[SEGMENT_SIZE] segment_t;
segment_t ST[ ];   // the segment table

MarkPtrType * get_bucket(uint bucket) {
  segment = bucket / SEGMENT_SIZE;
  if (ST[segment] == NULL)
    return UNINITIALIZED;
  return &ST[segment][bucket % SEGMENT_SIZE];
}

void set_bucket(uint bucket, NodeType *head) {
  segment = bucket / SEGMENT_SIZE;
  if (ST[segment] == NULL) {
    new_segment = new segment_t;
    new_segment[0..SEGMENT_SIZE-1] =
      UNINITIALIZED;
    if (!CAS(&ST[segment], NULL, new_segment))
      free(new_segment);
  }
  ST[segment][bucket % SEGMENT_SIZE] = head;
}
```

Figure 2.9: Dynamic Sized Array

2.3 Correctness Proof

This section contains a formal proof that our algorithm has the desired properties of a resizable hash table. Our model of multiprocessor computation follows [41], though for brevity, we will use operational style arguments.

Our linearizable hash table data structure implements an abstract *set* object in a lock-free way so that all operations take an expected constant number of steps on average. Our correctness proof will thus have to prove that our concurrent implementation is linearizable to a sequential set specification, that it is lock-free, and that given a "good" class of hash functions, all operations take an expected constant number of steps on average.

2.3.1 Correct Set Semantics

We begin by proving that the algorithm complies with the abstract set semantics. We use the sequential specification of a "dynamic set with dictionary operations" as defined in [12], including the three functions *insert*, *delete* and *find*. The *insert* operation returns 1 if the key was successfully inserted into the set, and 0 if that key already existed in the table. The *find* operation returns 1 if the key is in the set, 0 otherwise. The *delete* operation returns 1 if the key was successfully deleted from the set and 0 if it was not found.

Given a sequential specification of a set, our proof will provide specific linearization points mapping operations in our concurrent implementation to sequential operations so that the histories meet the specification.

Let *list* refer to the non-blocking ordered linked list of all items, pointed to by the buckets of the hash table. Execution histories of our algorithm include sequences of `list_find`, `list_insert`, and `list_delete` operations on this list. Though we argue about these as operations on the shared *list* and not as abstract set operations, our proof will treat these operations as atomic operations. This is a valid approach since they are linearizable by definition of the list-based set algorithms [30, 67]. We do however need to make additional claims about properties of operations on the *list*, since we will apply them to various "midpoints" pointed to by buckets, and not only to the start of the list as in the original use of these algorithms of [30, 67]. To this end we present the following

invariant which refers to the structure of the list in any state in the execution history of our algorithm.

Invariant 1. *In any state:*

- *all keys in the list starting at* `T[0]` *are sorted in an ascending order.*
- *for every* $0 \leq i < size$ *if* `T[i]` *is initialized, then the node pointed by* `T[i]` *holds the key* `so_dummykey[i]` *and is reachable from* `T[0]` *by traversing the list following the nodes' next pointers.*

Proof. Initially, the invariant holds. We will show that every operation that modifies the data structure preserves the invariant. Lines I9 and D6 manipulate the shared counter, but have no impact on the invariant. Line I10 doubles `size`, which adds new buckets, but since `size` only grows, those new buckets are uninitialized, and the invariant is unaffected.

Assuming that the invariant is true just before line I5, we will show that it is preserved. If `list_insert` fails, the shared state has not changed. Otherwise, we use the induction assumption that `T[bucket]` points to a node holding the key `so_dummykey(bucket)`, and that `node` is in the list beginning at `T[0]`. The procedure `list_insert` inserts `node` to the list `T[bucket]`. This trivially preserves the second condition of the invariant for the bucket. The new node's key is the bit reverse of `key OR 0x800...0`. The array index `bucket` and the value of `key` share the same $\log size$ least significant bits, while the rest of `bucket`'s bits are 0. Therefore, the new node's key is ordered after the first node of `T[bucket]`, whose key is the bit reverse of `bucket`. The first part is also preserved, that is, the list reachable from `T[0]` remains sorted since all keys before `T[bucket]` are by the inductive assumption ordered and have lower keys than `so_dummykey(bucket)` and so are properly positioned before the new node, and all other keys are positioned properly by the inductive assumption and the correctness of the `list_insert` operation, since they are a part of the list pointed to by `T[bucket]`.

The `list_delete` operation of line D4 only deletes a key, and thus cannot affect the order. The deleted `node` cannot be the first node of `T[bucket]`, since the least significant bit of its key is 0 and the deleted key's least significant bit is 1.

The function `list_insert` in line B5 inserts a `node` with key `so_dummykey(buc-`

ket) to the sublist T[parent], starting with a node holding so_dummykey(parent). The key parent is defined by turning off the index bucket's most significant "1" bit, so the insertion is not before the first node of the sublist starting at T[parent], and as in the above proof for the case of I5, the invariant is preserved.

Finally, the assignment in B8 sets T[bucket] to either the dummy node created at B4, or the one assigned at B7. In the first case, since a dummy node created in line B4 is inserted, the second condition of the invariant follows immediately from the correctness of the list_insert operation. The first condition follows since the dummy node is inserted in order *after* its parent node which is necessarily ordered before it. In the second case, list_insert failed because the key so_dummykey(bucket) was in the list and cur was by the definition of *list_insert* set to the node holding that key, so both parts of the invariant follow. □

We now define the set H of keys whose items are in the hash table in any given state.

Definition 2.3.1. *For any pointer p, let $S(p)$ be the set of keys in the sorted linked list beginning with the pointer p. Let the* hash table set

$$H = \{\texttt{k} \mid \texttt{so\_regularkey(k)} \in S(\texttt{T[0]})\}.$$

The set H defines the abstract state of the table. For each one of the hash table operations, we will now show that one can pick a linearization point within its execution interval, so that at this point it has modified the abstract state, that is, the set H, according to the specified operation's semantics. Specifically, we will choose the following linearization points:

- the insert operation is linearized in line I5, at the list_insert operation,
- the find operation is linearized in line S4, at the list_find operation, and
- the delete operation is linearized in line D4, at the list_delete operation.

We start with the following helpful lemma.

Lemma 2.3.2. *In lines I5, S4, and D4,* `T[bucket]` *is already initialized, and at B5* `T[parent]` *is already initialized.*

Proof. All of the lines above follow a validation that `T[bucket]` is initialized. If `T[bucket]` is not initialized, `initialize_bucket` is called and the `bucket` is initialized in B8. □

Note that in the proof above we were not interested in whether the initialization sequence (where initializing a bucket causes initialization of the parent) actually terminates, but rather that if it did terminate then all parents of a bucket were initialized.

Lemma 2.3.3. *If* `key` *is in H in line I5, then* `insert` *fails and if it is not,* `insert` *succeeds and* `key` *joins H.*

Proof. When `key` is in H, `so_regularkey(key)` $\in S($`T[0]`$)$. According to Lemma 2.3.2, `T[bucket]` is initialized, and using Invariant 1 we conclude that the node pointed by `T[bucket]` has the key `so_dummykey(bucket)` and it is a part of the list. The list is sorted, and

$$\begin{gathered}\texttt{so\_dummykey(bucket)} = \texttt{REVERSE(bucket)} = \\ \texttt{REVERSE(key mod size)} < \texttt{REVERSE(key OR 0x800..0)} = \\ \texttt{so\_regularkey(key)}.\end{gathered} \tag{2.1}$$

Thus, the searched key is in the sublist, $S($`T[bucket]`$)$. The `list_insert` at I5 will fail and so will `insert`. If `key` is not in H, it is also not in $S($`T[bucket]`$)$, and `list_insert` inserts `so_regularkey(key)` in the `bucket`'s sublist. From that state on, `so_regularkey` $\in S($`T[0]`$)$, i.e. `key` is in H. □

Lemma 2.3.4. *If* `key` *is in H at line S4, the* `find` *succeeds, and otherwise the* `find` *fails.*

Proof. If line S4 is executed when `key` is in H, then `so_regularkey(key)` is in $S($`T[0]`$)$. `T[bucket]` is assigned to a node in that list, holding the key `so_dummykey(bucket)`. Using Equation 2.1, we conclude that the searched key is in $S($`T[bucket]`$)$, so `list_find` succeeds and so does `find`. If in line S4 `key` is not in H, it cannot be in $S($T[bucket]$)$, so `list_find` fails. □

Lemma 2.3.5. *If* `key` *is in* H *in line D4,* `delete` *succeeds and removes* `key` *from* H*, and otherwise* `delete` *fails.*

Proof. If `key` is in H, then `so_regularkey(key)` is in $S($`T[0]`$)$. `T[bucket]` is assigned to a node inside that list, where the key of that node is `so_dummykey(bucket)`. Using Equation 2.1, we conclude that the searched key is in $S($`T[bucket]`$)$, so `list_delete` removes it. If `key` is not in H, it cannot be in $S($`T[bucket]`$)$, so `list_delete` fails. □

From Lemma 2.3.3, Lemma 2.3.4, and Lemma 2.3.5 it follows that:

Theorem 2.3.6. *The split-ordered list algorithm of Figure 2.7 is a linearizable implementation of a set object.*

2.3.2 Lock Freedom

Our algorithm uses loads and stores together with implementations of a list-based set, a shared counter, and memory allocation routines as primitive objects or operations. As we will show, in terms of these primitive operations the algorithm's implementation is wait-free, that is, each thread always completes in a finite number of operations. This implies that its overall progress condition in terms of primitive machine operations will be exactly that of the underlying implementation of those objects. Since we used the lock-free list-based sets of [30, 67] and a lock-free shared counter as building blocks in this presentation, our implementation will also be lock-free. As noted in the introduction, in some cases there are advantages in using the obstruction free list-based set algorithm of [59]. If [59] is used together with a lock-free shared counter, our hash table will be obstruction free [45].

Theorem 2.3.7. *The split-ordered list algorithm of Figure 2.7 is a wait-free implementation of a set object in terms of* `load`, `store`, `fetch-and-inc`, `fetch-and-dec`, `list_find`, `list_insert` *and* `list_delete` *operations.*

Proof. The functions `insert`, `find`, `delete` and `initialize_bucket` all take a finite number of steps, each of which is a machine level `load` or `store` operation or an operation on the list based set object or the shared counter. The `initialize_bucket` procedure is the only one with a recursive call. However,

the recursion of `initialize_bucket` is limited, since each step is executed on the `parent` of a `bucket`, which satisfies `parent` $<$ `bucket`. Since `bucket` 0 is initialized from the start, the recursion is finite, and the implementation is wait-free. □

The lock-freedom property means that a thread executing the hash table operation completes in a finite number of steps unless other threads are infinitely making progress. Thus, it is a weaker requirement than wait-freedom, and by combining implementations the following is a corollary of Theorem 2.3.7:

Corollary 2.3.8. *The split-ordered list algorithm of Figure 2.7 with lock-free implementations of* `list_find`, `list_insert`, `list_delete`, `fetch-and-inc`, *and the* `fetch-and-dec` *operations is lock-free.*

Corollary 2.3.9. *The split-ordered list algorithm of Figure 2.7 with obstruction-free implementations of* `fetch-and-inc`, `fetch-and-dec`, `list_find`, `list_insert` *and* `list_delete` *operations is obstruction-free.*

The `fetch-and-inc` and `fetch-and-dec` operations have known lock-free implementations [69].

2.3.3 Complexity

The most important property of a hash table is its expected constant time performance. When analyzing the complexity of hashing in a concurrent environment there are two adversaries one needs to consider: one controlling the distribution of hash values of keys by the hash function (i.e. how good is the hash), the other controlling the scheduling of thread operations. We will follow the standard practice of modelling the hash function as a uniform distribution over keys [12]. The uniformity of keys we assume is global, that is, it extends across all threads in a given execution (A simple way to think of this is that we apply the standard uniform distribution assumption [12] on the linearization of any given execution). We will use the term *expected time* (or *expected number of steps*) to refer to the expected number of machine instructions per operation in the worst case scheduling scenario, assuming a hash function of uniform distribution. We will use the term *average time* (or *average number of steps*) to refer to the number of machine instructions per operation averaged over all

executions, also assuming a uniform hash function. It follows that constant expected time implies constant average time.

In our complexity analysis we assume that loops within the underlying linked list code involve no more than a constant number of retries. This assumption is realistic since a non-constant number of retry loops implies *Compare&Swap* failures caused by contention within a single bucket, which cannot occur due to the global uniformity of the hash function.

We will show that under any scheduling adversary, our algorithm performs all hash table operations in constant average time. The complexity improves to constant expected time if we assume a *constant extendibility rate*. This is a restriction on the scheduler that requires that the table is never forced to extend a non-constant number of times while a thread is delayed by the scheduler. It means that given a good hash function, the adversary cannot cause any single operation to take more than a constant number of steps unless it delays its progress through more than a constant number of global resize operations. Formally, when there are n items in the data structure, a thread must complete a single operation before $n \cdot 2^c$ successful insertions of elements by other threads were completed, where $c \in O(1)$. We believe this is the common situation in practice.

Two algorithmic issues require a detailed proof: one is the complexity of list operations, which is essentially the complexity of executing a `list_find`, and the other is the complexity of `initialize_bucket`, which involves recursive calls.

Denote by n the total number of items in the set, and by s the number of buckets. For the complexity analysis, we are not interested in the cases where the table is small, so we make the assumption that s is greater than the number of threads. Let L denote the load factor `MAX_LOAD` in our code, typically a small constant.

Lemma 2.3.10. *For any number p of threads, at all times the following condition holds:*

$$\frac{n-p}{s} \leq L$$

Proof. Focus on the successful completed `insert` and `delete` operations. Each successful insertion incremented `count` by 1, and each successful deletion decremented it. In any state there are no more than p concurrent operations. Every one of the "already completed" insert operations checked, when executing line I9, that the ratio of `count` and `csize` is not more than L, and doubled the `size` if the gap was exceeded. At all times, there are no more than p currently executing `insert` operations. Therefore, when $n/s > L$ and a resize is needed, no more than p new keys can be inserted to the data structure before the resize takes place. □

Lemma 2.3.11. *Assuming a hash function of uniform distribution, the probability that a bucket is not accessed during the time where the table size is s, is asymptotically bounded by $e^{-L/2}$.*

Proof. Focus on a growing table from size $s/2$ to s and then to $2s$. According to Lemma 2.3.10, in the state in which line I10 doubled the table from $s/2$ to s, the number of items in the table was less or equal to $p + Ls/2$. When later in line I10 the table doubled in size to $2s$, the condition of line I9 implies that the number of items was at least Ls. The last two observations imply that during the set of states in which `size` was s, the item count increased by at least $Ls/2 - p$, i.e. line I9 was executed at least $Ls/2 - p$ times. When we consider at most p processes that may have begun the insert operation when `size` was less than s, we get that line I2 was executed at least $Ls/2 - 2p$ times.

Assuming a uniform distribution of the keys, the probability that a bucket b was not accessed during this period is at most $(\frac{s-1}{s})^{Ls/2-2p}$. When p is significantly smaller than s, as assumed, the last expression is asymptotically equal to $e^{-L/2}$. □

Lemma 2.3.12. *For any key k, when the table size is s and the bucket $k \bmod size$ is initialized, there is no dummy node with key d such that $k \bmod size \prec d \prec k$, that is, d's split-order value is between those of $k \bmod size$ and k.*

Proof. Assume by way of contradiction that d is the key of a node such that: $k \bmod size \prec d \prec k$. It is the case that $d < size$ because d is in the list, and bucket indices are always smaller than the table size. Therefore, d has less than $\log_2(size)$ non-zero bits. The keys k and $k \bmod size$ have at least $\log_2(size) - 1$ identical less significant bits. The split-order value of d is between them,

so it must have the same low $log_2(size) - 1$ bits, that actually constitute all of its non-zero bits. This implies that $d = k \bmod size$ under the split-order, a contradiction to the assumption that $d \succ k \bmod size$. □

Lemma 2.3.13. *If the hash function distributes the keys uniformly then:*

- *In any execution history, the list traversal of* `list_find` *takes constant time on average.*
- *Under the constant extendibility rate assumption, the traversal of* `list_find` *takes expected constant time.*

Proof. For a table of size s, the expected number of uninitialized buckets among the first $s/2$ buckets is no more than $s/2 \cdot e^{-L/2}$, by Lemma 2.3.11. For each of the initialized buckets, there is a dummy node in the list holding the bucket index as the split-order value. Therefore, there are at least $s/2 \cdot (1 - e^{-L/2})$ dummy nodes with keys from $0..s/2 - 1$. Those values divide the integer range into $s/2$ equal segments, while the missing items are distributed evenly. Using Lemma 2.3.10, there are on average less than

$$\frac{n}{s/2 \cdot (1 - e^{-L/2})} \leq \frac{Ls + p}{s/2 \cdot (1 - e^{-L/2})} = \frac{2L + 2p/s}{1 - e^{-L/2}} \tag{2.2}$$

nodes between every two dummy nodes. The operation `list_find` is called to search for a key *k* from the bucket *k mod size*, so using Lemma 2.3.12 we conclude that in the state in which it was called there were no dummy nodes between the bucket's dummy node and the node at which the search would be completed. We have just computed that dummy nodes are distributed in intervals of less than $\frac{2L+2p/s}{1-e^{-L/2}}$ nodes, implying that if the table size does not change, the search will take no more than a constant expected number of steps.

We will now show that if the search took more than constant time, there were enough successful inserts to maintain a constant number of steps on average. If `list_find` took $\Omega(r)$ steps, $\Omega(r)$ dummy nodes must have been traversed, since at any time the expected distance between them is constant. All of these dummy nodes were inserted to the list after `list_find` started. The number of dummy nodes in the original bucket doubles each time the table is extended, so there were $\Omega(\log r)$ table resize events. Since there were exactly n items in the table when the `list_find` operation started, the number of items had to rise

by $\Omega(rn)$, i.e. $\Omega(rn)$ successful insertions to the list. There were no more than p threads that successfully executed `list_insert` but then were delayed before completing the `insert` routine. Therefore, we can consider only $\Omega(rn - p)$ as complete hash table insertions. According to the constant extensibility rate assumption, a thread must complete a single operation within $n \cdot 2^c$ successful insertions. Looking at the single operation that took $\Omega(r)$ steps, we now know that during that time there were at least $\Omega(rn - p)$ successful inserts, but we also know that the operation lasted less than $n \cdot 2^c$ successful operations. We get that $\log(r - p/n) \in O(1)$, and thus $r \in O(1)$. □

Lemma 2.3.14. *Given a hash function with an expected uniform distribution, the number of steps performed by the function* `initialize_bucket` *is constant on average. Under the constant extendibility rate assumption, the number of expected steps in the worst case execution is constant.*

Proof. A recursive call to `initialize_bucket` terminates when the parent bucket is initialized. To have m recursive calls, m uninitialized ancestor buckets are needed. Applying Lemma 2.3.11, this may happen with probability less than $e^{-L(m-1)/2}$. The number of m-deep executions among m calls to `initialize_bucket` is $m \cdot e^{-L(m-1)/2} \in O(1)$, implying that the expected number of recursive calls is constant. By Lemma 2.3.13, the `list_insert` call inside `initialize_bucket` costs a constant number of steps on average. If we assume constant extendibility rate (threads are not delayed while the table is doubled a non-constant number of times), a recent ancestor of every bucket is always initialized, and the recursion depth is constant. Also, according to Lemma 2.3.13, the execution of `list_insert` is of expected constant time. □

Theorem 2.3.15. *Given a hash function with expected uniform distribution, all hash table operations complete within a constant number of steps on average. Assuming a constant extendibility rate, all hash table operations complete within expected constant number of steps.*

Proof. Beside executing a constant number of simple instructions, all hash operations call a list traversing routine twice at most (actually, only `hash_delete` may cause `list_find` to run twice). By Lemma 2.3.13 the list traversals cost a constant average number of steps, and by Lemma 2.3.14 the `initialize_bucket` operation also completes within a constant average number of steps. Both of

the above lemmas imply that under the constant extendibility rate assumption, the number of steps is constant in the worst case execution assuming a uniform distribution. □

2.4 Performance

We ran a series of tests to evaluate the performance of our lock-free algorithm. Since our algorithm is the first lock-free extensible hash table, it needs to be proven efficient in comparison to existing lock-based extensible hash table algorithms. We have thus chosen to compare our algorithm to the resizable hash table algorithm of Lea [54] (revision 1.3), originally suggested as a part of *util.concurrent.ConcurrentHashMap*, the proposed Java™ Concurrency Package, JSR-166.

Lea's algorithm is based on an exponentially growing table of buckets, doubled when the average bucket load exceeds a given load factor. Access to the table buckets is synchronized by 64 locks, dividing the bucket range to 64 interleaved regions, i.e. lock i is obtained when bucket b is accessed if $b \; mod \; 64 = i$. `Insert` and `delete` operations always acquire a lock, but `find` operations are first attempted without locking, and retried with locking upon failure. When a process decides to resize the table, it locks all 64 locks, allocates a larger array and rehashes the buckets' items to their new buckets, utilizing the simplicity of power-of-two hashing. This scheme offers good performance, in comparison to simpler schemes that separately lock each bucket, by significantly reducing the number of locks that need to be acquired when resizing. Figure 2.10 illustrates the effect of different concurrency levels on Lea's algorithm performance.

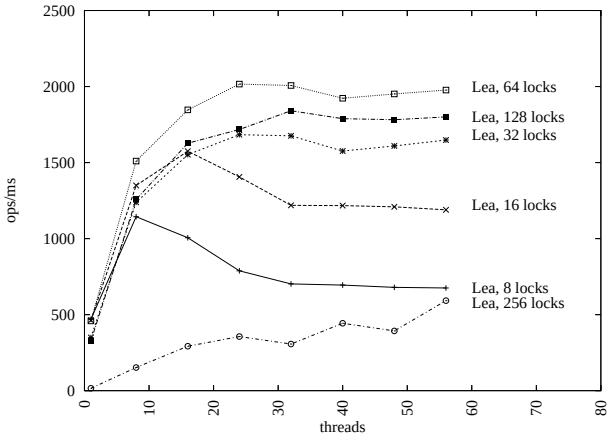


Figure 2.10: Lea's algorithm with different concurrency levels

We translated the Java™ code by Lea to C++ and simplified it to handle integer keys that also serve as values, exactly as in our new algorithm's code. There is a trade-off in this algorithm: the more locks used, the lower the contention on them, but the higher the global delay when resizing. We thus ran an experiment to confirm that in the translated algorithm there is no significant advantage to using more or less than 64 locks.

We compared our split-ordered hashing algorithm to Lea's algorithm using a collection of experiments on a 30-processor Sun Enterprise 6000, a cache-coherent NUMA machine formed from 15 boards of two 300MHz UltraSPARC® II processors and 2GB of RAM on each. The C/C++ code was compiled with a Sun *cc* compiler 5.3, with the flags `-xO5` and `-xarch=v8plusa`. We executed each experiment three times to lower the effect of temporary scheduling anomalities.

Lea's algorithm has significant vulnerability in multiprogrammed environments since whenever the resizing processor is swapped out or delayed, the algorithm as a whole grinds to a halt. The significant latency overhead while resizing would also make it less of a fit for real-time environments. However, our tests here are designed to compare the performance of the algorithms in the currently more common environments without multiprogramming or real-time requirements.

Since Lea's algorithm behaves differently when hash table operations fail rather than succeed, we also tested the algorithms in scenarios where they begin after a significant amount of elements have been inserted. As the range from which the elements are selected is limited, the more we pre-insert, the more chances are for an element to be already in the table when search for. Additionally, we ran a series of experiments measuring the change in throughput as a function of concurrency under various synthetic distributions of *insert*, *delete* and *find*.

To capture performance under typical hash-table usage patterns [53] we first look at a mix that consists of about 88% *find* operations, 10% *insert*s and 2% *delete*s. Our first graph, in Figure 2.11, shows the results of comparing the algorithms under such a pattern. The hash table load factor (the number of items per bucket) for both tested algorithms was chosen as 3. In the presented graph we show the change in throughput as a function of concurrency. As can be seen, at high loads the lock-free split-ordered hashing algorithm significantly

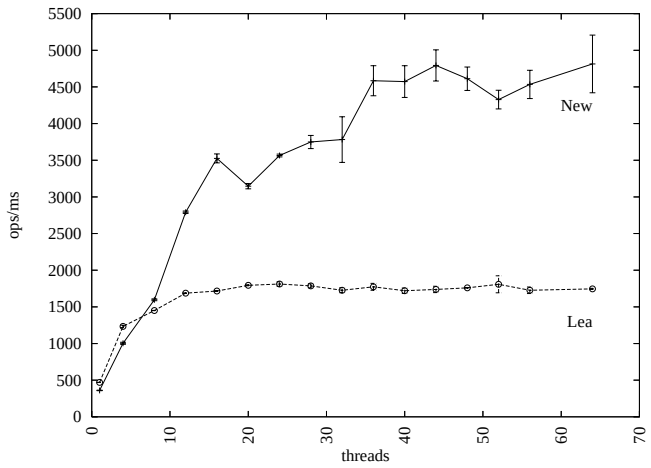


Figure 2.11: Throughput of both algorithms. Standard deviation is denoted by vertical bars

outperforms Lea's when the concurrency level goes beyond eight threads.

The first data point, corresponding to the throughput when executed by a single thread, is a measure for the overhead cost of the new algorithm. According to this data point, the new algorithm is 23% slower than the lock-based algorithm when run by a single thread.

- Lea's algorithm reaches peak performance at about 24 threads and at the same concurrency level, our new algorithm has two times higher throughput.
- Our algorithm reaches peak performance at 44 threads, where it is almost three times faster than Lea's.
- Our algorithm's performance fluctuates after reaching peak performance because it involves significantly higher concurrent communication and is thus much more sensitive to the specific layout of threads on the machine and to the load on the shared crossbar.
- Lea's algorithm suffers a much milder deterioration caused by the architectural critical paths because it never reaches high concurrency levels and its overall performance is limited by the bottlenecks introduced by the shared locks.

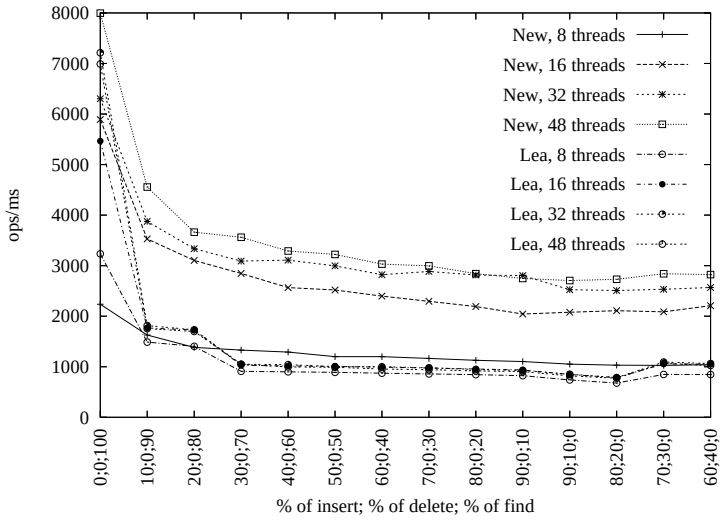


Figure 2.12: Varying operation distribution

Figure 2.12 shows the results of an experiment varying the chosen distribution of `inserts`, `deletes`, and `finds`. Note that our algorithm consistently outperforms Lea's algorithm throughout the full range of tested distributions. We also ran an experiment that varies the load factor in our algorithm. As seen in Figure 2.13, the load factor does not affect the performance significantly, and its effect is in any case minimal when compared to those of the thread layout and the overall communication overhead.

Figure 2.14 shows the throughput of both algorithms when the amount of pre-insertions varied among 0, 300K, 600K, and 900K. The range from which elements were selected was $[0, 1e + 6]$, so pre-insertions affected significantly the success rate of the hash table operations. The performance of Lea's algorithm slightly improves on lower concurrency levels, but from 12 threads and on the new algorithm is faster.

We also tested the robustness of the algorithms under a biased hash function, mimicking conditions in case of a bad choice of a hash function relative to the given data. To do so we generated keys in a non-uniform distribution by randomly turning off 0 to 3 LSBs of randomly chosen integers. Our empirical data shows that our algorithm shows greater robustness: it was slowed down by approximately 7%, while Lea's algorithm's performance decreased by more than

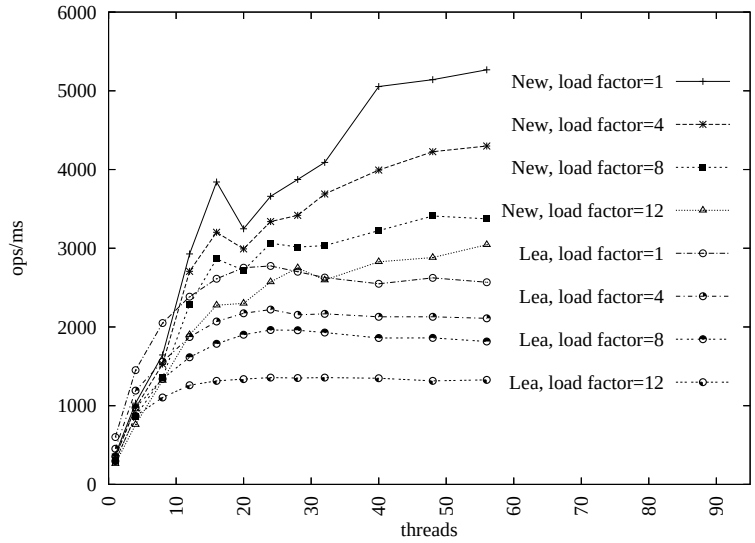


Figure 2.13: Varying load factor

30%. The reason for this is that a biased hash function causes some number of buckets to have many more items than the average load. The locks controlling these buckets in Lea's algorithm are thus contended, causing a performance degradation. This does not happen in the lock-free list used by the new algorithm.

Based on the above results, we conclude that in low-load non-multiprogrammed environments both algorithms offer comparable performance, while under medium to high loads, split-ordered hashing scales better than Lea's algorithm and is thus the algorithm of choice.

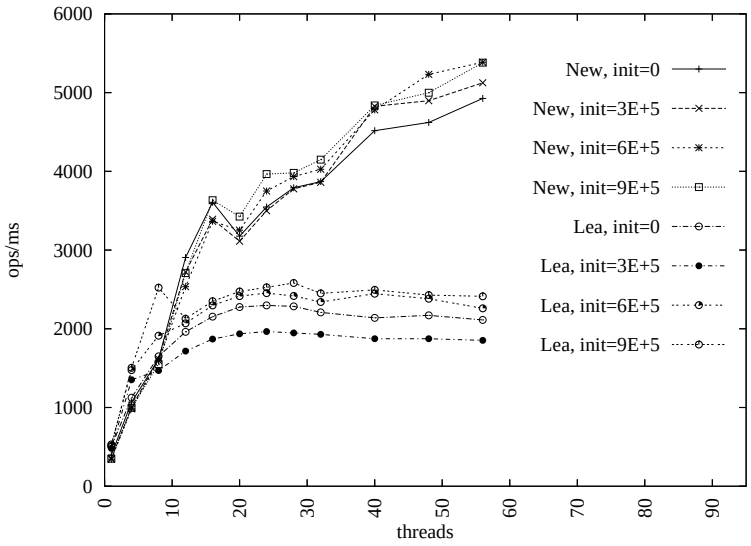


Figure 2.14: Varying amount of pre-insertions

Chapter 3

Predictive Log Synchronization

3.1 Introduction

In the next two chapters, we present two different schemes based on the novel concept of *Log-Synchronization*. These two schemes are aimed at providing a general synchronization solution for large and highly concurrent data structures, and in some sense, they can be considered as an alternative to traditional locking and software transactional memory.

In this chapter, we focus on the *Predictive Log-Synchronization* (PLS) framework. PLS allows concurrent access to a shared data structure, but also simplifies concurrent programming and program verification by requiring programmers to write only specialized *sequential code*. This code is then transformed into a *non-blocking* concurrent program in which threads coordinate all data structure operations via a shared log. Like with transactional memory, being non-blocking eliminates the possibility of deadlocks and mitigates some of the effects of processor delays.

3.1.1 Log-Synchronization

Log-synchronization is founded on the belief[1] that in many concurrent data structures used in real-world applications, the ratio of high level operations (such as class methods) that modify the structure to ones that simply read it, greatly favors read-only operations. What's more, in the modifying method calls themselves, the ratio of operations other-than-writes (reads, branches, stack calls) to writes greatly favors the other-than-write operations. This implies that delegating all writes to a single lock-protected thread at a time may not significantly harm the overall throughput, especially if one can reduce the number of operations on the lock's critical path to only modifying instructions. Moreover, as we show, it can greatly increase the throughput of concurrent read-only operations. Log-synchronization allows one to readily modify existing coarse-grained lock-based code to allow parallelism, atomicity (i.e. linearizability [40]), and most notably, *no deadlocks*.

[1] Based on accepted folklore, at this point unsubstantiated by statistical data.

3.1.2 Predictive Log-Synchronization

In a nutshell, in PLS the shared data structure is duplicated, protected by a high level lock, and appended with a special *log* of high level operations. A thread owning the lock performs all data structure modifications logged by others on one copy, allowing all threads to concurrently read the other unmodified copy, and switches copies before releasing the lock. PLS is non-blocking since threads failing to acquire lock ownership make progress by inspecting the log and *predicting* the result of their own operation. Concurrently, all read-only operations can access the unmodified copy of the data structure in a non-blocking manner and with virtually no overhead (without processing the contents of the log.)

PLS is thus, in a sense, orthogonal to transactional memory. It is motivated by our belief that for many classes of applications, the number of calls to high level operations that modify a data structure is significantly smaller than that of ones that simply read it. What's more, many natural data structures (consider stacks, queues, heaps, search trees) have inherent sequential bottlenecks limiting the throughput of modifying high level operations. Thus, delegating all data structure modifications to a single lock-controlled thread at any given time does not significantly harm performance, yet, as we show, will allow high read-only throughput and simplified concurrent programming.

The idea of using a log to collect "write" operations and execute them in a batch mode has been proposed in the context of *log structured file systems* [81]. The idea of logging lists of operations on a lock-protected object to be executed by the thread owning the lock was proposed by Oyama et al [74] as a low overhead means of increasing locality in performing modifications to shared locations within a mutually exclusive section. In [93], Welc et al. use logs as a conflict detection mechanism for transactions, and in [92], they use logging to support rollback for delayed low-priority threads holding a resource. McKenney and Slingwine in [64] proposed the *read-copy update* (RCU) methodology, according to which modifying threads cooperatively schedule callbacks in such way that concurrent readers are not required to acquire locks in order to get a consistent view of the data structure. Writer thread operations are logged as callbacks, but unlike PLS, the log is not read by peer threads. RCU does not eliminate the need to use locks and handles synchronization based on integra-

tion with the operating system through the use of interrupts. PLS differs from the above approaches in a fundamental way: it uses the log to allow readers to predict the effect of write operations before they actually take effect, allowing them to proceed concurrently and in a non-blocking manner even if the actual modifications are performed sequentially by some process at a later time. The idea of switching among duplicate copies of memory locations to allow consistent concurrent reads appears in Riany et al [79] and was well known in various forms in the garbage collection and database literature long before.

In Chapter 4 we show how to construct an address-based word-level log synchronizer as a replacement for monitors. PLS differs significantly from that construction, which is a lower level mechanism, using the log to record word level operations, instead of method descriptions. More importantly, our address-based log synchronizer does not use prediction and is thus blocking, in contrast to PLS which introduces prediction as a means of providing non-blocking progress even though the data structure is controlled by a lock.

3.1.3 Performance

We present a set of proof-of-concept micro-benchmarks intended to show that there is merit to the predictive log-synchronized approach, but are in no way intended to be a full and comprehensive exposition of PLS performance. We note that our benchmarks show high throughput but deteriorating PLS performance as concurrency increases. We believe that the results would look much better if we had some control over scheduling and/or could heuristically decide whether a thread should apply prediction or just backoff when failing to acquire the shared lock.

Though log-synchronization will not benefit every data structure under every workload, as an example of its potential, we chose a commonly benchmarked data structure: a red-black tree data structure representing a set (used in benchmarking various software transactional memory systems [29, 62, 44, 14]) and wrote it in a predictive log-synchronized manner. We compared our implementation to lock-based and an STM-based Java^{TM} implementations of a red-black tree set provided in [61]. The STM we used was the implementation by Marathe et al. of Harris and Fraser's OSTM [21]. This is an "object-based" STM in which transactions are carefully designed to open and close objects for

read and write to minimize STM calls. Our experiments show that PLS performs better than monitors and in write-dominated workloads, better than the object-based OSTM [61].

3.1.4 PLS vs. STM

The following is a short summary of our current understanding of the relative benefits and drawbacks of PLS relative to STM.

- PLS is directed at the implementation of concurrent data structures as shared objects. Its scope is thus not as general as STMs which can support transactions applied across objects. One could extend PLS in this direction but this would be outside the scope of this thesis.
- The memory consistency condition provided by most STMs is Herlihy and Wing's linearizability [47], while the consistency provided by PLS is Lamport's sequential consistency [52]. Linearizability is a stronger property that guarantees better composability properties then sequential consistency.
- STMs offer concurrent transactional programming as an interface. Transactions simplify code by collecting operations into atomic transactions, but require the programmer to understand concurrency, granularity vs. performance tradeoffs, nesting, early-release, and various other semantic issues. The PLS Log-based programming may be simpler as it involves no concurrent reasoning since it is based on writing only sequential code. However, this sequential programming is not as straightforward as transactions: it requires an understanding of log-based computation. It is not clear which of these two approaches provides the better programming environment.
- Transactional code is concurrent (one must restrict granularity of atomicity to achieve good performance) and, though simpler than lock based code, is hard to verify correct, while log-synchronized code is sequential and thus significantly simpler to verify. This could prove to be a big benefit of PLS, as verification of concurrent programs is hard.

- There are various complex programming language issues with transactions that have yet to be solved, among them the handling of nesting and I/O. These issues are easily solved in PLS by requiring any operation containing nesting or I/O to be executed only when holding the lock.

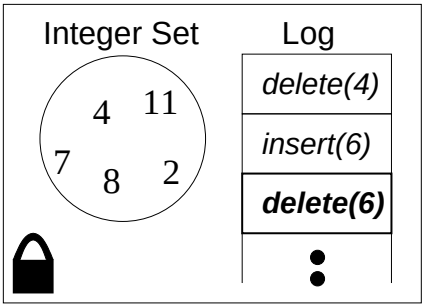


Figure 3.1: PLS Components of an Example Integer Set

3.2 PLS Design

This section explains the programming and algorithmic design aspects of PLS.

3.2.1 Log-based Programming

How does the log-based programming of the PLS framework work? Consider the following log-based computational model: a data structure has an initial state and a log that contains the history of all high level operations (method calls) applied to the initial state. Clearly the application of the log to the initial state describes the current state of the structure at any point in time. To keep the log from growing without end, we can at any point apply in-order a subsequence of the operations at the head of the log to the state to derive a new state, and remove these operations from the log. This means that in order to compute the outcome of a given operation, one need only apply the shortened log to the state.

However, we can be even more efficient. Consider for example a log-based computation of a set object. In Figure 3.1, we add to the log an attempt to $delete(6)$ which deletes the number 6 from the set and returns true if successful and false if 6 was not found in the set. Figuring out the result of the delete operation requires searching for 6 in the state representing the set, and then searching the log for the possible effects of preceding operations on this state. In the example, the element 6 was not found in the set represented by the state, and in order to return the result one need search the log *only* for $insert(6)$ and

delete(6) operations and apply them to the state. There is no need to apply any other operations in the log. Log synchronized programming thus has two main components:

State and Operation Design Design the state representation (the data structure) and the methods that modify it, all of which consist of standard sequential code (in our example this could be the set represented as a linked-list or a red-black tree, and the corresponding implementation of sequential code for insert, delete, and find on them. One could use the existing sequential code base for this part).

Efficient Prediction Design the log application pattern, that is, the way operations need to be applied from the log to the state in order to compute the outcome of a given operation. Programmers must specify what data is extracted from the state, and how operations in the log affect the result of another operation (in our example, a delete on a set requires applying the sequence of inserts and deletes only for the same value, independently of how the state and operations are implemented). This part of the programming is done once and for all for each data type.

There is a third *adjustments* component which we will discuss later, essentially providing hint pointers into the state data structure to speed-up modifications to it. The adjustments part requires an understanding of the internal structure of the sequential implementation of the data structure operations. The adjustments part is not really an unknown style of programming, rather, the adjustments are the same as one would create an efficient reconstruction sequence for a data structure as is done for backups.

Thus, all of the above log-based programming elements are free from any concurrency considerations. The concurrency is all hidden within the PLS implementation. Given a log-based description (program) for a given data-structure, we apply an automatic transformation that generates a concurrent PLS implementation that is a *sequentially consistent* [52] implementation of the data structure specified by its state and operations. This means that the result of any concurrent program execution is equivalent to that of some sequential ordering of all its operations in which each thread's operations are executed according to its own program order.

Though data structures for which there are no efficient prediction solutions may exist, looking through many commonly used structures, we could not find such structures. One should also remember that there are complex data structures, take Fibonacci heaps [22] as an example, for which even the transactional memory interface is non-trivial to use in order to compose an efficient implementation. We conducted an ad-hoc "programming experiment" [2] in which we timed the marginal programming effort of applying the PLS framework to a Fibonacci heap algorithm, assuming existing log-based code of a vanilla heap. Our experience was that the PLS based concurrent Fibonacci heap code required one hour of programming.

3.2.2 The PLS Algorithm

As we explained earlier, the log synchronized approach to introducing parallelism into a data structure is to allow threads to make progress by reading a consistent copy of the data and, if needed, using a log of high level operations to deduce the outcome of their operations. A modifying thread that does manage to get hold of the lock controlling the state, applies the operations in the log to the data structure representing the state, as a service to the other threads.

Let us return to our earlier example of a log-based implementation of a set of integers. Assume now that we wish to provide a PLS implementation of this algorithm, one in which concurrent threads perform inserts, deletes, and lookups. Each thread performing an insert or delete starts by appending its current operation to the shared log, and a single thread successfully acquires the lock. While this thread executes the set of operations in the log, other threads make progress. Recall the example of Figure 3.1: a thread deleting the number 6 from the set searches for 6 in the set, and analyzes the effect of preceding operations in the log on its result. Even if the element 6 was not found in the shared set, the thread searches the log for $\mathit{insert}(6)$ and $\mathit{delete}(6)$ operations appearing before its own. Only then it can decide on the result of the requested $\mathit{delete}(6)$ operation.

Unfortunately, it cannot be assumed that during the modification of the data structure by the modifying thread, other threads would read consistent data. Our solution is to have two copies of the data structure, one used for writing

[2] Not in any way a benchmark.

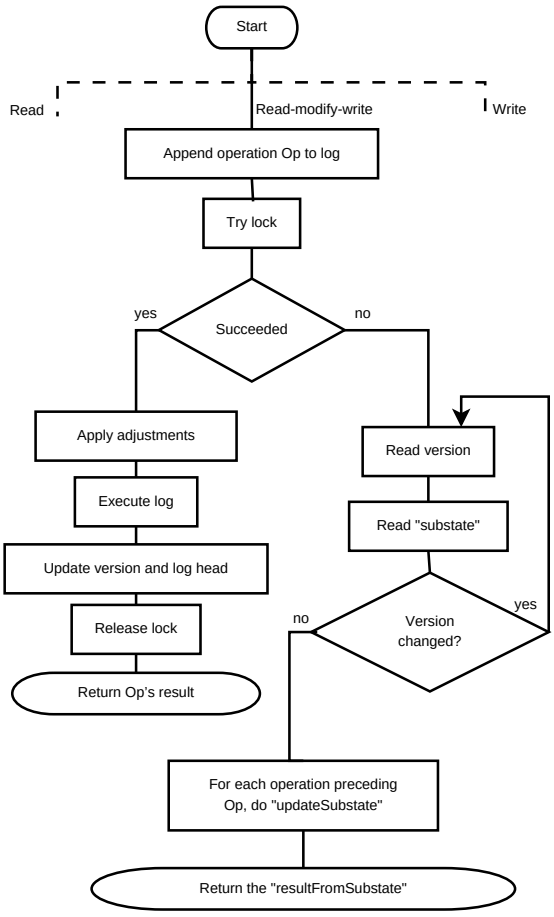


Figure 3.2: Flowchart of Read-Modify-Write PLS Execution

and the other for reading. The two copies are switched at the end of every modifying session, a process that is coordinated via a global version count. When the version count is even, the first copy is used for reads and the second for writes, and when odd, they switch roles. That way, the "logical" application of all the changes of the last session is done by merely incrementing the version. We expand later in this section on how we handle version changes during reads.

There are two major implications of the need to maintain two data structures instead of one. One is the greater amount of space consumed, and the other is the cost of applying the changes to the other copy as well, a process we call

adjustment. Note that for many data structures, one can optimize both space and computation to significantly less than double the cost of a single structure. Space consumption can be minimized by not duplicating data fields that are not accessed concurrently, and the cost of the second modification (the adjustment) can be minimized by using information recorded while modifying the first structure.

High level operations performed on shared data structures can be divided into three groups: *read-only*, *write-only*, and *read-modify-write*. Read-only operations do not mutate the data, write-only operations modify the data but do not return a result, and read-modify-write operations change shared data and return a result that is dependent on the data read. The three types imply different constraints on the structure's consistency, and are thus treated differently by the predicting threads. Threads performing write-only operations do not need to predict a result, they may complete by simply appending their operation to the operation log, to be executed at a later time by the modifying thread. This will still be sequentially consistent. We describe the implementation of read-only operations in the sequel.

Consider now the implementation of read-modify-write operations as summarized in the flowchart in Figure 3.2. After appending the operation to the log, the modifying thread tries to acquire the object lock. Upon success, following the left-hand side of the chart, that *modifying* thread re-applies the modifications of the previous session (the adjustments) to the writable copy of the data structure. Then, it executes each one of the operations in the log, including its own, according to the order they were queued. Before releasing the lock, the version count is incremented and the log head is updated to point to an empty log or a log containing the very recently appended operations, if such exist. The version change immediately switches the roles of the two data structure copies, making the modifications appear to take place instantaneously.

The execution of threads that have not successfully acquired the lock, the *predicting* threads, is depicted on the right-hand side of the flowchart. This execution is dependent on complementary sequential code supplied by the programmer, consisting of three parts: the first, called `makeSubstate` is responsible for extracting the needed information from the current state of the data structure's readable copy. We refer to this information as the *substate*. The second, `updateSubstate` is for applying the effect of a preceding operation in the log on

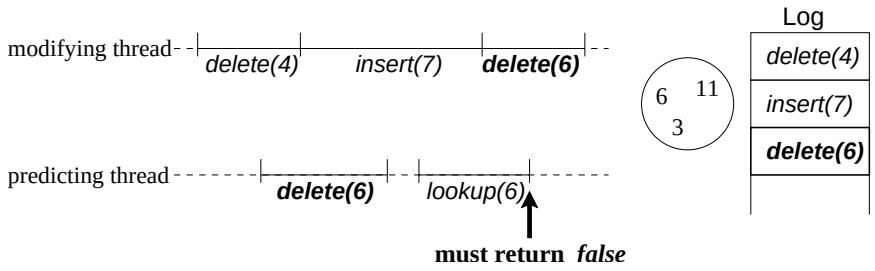


Figure 3.3: Enforcing Sequential Consistency

the substate. The third, resultFromSubstate is for extracting the operation's final result from the substate.

In the integer set example, when predicting the result of *delete*(6), makeSubstate returns the substate, a boolean denoting whether 6 is in the readable copy of the set structure. The function updateSubstate applies the effects of other operations on the substate: for *insert*(6) operations, it changes the substate to *true*, and for *delete*(6), the substate is changed to *false*. After inspecting the entire log, the predicting thread calls resultFromSubstate (which in this case is the identity function). The result of *delete*(6) should be *true* if and only if the element 6 was in the set.

Note that the prediction function's implementation is derived only from the shared data structure's semantics, and not from its implementation. For example, the prediction on a list-based set and on a tree-based set would use the same code. It is an interesting question whether the log synchronization framework fits every data structure ever invented, and the answer is probably not. We have found that many of the classic data structures in the literature (sets, maps, heaps, queues) can be supported.

Back to the right-hand side of Figure 3.2, the predicting threads call makeSubstate to extract the relevant information from the data structure. The substate has to be sampled carefully, as concurrent version changes can retrieve inconsistent data. Therefore, the makeSubstate function is called repeatedly until the version stabilizes. Then, for each operation in the log preceding their own, the predicting threads call updateSubstate on the substate object. Finally, they return the result obtained by a call to resultFromSubstate.

We now turn to read-only operations. Read-only operations do not need to queue themselves in the log, yet they must read from the readable copy of the data structure, that is, they need to call makeSubstate and resultFromSubstate. However, this is not enough to enforce sequential consistency. To provide sequential consistency a reading thread must consider the effect of all of its own modifying operations (write-only and/or read-modify-write) performed before the current read-only operation. We demonstrate this subtlety in Figure 3.3, illustrating the timelines of a modifying thread and a concurrent predicting thread. The right-hand side of the diagram depicts the current state of the operation log and the readable copy of the set. The predicting thread executes *delete*(6) and then *lookup*(6). Its delete operation (emphasized) is queued at the third slot in the log, following two other operations. Naturally, the *lookup*(6) call should return *false*, but since the readable copy of the set does not reflect the latest operations, the predicting thread would find 6 in the readable copy of the set and return *true*.

As the above example shows, in cases where sequential consistency is a requirement, a read-only thread must traverse the log up to its last queued modifying operation and predict by calling updateSubstate for each of these operations. Note that the log synchronization mechanism is designed in such way that read-only operations involve minimal overhead, as it is the premise of log-synchronization that these operations are the most common in many real-life applications.

3.2.3 Progress and Consistency Issues

Predictive log synchronization theoretically provides lock-free execution of read and write operations on the data structure. However, this lock-freedom property is somewhat weak since in some scenarios the number of steps a threads performs per operation increases unboundedly, although it is bounded for each individual operation. For example, when the modifying thread is delayed, the operation log length increases, while predicting operations are forced to traverse an increasingly long chains of operations.

The read-only operations' implementation as described above does not lead to linearizability, as reader threads scan the log only to the point of their latest modifying operation. In a scenario where a writer thread executes an opera-

tion entirely preceding the read operation, the effect of the earlier write may be ignored by the reader. In order to achieve linearizability, read-only operations must also be totally ordered and thus have to be logged as well. This would impose an all-predict-all scheme, making the amount of work performed increase by the number of threads, generating no speedup. We avoid this at the cost of meeting weaker sequential consistency condition [52]. This is because one of our main goals is to provide low-overhead reads, and appending every read operation to the log and processing it would significantly degrade performance. Sequential consistency is what concurrent programmers and hardware designers usually think of when they specify something as atomic. However, the drawback of sequential consistency versus linearizability is that composed linearizable objects are linearizable by definition, while composed sequentially consistent objects are not. This implies that when composing sequentially consistent objects on must go through the process of proving that the new algorithm is sequentially consistent, a drawback for the software designer.

In some data structures, there may be an additional data consistency issue. A predicting thread working on an old version would eventually discover that fact, but it could be too late if the thread was reading from the currently writable copy. In this case, reading partially modified data can lead to memory access violations and infinite loops. The simple algorithmic solution is to have the lock owner thread perform prediction instead of mutation until all readers still reading the older version are no longer active. Implementing this solution without introducing an overhead is not straightforward because of the need to minimize the amount of inter-thread communication. To do so, our algorithm has reader threads share their status by attaching a locally updated record to a global linked list, which is traversed by the modifying thread when trying to initiate a new session. Aged elements are removed from the list so as to keep the list length proportional to the number of active threads.

3.2.4 The Programmer's Interface

To perform a log-based operation on the data structure, the programmer instantiates an object of a subclass of Operation, shown in Figure 3.4. The Operation derivatives contain a field for the result, a boolean field signaled by the modifying thread when its work is completed, and a next pointer to the next

operation in the log. The programmer must implement four abstract methods, three of which were discussed in the previous subsection, and the fourth, executeOn is the traditional sequential implementation of that operation on the given data structure.

```
public abstract class Operation {
    Object result;
    boolean finished;
    AtomicReference<Operation> next;

    public abstract Object makeSsubstate(
        Operation head, Object ds);
    public abstract void updateSubstate(
        Object substate);
    public abstract Object resultFromSubstate(
        Object substate);
    public abstract Object executeOn(
        Object ds);
}
```

Figure 3.4: The Operation Class

In Figure 3.5, we demonstrate how the integer set would be implemented in the framework. The example consists of the IntSetSubstate class representing the substate generated during the prediction process, and the InsertOperation class which is instantiated when performing inserts (we left out trivial details such as constructors). The executeOn method simply calls the sequential implementation of insert. The method makeSubstate instantiates a new substate containing the parameter to insert, determining whether it is present in the data structure by calling the sequential find. The role of updateSubstate is to apply the effect of an insert operation on a given substate: if the parameter associated with the substate equal to the insert operation's parameter, the isFound field is set to *true*. The resultFromSubstate method returns the negation of isFound, as insert operations succeed if and only if the element was *not* in the set before.

```
class IntSetSubstate {
    public int value;
    public boolean isFound;
}

class InsertOperation extends Operation {
    int parameter;
    public Object executeOn(IntSet ds) {
    this.result = ds.insert(parameter);
    return this.result; // a boolean
    }
    public Object makeSubstate(
        Operation head, IntSet ds) {
    return new IntSetSubstate(
            parameter, ds.find(parameter));
    }
    public void updateSubstate(
        IntSetSubstate substate) {
    if (parameter == substate.value)
        substate.isFound = true;
    }
    public Object resultFromSubstate(
        IntSetSubstate substate) {
    return !substate.isFound;
    }
}
```

Figure 3.5: Example: Log-based Integer Set Insert

3.3 Implementation Details

In this section we give some of the details corresponding to the system design presented in the previous section. To simplify the presentation, we do not include some of the mechanisms, keeping the focus on the ones at the core of the PLS framework. The syntax used to present the code is of the Java™ programming language.

When a predictive log-synchronized data structure is created, an identical secondary structure is allocated and initialized as well, and both copies point to a mutual object of class Log. The bulk of the mechanism is implemented as part of the Log class.

The Log class, presented in Figure 3.6, consists of the following fields: version – the global version count, structures – pointers to the two copies of the data structures, mutex – the lock, and headPointers – pointers to the head of the log (One used by the readers and one to be used when the version is incremented. The two head pointers are initialized to point at a dummy operation node). Additionally, we keep a list of adjustments. The list records the changes to the writable copy to be applied to the other copy at the beginning of the next session.

After instantiating the operation, the programmer calls one of the methods readModifyWrite, read, and write, based on the type of operation at hand. The methods write and readModifyWrite append the operation to the log[3] and try to acquire the mutex in order to execute on the writable copy. If the mutex is taken, write simply returns where readModifyWrite calls predict. The read method simply calls predict.

The code for mutate, depicted in Figure 3.7, first adjusts the writable copy based on the latest changes applied to the other copy. Then, it traverses the log and executes each of the logged operations on the writable copy. Finally, it sets the new head pointer to point to the position in which it stopped traversing the log.

In Figure 3.8 we present the code for the predict method, which is less trivial. When executing predict, the thread must complete reading the substate from

[3] Appending to the log is done using the atomic primitive *Compare&Swap* on the last element in the log

the readable copy before the session ends and the version is incremented. If it fails to do so and the operation has not been completed by the modifying thread (for read-modify-write operations) it must retry. Upon success, in lines 11-16, the thread decides how far back in the log it has to traverse. When the operation is a read-modify-write one, the log is traversed up to the position where the operation was enqueued. For read operations, if there is a pending modifying operation requested by this thread (local is a thread-local object), the prediction should proceed up to that operation. If the last operation has completed, or in the case where sequential consistency is not a requirement, there is no need to inspect the log at all.

3.3.1 A Log-based Red-Black Tree Integer Set

An integer set implemented as a sequential red-black tree can be made concurrent using the PLS framework by creating appropriate InsertOperation, DeleteOperation, and LookupOperation classes. The InsertOperation class is shown in Figure 3.5, the other two are as simple. The executeOn method of these three classes would consist of sequential red-black tree code. At the beginning of every session, the writable copy of the data structure has to be updated with the same modifications made to the other copy during the previous session. This list of modifications, the *adjustments*, is easily constructed during their original execution and used by the modifying thread of the new session. Hence, when applying PLS on an existing sequential red-black tree, the only non-trivial programming task is the implementation of the operation classes.

In the red-black tree case, one can optimize the adjusting process by saving information from the first execution, such as pointers to tree nodes associated with the operation. All of the three set operations begin with a tree traversal to destination nodes that can be recorded. Each node in the tree can contain a pointer to its "twin" node in the other copy, which can be used as a shortcut when applying the adjustment instead of repeating the entire tree traversal.

```
public class Log {
    int version;
    Object structures [2];
    ReentrantLock mutex = new ReentrantLock();
    Operation headPointers[2];
    ArrayList<Adjustment> adjustmentList;
     ...
    public Object readModifyWrite(Operation op) {
        Object result;
        appendToLog(op);
        if (tryLock()) {
            result = mutate(op);
            release ();
            return result;
        }
        return predict(op, false);
    }

    public Object read(Operation op) {
        return predict(op, true);
    }

    public void write(Operation op) {
        appendToLog(op);
        if (tryLock()) {
            mutate(op);
            version += 1;
            release ();
        }
    }
     ...
}
```

Figure 3.6: The Log Class

```
private Object mutate(Operation op) {
    // ds is assigned to the writable copy
    Object ds = structures[1 − (version % 2)];
    for (adj : adjustmentList)
        adj.adjust(ds);
    adjustmentList.clear();
    Operation prev = headPointers[version % 2];
    Operation e = prev.next.get();
    while (e != null) {
        e.executeOn(ds);
        e.finished = true;
        prev = e;
        e = e.next.get();
    }
    headPointers[1 − (version % 2)] = prev;
    return op.result;
}
```

Figure 3.7: Code for mutate

```
  private Object predict(Operation op,
                                  boolean isRead) {
      do {
          oldver = this.version;
 5        savedHead =
               headPointers[oldver % 2].next.get();
          savedLastOpFinished =
               local.lastModifyingOp.finished;
          substate = op.makeSubstate(
10                      savedHead, structures[oldver % 2]);
          if (op.finished)
               return op.result;
      } while (oldver != version);
      if (isRead)
15        upto = (savedLastOpFinished
                    || noSequentialConsistency)
                  ? savedHead
                  : local.lastModifyingOp;
      else
20        upto = op;
      for (Operation e = savedHead; e != upto;
                                  e = e.next.get())
          e.updateSubstate(substate);
      return op.resultFromSubstate(substate);
25}
```

Figure 3.8: Code for predict

3.4 Performance

We present here a set of proof-of-concept microbenchmarks intended to show that there is merit to the predictive log-synchronized approach, but are in no way intended to be a full and comprehensive exposition of PLS performance. We chose a single specific data structure: a red-black tree data structure representing a set (used in benchmarking various transactional memory systems [29, 62, 44, 14]) and wrote it in a predictive log-synchronized manner.

We tested the red-black tree data structure in various configurations considered to be common application usage patterns. One of the parameters was the tree size, derived from the range from which keys are selected at random. In the "large" tree, we used a range of $0..10^6$ and initialized the tree by executing 10^5 insertions, while in the "small" tree we used keys taken from the range $0..200$, initialized by executing 100 insertions. After initialization, a varying number of threads execute randomly selected operations among *insert*, *delete*, and *lookup* of a random integer within the range. Each test was repeated five times; the data presented in this section is the average of those runs. For our experiments, we used a 16-processor SunFire™ 6800, which is a cache coherent multiprocessor with 1.2GHz UltraSPARC® III processors, running the Solaris™ 9 operating system.

In our first set of microbenchmarks we compared the performance of three techniques that provide simple programming of concurrent red-black trees.

Locks Coarse grained locking via the Java *synchronized* monitor.

OSTM A Java based implementation by Marathe et al. [62, 61] of the OSTM by Harris and Fraser [21], which is a representative state-of-the-art software transactional memory systems. Unlike the C-based implementation in [21], this implementation does not use a specially tailored closed memory system, rather, it uses Java's built-in GC. In the red-black tree code, it uses a different mechanism than the early release used by Fraser [21] to eliminate various tree hotspots. A comparison of OSTM to other software transactional memory systems can be found in [61].

PLS A Java based implementation using PLS.

In addition, in order to analyze the cost of prediction, we benchmarked a vari-

ation of the PLS algorithm in which threads only read the substate without predicting (the resulted implementation is not sequentially consistent.) We did not use a contention manager in the either the OSTM or PLS implementation. A contention manager is a heuristic software mechanism [83] that decides when to back-off in the face of access contention on the data structure.

Graphs (a) and (b) of Figure 3.9 illustrate the throughput in accessing a large tree. Graph (a) shows the results of an experiment in which operations were evenly distributed between inserts and deletes (no lookups), on a large red-black tree. When all of the executed operations modify the data structure, PLS has higher throughput than OSTM on low concurrency levels due to lower overhead, but from 16 threads and up, the limited parallelism of PLS degrades its performance. On the large tree, where keys are often located as deep as 16 levels below the root, the OSTM algorithm involves relatively large transactions and thus suffers from significant overhead, most likely due to memory management. In Graph (b), where the execution is dominated by lookup operations, PLS performs significantly better due to the fact that non-modifying operations involve very low overhead.

Graphs (c) and (d) depict the experimental results on a small set containing 100 elements on average. In Graph (c), we see both PLS and OSTM utilize machine parallelism, where OSTM does it successfully due to small transaction size, and PLS due to a significant portion of insert (or delete) operations that do not modify the set as their parameter exists (or is absent, respectively) in the tree. In the experiment whose results are shown in Graph (d), although the OSTM algorithm outperformed the lock-based one, PLS was faster by a factor of 2 to 3.

We note that our results for the OSTM implementation agree with those of [61] but are dramatically worse than those of the C-based implementation of Harris and Fraser in [21]. There are probably various factors responsible for the difference. Harris and Fraser used a different mechanism than [61] to perform early release of hotspots like the tree's sentinel nodes. They used their own specialized closed memory allocation mechanism, and their C-based implementation naturally has a lower level of indirection than Java code.

As another example of data structure design using PLS, we conducted exper-

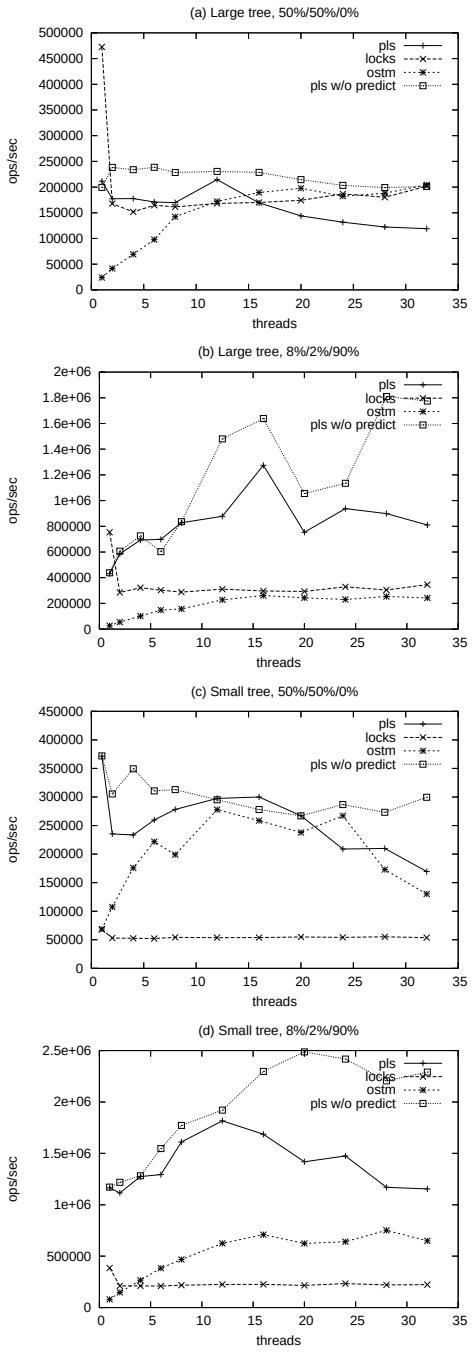


Figure 3.9: Throughput of Red-Black Trees Implemented Using PLS, OSTM, and Java Monitor Locks

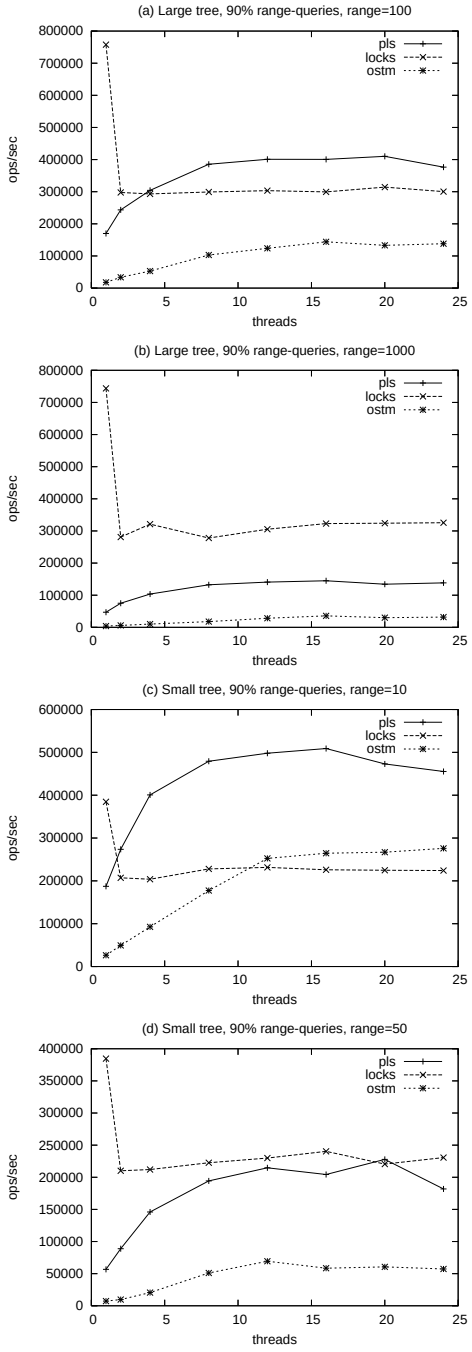


Figure 3.10: Red-Black Trees, 90% Range Queries, 8% Inserts, 2% Deletes

iments on a red-black tree supporting fixed size range queries[4]. We implemented range queries in PLS by maintaining the substate as a subset of elements which we update during the prediction process. The graphs in Figure 3.10 illustrate the the results of experiments where 90% of the operations were range queries, 8% inserts, and 2% deletes. On the small tree, we picked a range size of 10 (Graph 3.10(a)) and 50 (3.10(b)), and on the large tree we used 100 (3.10(c)) and 1000 (3.10(d)). The performance of both PLS and OSTM degraded with respect to the lock-based algorithm when the range size was increased, but for different reasons. In PLS, operations always succeed, but a source of extra overhead is the substate maintenance, while in OSTM the increased transaction size increases the chances of a transaction encountering a modification within its range, which causes more transaction failures.

The preliminary benchmarks for PLS performance on red-black trees suggests PLS as a viable approach as a method for utilizing concurrency. However, one must remember that red-black trees are the classical example of a data-structure that benefits from the predictive approach since it is inexpensive to compute the results of operations from the log. For example, we implemented a *heap* data structure on the PLS framework, supporting *insert*, *deleteMin*, and *getMin*. The prediction process of *deleteMin* and *getMin* operations required maintaining the set of keys inserted and removed during the current session. The amount of computation overhead in that case was unacceptable and the overall throughput measured was significantly lower than the one of a simple coarse-grained locking approach. A heap data structure is an example of a data structure for which the PLS technique is a bad fit, mostly because the effect of *deleteMin* is difficult to determine by inspecting its arguments. One cannot predict the results of operations succeeding a *deleteMin* without performing some sort of non-trivial emulation.

Clearly more work is needed to adapt contention management mechanisms, and further evaluation is necessary to understand the benefits and drawbacks of PLS for the design of various data structure classes.

[4]A range query is an operation that extracts a subset of keys within a given key range

3.5 Conclusion

In this chapter we introduced predictive log-synchronization, a new concurrent programming paradigm that may be viewed as conceptually contradicting the accepted approaches to parallelization such as fine grained locking and transactional memory. These approaches aim to optimize parallelism when accessing sets of disjoint locations in memory, while PLS optimizes parallelism in reading from memory, at the cost of not providing parallelism among modifications to disjoint locations. Moreover, PLS is based on using a single coarse-grained mutual exclusion lock, a construct traditionally held accountable for limited scalability.

The proposed solution is not claimed to be asymptotically scalable: the throughput of modifying operations is technically bounded by the throughput of a single processor. However, one of our principal conjectures is that the distribution of high level operations in real-world applications is read-dominated. Unlike other paradigms, the PLS framework does not require parallel reasoning abilities from the programmer, but does oblige the introduction of supplementary code for the semantic specification of each type of data structure.

We believe that the PLS framework, presented here in its initial form, has the potential of being an interesting alternative to lock monitors and software transactional memory mechanisms in a variety of circumstances.

Chapter 4

Address-Based Log Synchronization

4.1 Introduction

In Chapter 3 we saw the Predictive log synchronizer, PLS, which is a general solution for synchronizing high-level operations on a shared data structure. PLS is highly optimized for read dominated workloads, but its main drawback is requiring additional specialized, though sequential, code to be provided by the programmer. In this chapter, we aim to overcome this drawback by introducing the *address-based log-synchronizer*.

The *address-based log-synchronizer* is actually a novel mutual-exclusion based coordination mechanism, especially fit for object oriented languages such as Java™.

4.1.1 Address-Based Log-Synchronization Fundamentals

In a nutshell, the address-based log-synchronizer replaces the monitor structure of a given object. Instead of using a coarse-grained locking approach, it detects conflicts on the address level entailing minimal overhead for read dominated workloads. The log-synchronization fundamentals are:

- **Virtual execution:** To execute an operation that modifies the shared object, threads do a "virtual" execution: they record their loads and stores in a *request* structure without actually storing values, and then append the request to a global *log*.
- **Shared field augmentation:** All shared object fields, that is, ones potentially accessed by more than one thread, are augmented at compile time by a *version* number and a *mark* flag. Object fields are classified as shared either manually using a special keyword or automatically by the compiler. The manual approach would require a language enhancement, while the automatic one might not be optimal identifying shared fields.
- **Request handling:** Threads may try to acquire a global *lock* and execute the operations in the log on behalf of the requesting threads. Requests are processed by first detecting conflicts in their read-sets and then, if none detected, applying the recorded stores. When applying stores, the associated version fields are updated to the value of a global *version count*.

The version count is increased to this value immediately after handling the request.

- **Early conflict detection:** Read address conflicts can be detected as soon as they are appended to the thread's read set. Before virtually executing operations, threads record the value of the global version count. After each load (memory read), the associated version is compared to the recorded version count, and if newer, the operation fails.

- **Early completion:** For each address in the write set, the requesting thread turns on the associated *mark* flag before appending the request to the log. These marks are erased after the request is processed. When the global lock is occupied, it may be possible for a thread to successfully complete as long as the following conditions are met: (i) All read-set versions are older than the version recorded before the virtual execution. (ii) None of the read-set addresses are marked.

- **Lightweight non-modifying operations:** When the operation executed is a *non-modifying* operation, there is no need to explicitly validate the read set after the virtual execution, as conflicts are detected on the post-load version validations. This property makes the LS framework impose very small overhead costs on reading-only operations.

- **Failure and retries:** Failure occurs when the read set has changed during the operation's virtual execution. Upon failure, the thread may retry or aggressively acquire the lock to guarantee a successful execution.

Address-based log-synchronization thus provides parallelism in two ways, by performing all non-modifying instructions of a given method in parallel, reducing the operations on the shared lock's critical path to essentially the modifying ones, and by allowing read-only operations to proceed concurrently in an uninterrupted fashion.

The idea of logging lists of operations on an object to be executed by the thread owning the lock was originally proposed by Oyama et al [74] as a low overhead means of increasing locality in performing modifications to shared locations within a mutually exclusive section. However, Oyama et al do not use the log as a means to arbitrate among conflicting concurrent read and write requests.

They also do not have any of the other elements of log-synchronization: multiple copies, concurrent reading, and arbitration among concurrent modifying threads. The log-synchronizer utilizes a similar idea to build a new mechanism that allows a thread holding the object lock to play the same role played by the shared bus in a hardware transactional memory scheme [39]: it arbitrates among concurrent high-level method calls, deciding which ones can be completed because they do not overlap with modifications made by others. As we show, having a single thread decide the outcome of modification requests to the data structure in a centralized manner allows for a lower overhead mechanism than that of non-blocking STM implementations, where threads must be ready to commit and/or roll-back other threads transactions to allow progress.

Finally, as mentioned above, the log-synchronizer, because it is based on a single mutual exclusion lock, can interact with hardware transactions at the cost of only a single additional cache line per hardware transaction. It can also execute any method calls with I/O or nesting without limitation by executing them in master mode.

4.1.2 Address-Based Log-synchronization vs. PLS and STMs

The *address-based* log-synchronizer differs from the *predictive* one (PLS, described in Chapter 3) in the following major attributes: first, the data structure is not wholy duplicated, but only selected fields of shared objects in it. Second, the granularity of operations is as low as memory accesses, rather than high-level operation. Third, there is no need for prediction as well as extra code specially written. Modifying threads block until their memory accesses are executed by the lock-owning thread.

The address-based log-synchronizer is a mechanism for executing sequences of operations, or transactions, atomically in software. However, we believe that it should not be classified as an STM for the same reasons that monitors, which also fit the above definition, are distinguished from STM mechanisms.

Though there is no formal definition of what constitutes an STM mechanism, there are fundamental differences between all known STM implementations and log-synchronization both on a conceptual level, on the level of the supported semantics, and in the core algorithmic paradigm.

We explained the conceptual differences in Chapter 3.

In terms of its semantics, like monitors and unlike STMs, the address-based log-synchronizer easily supports unconstrained nesting and I/O within method calls. Support for I/O and nesting in STMs remains an open problem [72].

On an algorithmic level, STMs are distributed coordination mechanisms while the log-synchronizer is centralized one. They relate somewhat like peer-to-peer and client-server in that they are fundamentally different but have things in common. The log-synchronizer has a single thread arbitrate and execute the modifications on behalf of all others, while software transactional memory protocols have all threads potentially commit, abort, or rollback modifications of other threads they conflict with. This fundamental "centralistic" algorithmic property of log-synchronization eliminates much of the complexity and coordination traffic that is inherent to STM implementations, and reduces the need for complex contention-management algorithms [26, 73].

For example, being centralized allows the single thread performance of the log-synchronizer to be very close to that of a mutual exclusion lock, even without the introduction of a fast-path mechanism. The single thread performance of the STM we tested was two orders of magnitude slower than a lock, and is not clear how one could get good single thread performance out of a distributed STM mechanism.

In many scenarios, the address-based log-synchronizer can provide parallelism with a significantly lower overhead than the STM constructions [5, 21, 44, 61, 62, 14, 88, 93]. Because it is essentially a lock executing in a mutually exclusive mode, the log synchronizer easily overcomes issues of nesting and I/O that are not addressed by current STMs, has an efficient fast-path mechanism, and a clean low overhead interface to hardware transactions, one that will allow it to smoothly extend transactions to arbitrarily large method calls.

To be used as a replacement for monitors, the address-based log-synchronizer does not need explicit user controlled transactions. The only requirement is that either the programmer or the compiler will identify the subset of a given object's fields that are potentially shared by multiple threads.

Address-based log-synchronization does suffer from the usual limitation of lock-based schemes, that it is blocking: a long delay of a thread holding the

lock will cause a similar delay in the program as a whole. Nevertheless, given the experience with current lock-based software it is our belief that this is not a significant drawback, and will become less important in the future as operating systems evolve to better deal with concurrency and the multi-core nature of hardware.

Finally, we note that in this chapter we describe the address-based log-synchronizer as a replacement for a monitor protecting a given concurrent object. This is a simple programming interface to use for parallelization. However, there is no inherent reason why a log-synchronizer could not be used as a general mechanism to support transactions across multiple objects or as the basis of a transactional programming language.

In summary, we believe that address-based log-synchronization provides a simple low-overhead alternative approach to the software transactional memory paradigm for parallelizing concurrent data structures.

4.1.3 Performance

We present a set of proof-of-concept benchmarks to show that there is merit to the log-synchronized approach, but are in no way intended to be a full and comprehensive exposition of log-synchronizer performance. Such an exposition would consist of an automatically generated log-synchronizer code using a compiler and within a given JVM. Instead, we chose a single specific data structure: a red-black tree data structure representing a set (used in benchmarking various transactional memory systems [21, 62, 14]), implemented it in *C* using "off-the-web" sequential red-black tree code, and manually added C-level log-synchronizer code. We did not add compiler based features that we believe would have significantly enhanced performance. Even so, on a typical benchmark our log-synchronized red-black tree has 2 to 3 times the throughput of a coarse grained lock-based solution and about 20 times the throughput of code transformed via Harris and Fraser's OSTM [21] state-of-the-art software transactional memory. Even without any read-only operations, our log-synchronized tree has twice the throughput of the lock-based tree and 5 times the throughput of the tree transformed via OSTM. The single thread performance of the log synchronizer is close to that of a lock even without a fast-path mechanism, while that of OSTM is two orders of magnitude slower. The ques-

tion of how to add a fast path mechanism to STM algorithms remains open.

In the remainder of this document we begin by presenting the language modifications needed to provide an efficient log-synchronizer based replacement for a monitor. We then show how based on these extensions one can implement log-synchronization in software. We explain how to deal with I/O and nesting, how to add a fast path mechanism to log-synchronization, and how log-synchronization can be added with very low overhead as an extension mechanism for hardware transactional memory to allow large transactions. Finally, we present our proof-of-concept empirical results.

4.2 A Log-Synchronizer Implementation

4.2.1 Log-Synchronized Objects

A log-synchronized object is associated with a *lock*, a *log* of operation requests, and an *object version*. Basically, threads compose requests, append them to the log, and may try to acquire the lock. While holding the object's lock, a thread processes the log and increments the version on each request. Requests are composed by *virtual execution* and processed in accordance to the *conflict detection* mechanism. Prior to executing the operation, threads record the object's version as their *starting version*, which is later compared to the versions of the data to detect conflicts. Reading-only operations are not required to log their requests as they are able to detect conflicts independently.

The compiler automatically modifies the log-synchronized object and its substructures by augmenting their *shared* fields with two additional words: a version and a mark word. Shared fields, which are fields having the potential to be accessed concurrently, can either be determined automatically (but conservatively) by the compiler, or by manual programming language declaration. The version word associated with shared fields enables us to detect occurances of recent changes to memory locations. We use the mark word as a fine grain lock, which is acquired for short periods of time, mostly during the time the owning request is in the log.

4.2.2 Virtual Execution

One of the core ideas behind log-synchronization is the *virtual execution* of concurrent code. It is the responsibility of the compiler to apply a transformation on the code of log-synchronized methods, generating the virtual execution code.

It is clear why store instructions must be replaced, but load instructions also need to be transformed to support the conflict detection mechanism. Store instructions are replaced by code recording the to-be-written address and value in a yet local buffer. Loads are augmented with code recording the addresses being read from in a separate buffer. Additionally, immediately after each load

instruction, the thread's *starting version* is compared to the field's version, and if older, the operation fails.

The fact that stores are not being actually applied must not be observable by load instructions of the same virtually executing thread. In the case a memory location is written to and later in the operation read from, the returned value must be the latest one written. Load insructions should therefore look up the read address in the threads' write set records and return the latest value written. Since performing such a lookup on every load would significantly affect performance, we use a probabilistic bit vector (known as Bloom filter [8]) to rule out the presence of addresses in the write set on the common case.

4.2.3 The Log

Threads queue in the object's log operation requests consisting of the read and write sets collected during the virtual execution. Requests also include a status flag initially set to *pending*, and the thread's *starting version*. The log is implemented as a non-blocking linked list: new elements are inserted at the tail using an atomic primitive such as CAS (Compare&Swap), and the oldest element is removed by advancing the head pointer. Note that the log tail may be accessed concurrently while its head is always modified by the lock owning thread.

Once the object lock is acquired, the thread processes the requests in the log by inspecting their read sets and the current values of their corresponding per-field versions. It decides whether the operation should succeed by applying a simple conflict detection algorithm. Each non-conflicting request is applied by iterating over its write set, storing values to addresses and setting the associated version values to the value of the soon-to-be object version, which is incremented immediately after processing the request. The request status is atomically set to *succeeded* or *failed*.

4.2.4 Conflict Detection

Essentially, conflicts arise when an address from one pending operation's read set is present in another operation's write set whereas the latter precedes the former in the log. Since requests are handled in the enqueue order and the

object version is incremented for each processed request, conflicts can be conservatively discovered by simply comparing the versions associated with the read set to the thread's starting version.

Conflicts can often be detected in this manner as early as in the virtual execution phase, since every load instruction is augmented with the validating condition. Another reason for post-validating loads is to avoid reading partially committed states (even within a doomed-to-fail operation,) which in some cases could cause memory access violations and infinite loops.

The above conflict detection mechanism would suffice unless the *early completion* functionality was involved in the log-synchronization process. When we allow operations to complete independently of preceding operations in the log, we must provide them with an efficient way to synchronize on whether operations succeed. There are two approaches to do that: one, the "prediction" approach is to traverse the log simulating the effect of each preceding operation on the request's read set. In Chapter 3, we use this prediction approach, but we applied it there on high-level operations rather than on read and write sets. Our experience with the prediction approach has led us to believe that on medium to large size read and write sets, the prediction overhead would be significantly greater than its benefit.

We base our solution on the other approach, according to which every memory location in a thread's write set is locked by being *marked* with a unique identifier during the entire period it resides in the log. When a thread identifies one of its read addresses as marked, the operation fails. Since the requesting thread and the log-executing thread must report an identical result for the request, they set the request's state to *suceeded* or *failed* atomically. The fine-grained locking approach is scalable as threads can usually make progress independently of others, unless there is a true write-write or read-write conflict.

Prior to enqueuing a request in the log, a thread traverses the addresses in its write set and locks them by marking them with a *LockId*. The LockId is a per-request unique value composed of the thread id and a private increasing counter. Only when a request leaves the log, the lock owner thread releases the locks by setting them all to zero.

When all requests in the log have their write addresses locked, a thread is able to eliminate the possibility of a conflict with its read set by simply verifying

that none of the read addresses are marked by other threads. This way, an operation can succeed without needing to simulate the entire history of loads and stores in the log.

Practically, atomically marking a set of addresses one by one can have a destructive effect on performance. Atomic primitives usually impose bus traffic and micro-architecture related penalties. We propose to perform the series of address markings by using a mutual exclusion method based on reads and writes only, such as the one proposed by Lynch and Shavit in [60]. According to the algorithm in [60], threads achieve mutual exclusion by operating on a pair of memory locations: they write to the first one, and then proceed to the second only after validating the first value. As the pair of memory locations we use the two halves of the mark word. In most cache coherent multiprocessors it is necessary to apply a memory barrier before validating the first store, which is often equivalent in cost to atomic primitives. However, rather than apply the two-location mutual exclusion method individually on multiple locations, we perform the first writes as a batch, then apply a **single** memory barrier, and finally do the second writes as a batch. This way, the amount of bus communication is independent of the write-set size.

4.2.5 Early Completion

When a thread completes its virtual execution, queues a request, but then fails to acquire the object lock, it does not necessarilly need to wait until its request is processed. Instead, it traverses its own read-set verifying that none of the addresses are marked by a different LockId. If none of the read-set addresses are marked, it is guaranteed that none of the requests in the log conflict with the read set, and the operation may succeed. The thread then atomically sets the request's status to *succeeded*,

4.3 Language Extensions

We illustrate the log-synchronizer using the monitor based implementation of the *synchronized* construct of the Java™ programming language. To provide an efficient implementation of log-synchronization, we slightly modify the pro-

gramming language. We introduce a new method modifier `logsynchronized`, which enables the log-synchronized mechanism for the specified method, in the same way that `synchronized` enables a monitor. More importantly, we add a new member field modifier called `shared`, used by the programmer to distinguish fields that should be managed by the new mechanism. The appearance of this modifier causes the compiler to allocate additional space for the field's values of version and mark, and to augment load and store instructions accessing it. The two The code in Figure 4.1 illustrates how the new keywords are used when implementing a stack.

```
class Stack {
    shared ListNode head = null;
    class ListNode {
        public int value;
        shared public ListNode next;
    }
    public logsynchronized void push(int newval) {
        ListNode newNode = new ListNode(newval, head);
        head = newNode;
    }
    public logsynchronized int pop() {
        int val = head.value;
        head = head.next;
        return val;
    }
}
```

Figure 4.1: Example: a log-synchronized Stack class

We note that it is possible to extend the compiler to determine shared fields automatically in a conservative manner, avoiding the need to declare them manually using the `shared` keyword. In practice, the compiler might mistakenly consider some fields as shared while they are not, yet this conservative approach would free the programmer from the burden of declaring variables as shared and would allow modification of the Java™ virtual machine to support log-synchronization without changing the language itself. We note however that one must be careful when applying such transformations since the log-synchronizer is not equivalent in function to a monitor as it cannot be used to implement mutual exclusion on objects outside the synchronized block.

4.4 Performance

We present here a set of preliminary proof-of-concept microbenchmarks intended to show that there is merit to the log-synchronized approach, but are in no way intended to be a full and comprehensive exposition of log-synchronizer performance. Such an exposition would consist of an automatically generated log-synchronizer code using a compiler and within a given JVM. Instead, we chose a single specific data structure: a red-black tree data structure representing a set (used in benchmarking various transactional memory systems [21, 62, 14]), implemented it in *C*, and manually added C-level log-synchronizer code. This crude implementation approach works against the log-synchronized implementation since its code is clearly less efficient than in-lined assembly code.

We compared our log-synchronizer to a coarse lock-based *C* implementation and to Harris and Fraser's state-of-the-art OSTM software transactional memory system [21]. According to the experiments conducted in [62], the OSTM scheme performed well on red-black tree benchmarks. Moreover, according to [62] OSTM is better suited for read dominated workloads than other STMs. Since the OSTM code was only available to us in Java™ and the log-synchronizer in *C*, we also implemented the coarse lock-based red-black tree in Java™ and used the relationship between the two lock-based implementations as a way to legitimize the comparison.

We compared the following four versions of a red-black tree:

- *C* implementation of a *log-synchronized* red-black tree where slave and reader threads *poll* the current session number after each load.
- *C* implementation of a coarse mutual exclusion based red-black tree.
- Java™ implementation of an OSTM based red-black tree.
- Java™ red-black tree with coarse locking.

In our experiments, the data structure was initialized with 10000 or 100 insertions of integers within the range of 0 to 10^6 or 200. Then a varying number of threads were intensively executing randomly selected operations among *insert*, *delete*, and *lookup* of a random integer within the same range. Each test was run five times. The data presented in this section is the average of those runs,

with a standard deviation usually measured below 3%. For our experiments, we used a 16-processor SunFire^{TM} 6800, which is a cache coherent multiprocessor with 1.2GHz UltraSPARC® III processors, with the Solaris^{TM} 9 operating system installed.

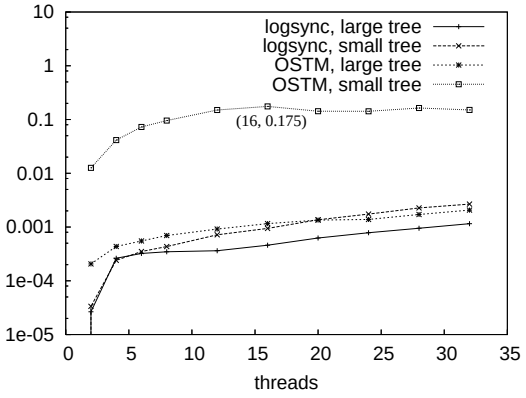


Figure 4.2: Transaction Failure Rate

We begin by considering the failure rate of the log-synchronized and OSTM transactions, in order to better understand the nature of our red-black tree benchmarks. The benchmark in Figure 4.2 shows the failure rate when method calls are split equally between inserts and deletes. As one can see, even with no read-only lookup method calls, we measured a rate of less than 0.2% log-synchronized slave failures in all of the experiments on the log-synchronized red-black tree. In the small tree experiment, OSTM failed significantly more frequently, but the absolute rate of transaction aborts was still low. Red-black trees are thus good candidates for transactional methods to outperform locks since they have highly distributed access patterns.

Graph (a) in Figure 4.3 illustrates the throughput of the log-synchronizer, OSTM, and two lock algorithms when the operations are evenly distributed between inserts and deletes (no lookups) on a large tree. As can be seen the log-synchronized tree allows more concurrency than the lock-based tree because slave threads are able to make progress while the master holds the lock, and thus the log-synchronized solution shows scaling up to 16 threads (the number of processors on the machine). What is happening is that in the log-synchronizer methods are increasingly executed by the slaves concurrently with the master, each

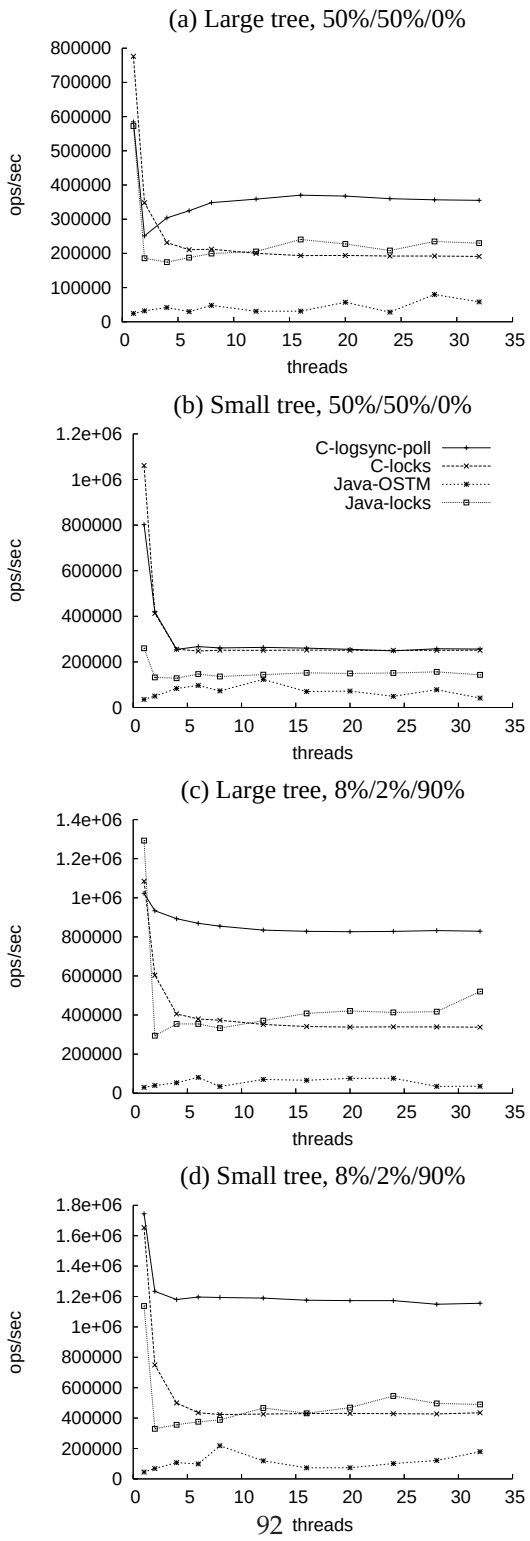
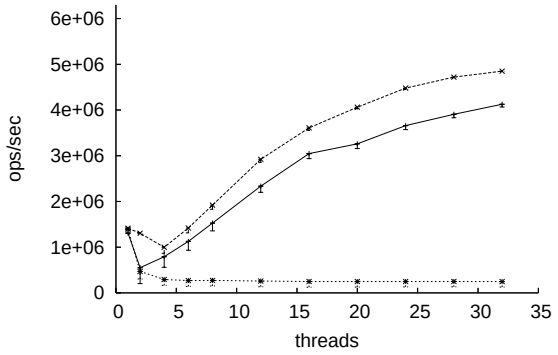


Figure 4.3: Throughput of Log-Synchronized and Lock-Based Red-Black Trees

(a) Large tree, separated threads



(b) Small tree, separated threads

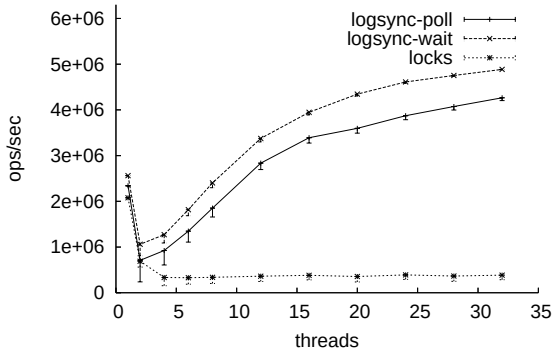


Figure 4.4: Throughput on Separate Reader and Writer Threads

running through the code, executing all branches and stack calls, creating a log entry of the resulting loads and stores. The master then makes a pass through the log, verifying the loads and writing the stores, without having to perform any of the branches and stack calls. The verification itself is low cost because of caching locality [74], which means that the only costly operations by the master are the actual stores. In short, we have cut down significantly on the operations in the critical path, which is why the log synchronizer has twice the throughput of the lock-based solution. As expected, even without conflicts, the OSTM performance is abysmal because of the overhead of acquiring and releasing transactional control over memory locations. The graphs clearly demonstrate that as the concurrency level rises, the OSTM throughput improves, but it is always an order of magnitude below the lock-based versions' throughput.

The performance gain of having slave threads working in parallel to the master is further demonstrated in Figure 4.6, where we added to the comparison on the same large tree, 50%/50%/0% benchmark a version of log-synchronization in which all operations are executed in master mode. Figure 4.6 also provides a measure of the log-synchronization overhead, captured by the gap between the graphs of the C based lock algorithm and the no-slaves log-synchronizer throughput.

In Graph 4.3(b) we ran the above test on a smaller tree. The allowed range of elements in the tree was $[0, 200]$, generating a relatively small structure where the log-synchronized version behaved quite differently. The frequent polling by the log-synchronizer had a negative effect on performance, which on small data structure supersedes the benefits of log synchronization. The result was that the polling log-synchronizer's performance was no better than that of the lock-based one. We will touch further on this topic in a moment.

It is accepted folklore that the real world usage patterns of most real-world applications using parallel access data structures are mostly characterized by read-only operations. Graphs (c) and (d) in Figure 4.3 depict the throughput of the algorithms with a read-only oriented distribution: 8% inserts, 2% deletes, and 90% lookups. Since log-synchronization allows read-only operations to execute concurrently with modifying operations, the performance gain is significantly larger, up to three times that of the lock based solution. However, because threads select the operation type uniformly, one out of ten operations is an insert or a delete, which is heavier and "blocks" the thread from contin-

uing to the subsequent lookup operations. Nevertheless, the log-synchronized tree has up to 20 times the throughput of that based on OSTM even though OSTM is suggested in the literature as a software transactional memory algorithm best geared for handling read-only transactions.

We benchmarked another usage pattern of concurrent data structures in which there are separate threads for reads and writes. We ran the experiment that generated the data of graphs (a) and (b) in Figure 4.4, this time with even numbered threads executing lookup operations only and odd numbered threads selected randomly between inserts and deletes. The graphs clearly show that log-synchronization enables intensive read operations while another thread is modifying the shared structure. The short bars marked below each data point stand for the portion of modifying operations (i.e. inserts and deletes). The continued rise in throughput beyond 16 threads is explained by the change in operation distribution: the number of modifying operations executed decreases with parallelism and since lookups are cheaper than inserts and deletes, the total amount of executed operations increases.

We return now to the limitations of the polling log-synchronizer on small data-structures. We implemented a "waiting" (as opposed to "polling") version of log synchronized red-black tree in which the master thread waits for old slave and reader threads to complete their old operations before beginning a session, rather than having them poll the session number on each load. This waiting scheme performed better than polling on the small tree with 50%/50% distribution. The graph in Figure 4.5 demonstrates waiting versus polling in that particular case. The waiting log-synchronizer did not do as well as the polling one in any of the other benchmarked distributions.

Finally, we note that even without a fast-path mechanism, the single thread performance of the log-synchronizer is close to that of the locks in all benchmarks. In summary, our empirical results suggest that log-synchronization has low overhead, and the potential of introducing parallelism in two ways: by allowing slaves to do part of the work concurrently with masters, and by allowing very low overhead concurrent read-only operations. There is much room for improvement here and we are already working on new versions of a log-synchronizer that overcomes some of the drawbacks suggested by the above experiments.

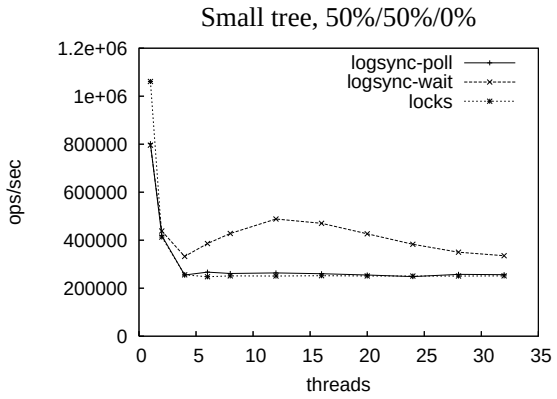


Figure 4.5: Waiting for old readers.

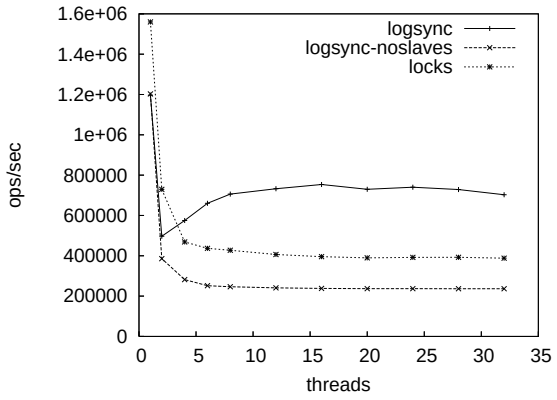


Figure 4.6: Effect of Slaves on Throughput

4.4.1 Other Related Work

In [64], McKenney and Slingwine introduced the idea of "read-copy update" (RCU). In RCU, the writer thread modifies copies of usually small objects while concurrently reading threads are guaranteed to be using a quiescent state of the shared data structure in a manner similar to that suggested by Herlihy [36]. That way, readers avoid the need to acquire a lock, enabling greater reading throughput. RCU does not handle efficiently cases involving complex objects, as these objects need to be copied as a whole on each modification. Additionally, unlike log synchronization, RCU is not designed to improve write throughput.

Strom and Auerbach [89] proposed a technique in which optimistic reader threads post-validate their operations using version numbers, rather than lock the data structure. Both the optimistic readers and the log-synchronization schemes involve solving the problem of readers working on an inconsistent state of the data structure.

Welc et al [93] have proposed *transactional monitors*, a technique for parallelizing monitor access in languages like Java™ by building an STM for transactions defined by the methods accessing the monitor. As we suggest here, the log-synchronizer can be used in the same way to provide parallelization via an implicit transactional interface as proposed in the transactional monitor, an interface defined by method calls and preserving the same semantics as a monitor. It can also use compiler support to automatically inject the additional log-synchronization code into the basic set of operations of a given method, an approach used in [93]. However, the similarities end here. The transactional monitor is essentially an STM mechanism [62, 61, 88], and as an STM, the transactional monitor offers very limited concurrency (compare to [62, 14]), since it determines read-write conflicts using a globally shared bit-mask stored in a single word, implying cache traffic and a very high frequency conflicts.

Moore et al. [71] recently proposed "Log-based transactional memory" (LogTM), which is a cache protocol based (hardware) implementation of transactional memory in which each thread logs loads and stores in a special per-thread region of virtual memory, used for transaction conflict detection and rollback. Apart from being a hardware scheme, it differs conceptually from log-synchronization in that it is distributed and there is not a one global log used for coordination.

Chapter 5

Transactional Locking

5.1 Introduction

In this chapter, we define the properties of a practical TM as we understand them, and propose two algorithms based on fine-grained locking, in an attempt to achieve the goal of a truly cheap, efficient, and easily adaptable TM implementation.

5.1.1 What makes a practical TM?

We have identified a set of properties that can be considered essential for the design of truly practical transactional memory implementations.

Mechanical code transformation Concurrent programming frameworks should not require exotic or non-trivial programming skills of application writers to operate efficiently. They should support "mechanical" methods to transform sequential or coarse-grained lock-based code into concurrent code. By mechanical we mean that the transformation, whether done by hand, by a preprocessor, or by a compiler, does not require any program specific information (such as the programmer's understanding of the data flow relationships.)

Availability We believe that in order to make efficient concurrent programming ubiquitous, it must not depend on special hardware constructs such as hardware transactional memory (HTM). Therefore, our focus is on software solutions.

Low overhead Emulating atomicity in software can involve significant overhead costs on transactional operations. In many cases, the source of overhead is additional levels of indirection when accessing memory, which are used for distinguishing memory locations under active transactions. In some cases, these overhead costs may overtake the benefits of the avoiding coarse-grained locking. Since workload of real-world applications commonly consists significantly more reads than writes, minimizing overhead on transactional reads is more critical than on writes.

Invulnerability to inconsistent states Some transactional memory implementations must actively avoid execution scenarios in which a thread can

view inconsistent memory state, i.e. a state which would be impossible if transactions were absolutely atomic. Although the thread's transaction may be doomed to fail, the execution may be vulnerable up to the next validation. During that period, unexpected behavior such as incorrect addressing and infinite loops is not improbable.

Another type of inconsistent state view, called "flickering", is a side-effect of STMs taking the update-in-place approach, according to which the new value is stored at the actual memory location rather than in a write-buffer. When a transaction aborts, the store is undone, but the temporary appearance of an irrelevant value in the shared memory could be under some circumstances at odds with the memory model.

Compatibility to memory allocation mechanisms Common data structures support operations that involve dynamic disposal of memory such as nodes in linked lists or trees. The actual freeing of memory must not occur while active transactions may still be accessing it. Some transactional memory implementations require specialized allocators and/or a separate heap in order to protect shared objects from being freed while used in active transactions. Some implementations rely on automatic garbage collection.

In addition to the above, we have identified two attributes of STM implementations which we consider "overrated". By compromising on these two attributes we were able to reduce overhead costs.

Non-blockingness The non-blocking approach (i.e. lock-free, obstruction-free) has gained popularity as it implicitly guarantees progress. A recent paper by Ennals [20] suggested that on modern operating systems deadlock avoidance is the only compelling reason for making transactions non-blocking, and that there is no reason to provide it for transactions at the user level. We agree with Ennals in the sense that non-blocking STMs can use schemes such as the Solaris `schedctl` to approximate non-blocking behavior by deferring preemption while holding locks.

Low transaction failure rate The overall performance of transaction-based execution is dependent on the failure rate of transactions. All transactional memory schemes are designed to allow parallelism, some are more conservative than others and fail transactions that might have completed

successfully otherwise. In various designs, the period by which transactions are considered overlapping differs. We have empirically found that the effect of differences in the benchmarked STMs' failure rates was less significant than the effect of overhead. One of the reasons for low failure rate is the naturally short duration of transactions. Since programmers are aware of the benefits of holding locks for time as short as possible, transactions derived from lock-based algorithms are naturally short.

5.1.2 Qualitative Comparison of STMs

There are currently proposals for hardware implementations of transactional memory (HTM) [39, 77, 4, 27], purely software based ones, i.e. software transactional memories (STM) [20, 34, 42, 44, 62, 14, 82, 88, 93], and hybrid schemes (HyTM) that combine hardware and software [5, 14].[1] Implementations depending on specialized hardware are mostly focused on supporting unbounded transaction size and duration, while the software based ones focus on reducing the overhead associated mostly with unavoidable data access indirection and conflict detection.

Since we propose a software-only scheme, we bring here a brief discussion on the design choices of existing STM implementations with regard to the new algorithms presented later in this chapter.

In Table 5.1 we provide a qualitative comparison of the STMs based on several design attributes. The first one is *blockingness*: DSTM, RSTM, OSTM, and WSTM are all non-blocking. DSTM and RSTM provide a slightly weaker progress guarantee, "obstruction-freedom", which means that any thread would complete in a finite number of steps when it is given enough time to operate in isolation. Obstruction-free algorithms combined with a contention manager in order to guarantee a reasonable progress guarantee. Ennals [20] argues that for transactional memory implementations, the only benefit in taking the non-blocking approach is deadlock freedom, and this can be achieved in various other ways.

The second column of Table 5.1 deals with the *acquire timing*, the transaction stage in which the written object is publicly associated with the transaction.

[1] A broad survey of prior art can be found in [34, 61, 78].

STM	blocking	acquire	validation	contained	gran.	mem. man.
DSTM	obs. free	encounter	incremental	no	object	custom/GC
RSTM	obs. free	encounter	inc./visible reads(*)	yes	object	custom/GC
OSTM	lock free	commit	amortized(*)	yes	object	custom/GC
WSTM	lock free	commit	amortized(*)	yes	word	custom/GC
Ennals, McRT	blocking	encounter	amortized(*)	yes	object	custom/GC
TL1-PO	blocking	commit	amortized(*)	yes	object	custom/GC
TL1-PS	blocking	commit	amortized(*)	yes	stripe	ordinary
TL2-PO	blocking	commit	immediate	no	object	custom/GC
TL2-PS	blocking	commit	immediate	no	stripe	ordinary

Table 5.1: Qualitative Comparison of Existing and Proposed STMs

Encounter-time acquire (or "eager" acquire in some places) means that objects are included in the shared transaction data upon the first access to them within the atomic block. *Commit-time* acquire (or "lazy" acquire) means that objects are only added locally to the transaction's read-set and write-set, and are exposed as a batch only close to commit. On one hand, when acquiring at commit-time, conflicts may be detected relatively late. On the other hand, some read-write conflicts are avoided as the reader is not aware of the writer during most of the writer's transaction. Typically, commit-time will use a speculative write-buffer, while encounter-time will use update-in-place with an undo-log and roll-back. The speculative write-buffer requires that reads look-aside into the write-buffer to avoid read-after-write hazards.

STM implementors have taken different approaches when dealing with the requirement that threads can avoid or recover from viewing inconsistent memory states, even during execution of to-be-aborted transactions. In DSTM, transactions are validated on each inclusion of a new object in the write-set. The cost of this *incremental validation* is thus $O(n^2)$ where n is the write-set size. Another approach is having threads register themselves as reading on every location they read from, i.e. being instantly abortable *visible readers*. However, this affects throughput by causing a higher cache invalidation rate as readers now modify shared data. Additionally, instantly aborting a transaction is non-trivial, requiring periodical checkpoints compiled into the code.

A popular solution for inconsistent state vulnerability is aimed at the symptoms, which are infinite loops, illegal memory accesses, and other exceptions. All misbehaviors except infinite loops can be treated by containing the execution within an exception catching environment. Infinite loops that contain

transactional reads or writes can be detected by amortizing extra validations inside the transactional operations. Other infinite loops can only be detected by compiling validation checks into code with loop that do not provably terminate, which is quite an intrusive solution.

In the granularity column, we classify the STMs by the type of transacted elements. In *object-based* ("per-word", PO) schemes, transactions consist of entire objects such as linked list nodes, while *word-based* ("per word", PW) ones manipulate individual word-size fields. A relaxed variation of word-based association is *stripe-based* (PS), where addresses with identical high-bits share the same lock, keeping the total number of locks constant. Part of a system's design would be the stripe's width, i.e. the size of a word (co)associated with a single lock. In general, operating in smaller granularity may reduce conflict rate, but usually increases the size of the transaction. In object-based implementations, the transactional data and/or meta-data is usually associated with the object by making it accessible through the object header. When granularity is words, this information must reside in a separate location, usually an associative map with the address as key. In some contexts, where it is not a viable option to insert fields into user data, locks must be associated by mapping.

STM schemes in which transactional data is stored inside object headers can be integrated with the traditional malloc-free dynamic memory allocation system only if readers are made visible. Otherwise, when an object is disposed while some active transaction has an old reference to it, the transactional data or meta-data might be accessed after being reclaimed. To prevent such scenarios, one must either make reads visible or utilize a garbage collector or alternatively a customized memory reclamation solution.

5.1.3 Transactional Locking

In this chapter, we introduce lock based STM implementations that are aimed at maximizing throughput while providing the desired practicality features, mainly the consistent state guarantees and compatibility to traditional memory management libraries. The essence of *Transactional Locking*, is executing transactions through acquiring the set of locks associated with the specific addresses in the transaction's write set. To detect conflicts, locks are *versioned*, i.e. include a monotonically increasing version number which is updated on

release. These high-level principles of transactional locking are similar to the ones used by Ennals [20].

The first scheme we propose is called **Transactional Locking I** (TL1). In TL1, threads execute transactions by first collecting the transaction's read- and write-sets, then acquire versioned locks[2] for all addresses in the write-set, validate that the read-set addresses were not modified since read, and release all locks with incremented versions. This approach is a combination of the one by Harris and Fraser's OSTM/WSTM [21, 34] and Ennals' [20]. TL1 resembles to OSTM/WSTM by acquiring the locks at commit-time, while it avoids the extra work of maintaining multiple copies of the data as it uses versioned locking, similarly to [20]. The TL1 scheme can be applied with either object or word granularity. In the *per-object* (PO) variation, each object is associated with a versioned lock residing in the object's header. In *per-stripe* (PS) TL1, individual words or object fields are associated with locks that are concentrated in a separate memory segment, acting as a map from memory "stripes" to a bounded set of locks.

By taking this combined approach, we benefit in two aspects: One is reduced overhead due to the use of locks instead of indirectly referenced data. The other is better compatibility with dynamic memory systems, gained through the commit-time locking property of TL1. In order to avoid referencing of reclaimed memory, threads must ensure the validity of their transaction on every acquire, and this can be done efficiently only if lock acquires are batched, as in commit-time locking schemes.

The second algorithm we propose is called **Transactional Locking II** (TL2). Unlike TL1, in TL2 the lock versions are not independent of each other, they all refer to a single global version, which is incremented on every transaction commit (in "vanilla" TL2, at least). The global version is sampled at the beginning of each transaction and validated against all read-set lock versions in the commit phase, during which write-set locks are acquired. Inconsistent state views are avoided by comparing each new transactional element's version to the global version at encounter-time. This low cost validation also enables very efficient reading-only transactions by not requiring the thread to post-validate its read-set. Reading-only transactions can simply complete successfully when the executing thread reaches the end of the transaction's code. Note that un-

[2]The usage of versioned locks is similar in spirit to Linux *SeqLocks*.

like in TL1, where lock versions were simply used to detect changed content, in TL2 we make use of the versions' magnitude.

The TL2 algorithm consists of the following six steps: (1) sample global version. (2) execute code while collecting read- and write-sets. (3) acquire write-set locks. (4) atomically increment global version. (5) validate read-set against the initial sampled global version. (6) store values, set new version numbers, and release locks. Reading-only transactions need only perform the first two steps, and therefore are almost overhead-free.

The drawbacks of TL2 compared to TL1 are its slightly larger and thus more conservative conflict window, and the centralized access to the global version, although usually limited to one write access per modifying transaction. TL2's pluses are its support for highly efficient reads and the completeness of its solution for inconsistent state prevention. In an effort to reduce the cost of global version access, we designed several variations to TL2 in which the global version is updated less frequently.

Since published [15], the TL2 algorithm has been chosen by several research groups in the area as a basis of comparison, acting as the state-of-the-art STM. In [11] and [56] it was used to evaluate hybrid hardware/software transactional memory implementations. In all but one benchmark, TL2 scaled very well, producing results mostly on the same scale with the proposed hybrid schemes. TL2 was also selected by Lourenco and Cunha [58] as the subject STM in their paper on testing patterns for STMs.

5.2 Transactional Locking I

We associate a special versioned-write-lock with every transacted memory location. A *versioned-write-lock* is a simple spinlock that uses a *compare-and-swap* (CAS) operation to acquire the lock and a store to release it. Since one only needs a single bit to indicate that the lock is taken, we use the rest of the lock word to hold a version number. This number is incremented by every successful lock-release.

We allocate a collection of *versioned-write-locks*. We use three alternative schemes for associating locks with shared data: *per object* (PO), where a lock is assigned per shared object, *per stripe* (PS), where we allocate a separate large array of locks and memory is stripped (divided up) using some hash function to map locations to different stripes, and *per word* (PW) where each transactionally referenced variable (word) is collocated adjacent to a lock. Other mappings between transactional shared variables and locks are possible. The PO scheme requires either manual or compiler-assisted automatic put of lock fields whereas PS can be used with unmodified data structures. PO might be implemented, for instance, by leveraging the header words of Java™ objects [3, 17]. A single PS stripe-lock array may be shared and used for different TL data structures within a single address-space. For instance an application with two distinct TL red-black trees and three TL hash-tables could use a single PS array for all TL locks. As our default mapping we chose an array of 2^{20} entries of 32-bit lock words with the mapping function masking (bitwise AND) the variable address with "0xFFFFC" and then adding in the base address of the lock array to derive the lock address. This mapping function incurs a stripe width of four bytes.

The following is a description of the PS algorithm although most of the details carry through verbatim for PO and PW as well. We maintain thread local read- and write-sets as linked lists. The read-set entries contain the address of the lock and the observed version number of the lock associated with the transactionally loaded variable. The write-set entries contain the address of the variable, the value to be written to the variable, and the address of the lock that "covers" the variable. The write-set is kept in chronological order to avoid write-after-write hazards.

The TL1 algorithm's steps are the following:

1. Run the transactional code, reading the locks of all fetched-from shared locations and building a local read-set and write-set.

   A transactional load first checks (using a Bloom filter [8]) to see if the load address appears in the write-set. if so the transactional load returns the last value written to the address. This provides the illusion of processor consistency and avoids so-called read-after-write hazards. We use the Bloom filter strictly as an optimization for accelerating the transactional loads, often avoiding the lookaside into the write buffer. If the address is not found in the write-set the load operation then fetches the lock value associated with the variable, saving the version in the read-set, and then fetches from the actual shared variable. If the transactional load operation finds the variable locked the load may either spin until the lock is released or abort the operation.

   Transactional stores to shared locations are handled by saving the address and value into the thread's local write-set. The shared variables are not modified during this step. That is, transactional stores are deferred and contingent upon successfully completing the transaction.

2. Attempt to commit the transaction. Acquire the locks of locations to be written. If a lock in the write-set (or more precisely a lock associated with a location in the write-set) also appears in the read-set then the acquire operation must atomically (a) acquire the lock *and*, (b) validate that the current lock version subfield agrees with the version found in the earliest read-entry associated with that same lock. An atomic CAS can accomplish both (a) and (b). Acquire the locks in any convenient order using bounded spinning to avoid indefinite deadlock. If bound is reached, release the locks, abort the transaction, and retry.

3. Re-read the locks of all read-only locations to make sure version numbers haven't changed. If a version does not match, roll-back (release) the locks, abort the transaction, and retry.

4. The prior observed reads in step (1) have been validated as forming an atomic snapshot of memory [1]. The transaction is now *committed*. Write-back all the entries from the local write-set to the appropriate shared variables.

5. Release all the locks identified in the write-set by atomically incrementing the version and clearing the write-lock bit (using a simple store).

A few things to note. The write-locks have been held for a brief time when attempting to commit the transaction. This helps improve performance under high contention. The Bloom filter allows us to determine if a value is not in the write set and need not be searched for by reading the single filter word. Though locks could have been acquired in ascending address order to avoid deadlock, we found that sorting the addresses in the write set was not worth the effort.

5.2.1 Contention Management

As described above TL admits live-lock failure. Consider where thread T1's read-set is A and its write-set is B. T2's read-set is B and write-set is A. T1 tries to commit and locks B. T2 tries to commit and acquires A. T1 validates A, in its read-set, and aborts as a B is locked by T2. T2 validates B in its read-set and aborts as B was locked by T1. We have mutual abort with no progress. To provide liveness we use bounded spin and a back-off delay at abort-time, similar in spirit to that found in CSMA-CD MAC protocols. The delay interval is a function of (a) a random number generated at abort-time, (b) the length of the prior (aborted) write-set, and (c) the number of prior aborts for this transactional attempt.

It is important to note that unlike both Ennals and Saha et al. [20, 82], we found that we do not need mechanisms for one transaction to abort another to allow progress/liveness even in encounter mode. Ennals allowed one transaction to passively ask another to abort itself. Saha et al. use a mechanism that allows a transaction to completely abort and undo the other's state. These mechanisms are unnecessary for performance or deadlock avoidance, and in a sense contradict the very philosophy behind transactional locking: rather than trying to improve on hand-crafted lock-based implementations by being non-blocking (hand-crafted lock-based data structures are not obstruction free), we try and build lock-based STMs that will get us as close to their behavior as one can with a completely mechanical approach, that is, one that truly simplifies the job of the concurrent programmer.

5.3 Transactional Locking II

Our TL2 algorithm is a two-phase locking scheme that employs *commit time* lock acquisition mode like the TL1 algorithm, differing from *encounter-time* algorithms such as those by Ennals [20] and Saha et al. [82].

For each implemented transactional system (i.e. per application or data structure) we have a shared *global version-clock* variable. We describe it below using an implementation in which the counter is incremented using an increment-and-fetch implemented with a compare-and-swap (CAS) operation. Alternative implementation exist however that offer improved performance. The global version-clock will be read and incremented by each writing transaction and will be read by every read-only transaction.

With every transacted memory location we associate a special versioned write-lock, which is incremented by every successful lock-release. A new element of TL2 is that unlike in the TL1 algorithm or algorithms like Ennals [20] and Saha et al. [82], the new value written in each location will be based on the shared *global version-clock* variable, a property which provides us with several performance and correctness benefits.

To implement a given data structure we allocate a collection of *versioned write-locks*. We can use the same alternative schemes for associating locks with shared data as in the TL1 algorithm: *per object* (PO), *per word*, or *per stripe*.

In the following we describe the PS version of the TL2 algorithm although most of the details carry through verbatim for PO and PW as well. We maintain thread local read- and write-sets as linked lists. The read-set entries contain the address of the lock (and unlike former algorithms does not need to contain the observed version number of the lock). The write-set entries contain the address of the variable, the value to be written to the variable, and the address of the lock that "covers" the variable (in many cases the lock and location address are related and so we need to keep only one of them in the read-set). The write-set is kept in chronological order to avoid write-after-write hazards.

5.3.1 The Basic TL2 Algorithm

We now describe how TL2 executes in commit mode a sequential code fragment that was placed within a TL2 transaction. As we explain, TL2 does not require traps or the insertion of validation tests within user code, and in the PS mode does not require type-stable garbage collection, working seamlessly with the memory life-cycle of languages like C and C++.

Write Transactions

The following sequence of operations is performed by a *writing transaction*, one that performs writes to the shared memory. We will assume that a transaction is a writing transaction. If it is a *read-only transaction* this can be annotated by the programmer, determined at compile time, or heuristically inferred at runtime.

1. **Sample global version-clock:** Load the current value of the *global version clock* and store it in a thread local variable called the *read-version* number (rv). This value is later used for detection of recent changes to data fields by comparing it to the *version* fields of their versioned write-locks.

2. **Run through a speculative execution:** Execute a transformation of the transaction code where load and store instructions are mechanically augmented and replaced so that speculative execution does not change the shared memory's state. Locally maintain a *read-set* of addresses loaded and a *write set* address/value pairs stored. This logging functionality is implemented by augmenting loads with instructions that record the read address and replacing stores with code recording the address and value to-be-written.

   The transactional load first checks (using a Bloom filter [8]) to see if the load address already appears in the write-set. If so, the transactional load returns the last value written to the address. This provides the illusion of processor consistency and avoids so-called read-after-write hazards.

   A load instruction sampling the associated lock is inserted before each original load, which is then followed by *post-validation* code checking that

the location's versioned write-lock is free and has not changed. Additionally, we make sure that the lock's version field is $\leq rv$ and the lock bit is clear. If it is greater than rv it suggests that the memory location has been modified after the current thread performed step 1, and the transaction is aborted.

3. **Lock the write set:** Acquire the locks in any convenient order using bounded spinning to avoid indefinite deadlock. In case not all of these locks are successfully acquired, the transaction fails.

4. **Increment global version-clock:** Upon successful completion of lock acquisition of all locks in the write-set perform an increment-and-fetch (using a CAS operation for example) of the global version-clock recording the returned value in a local *write-version number* variable wv.

5. **Validate the read-set:** Validate for each location in the read-set that its version number is $\leq rv$. We also verify that these memory locations have not been locked by other threads. In case the validation fails, the transaction is aborted. By re-validating the read-set, we guarantee that its memory locations have not been modified while steps 3 and 4 were being executed. In the special case where $rv + 1 = wv$ it is not necessary to validate the read-set, as it is guaranteed that no concurrently executing transaction could have modified it.

6. **Commit and release the locks:** For each location in the write-set, store to the location the new value from the write-set and release the locations lock by setting the version value to the write-version wv and clearing the write-lock bit (this is done using a simple store).

A few things to note. The write-locks have been held for a brief time when attempting to commit the transaction. This helps improving performance under high contention. The Bloom filter allows us to determine if a value is not in the write set and need not be searched for by reading the single filter word. Though locks could have been acquired in ascending address order to avoid deadlock, we found that sorting the addresses in the write set was not worth the effort.

Low-Cost Read-Only Transactions

One of the goals of the proposed methodology's design is an efficient execution of *read-only transactions*, as they dominate usage patterns in many applications. To execute a read-only transaction:

1. **Sample the global version-clock:** Load the current value of the *global version-clock* and store it in a local variable called *read-version* (rv).
2. **Run through a speculative execution:** Execute the transaction code. Each load instruction is *post-validated* by checking that the location's versioned write-lock is free and making sure that the lock's version field is $\leq rv$. If it is greater than rv the transaction is aborted, otherwise commits.

As can be seen, the read-only implementation is highly efficient because it does not construct or validate a read-set. Detection of read-only behavior of any given transaction can be done at the method level. This can be done at compile time or by simply running all methods first as read-only, and upon detecting the first transactional write, set a flag to indicate that this method should henceforth be executed in write mode.

5.3.2 Performance Enhancing Variations of TL2

Unlike most STM implementations, TL2 has centralized components, namely the global version. The key difficulty with the global version implementation is that it may introduce increased contention and costly cache coherent sharing. In the experimental evaluation of TL1 and TL2 this factor noticeably affected performance. We propose here variations of the "vanilla" TL2 algorithm aimed at providing less centralized memory access profiles.

Thread IDs Within the Version Word

One approach to reducing this overhead is based on splitting the global version-clock so it includes a version number and a thread id. Based on this split, a thread will not need to change the version number if it is different than the version number it used when it last wrote. In such a case all it will need to do is

write its own version number in any given memory location. This can lead to an overall reduction by a factor of n in the number of version clock increments.

- Each version will include the thread id of the thread that last modified it.
- Each thread, when performing the load/CAS to increment the global version-clock, checks after the load to see if the global version-clock differs from the thread's revious wv (note that if it fails on the CAS and retries the load/CAS then it knows the number was changed). If it differs, then the thread does not perform the CAS, and writes the version-clock it loaded and its id into all locations it modifies. If the global version number has not changed, the thread must CAS a new global version number greater by one and its id into the global version and use this in each location.
- To read, a thread loads the global version-clock, and any location with a version number $> rv$ or $= rv$ and having an id different than that of the transaction who last changed the global version will cause a transaction failure.

This has the potential to cut the number of CAS operations on the global version-clock by a linear factor. It does however introduce the possibility of "false positive" failures. In the simple global version-clock which is always incremented, a read of some location that saw, say, value $v + n$, would not fail on things less than $v + n$, but with the new scheme, it could be that threads 1..n-1 all perform non-modifying increments by changing only the id part of a version-clock, leaving the value unchanged at v, and the reader also reads v for the version-clock (instead of $v + n$ as he would have in the regular scheme). It can thus fail on account of each of the writes even though in the regular scheme it would have seen most of them with values $v...v + n - 1$.

Single CAS Attempt

In the original TL2 algorithm the transactional commit operation would (a) acquire locks covering the transaction's write-set, (b) atomically increment the global version number yielding a wv (write version) value, (c) validate the transaction's read-set set, and, contingent upon (c), write-back the values from

the write-set to their ultimate shared locations and then release and update the locks covering the write-set by storing wv into the lock-words. The increment of the global version number was accomplished with a loop using an atomic compare-and-swap (CAS) instruction.

We observed, however, that we can safely replace the loop with a single CAS attempt. Lets say we have two nearly simultaneous writers trying to atomically increment the global version number. The CAS performed by one thread succeeds but the CAS performed by the second fails, returning the value just installed by the 1st thread. We have determined that we can safely allow both threads to use the same wv. The thread whose CAS fails "borrows" the newly incremented value returned by the failing CAS instruction and uses that value as its wv. Note that we still incur CAS latency on every attempt to increment the global clock and we still generate cache-coherent read-write traffic on the clock but we have avoided CAS contention and the retries inherent in the original loop.

Allowing two writers to use the same wv is safe. If the CAS used to atomically increment the global version number fails then we have two writers racing: one atomic increment attempt succeeded and one failed. Because the two threads hold locks on their respective write-sets at the time they try to increment, it is guaranteed that their write-sets do not intersect. If the write-set of one thread intersects the read-set of the other then we know that one transaction will subsequently fail validation (either because the lock associated with the read-set entry is held by the other thread, or because the other thread already committed and released the lock covering the variable, installing the new wv). As such we can safely allow both threads to use the same (duplicate) wv. This relaxation provides a significant performance benefit on high-order SMP systems. Without loss of generality, we can extend this reasoning to more than two threads. The critical safety invariant is that any computed wv must be greater than any previously observed rv.

Increment on Abort

In this variation of TL2, the global version is not incremented on step 4 of the algorithm, it is only read at that point and the local variable wv is set to the value read plus one. When releasing the locks in step 6, the operating thread

sets their version to wv, as in the original scheme. **The global version is incremented when transactions abort**. Not modifying the global version on each transaction leads to threads failing to validate on step 5 of the algorithm, aborting their transactions. It is when a transaction aborts that the global version is atomically incremented.

In workloads where transaction write-sets are fairly distributed in memory, the price of the extra aborts is lower than the cost of cache invalidations generated by frequent updates to the global version. The overall throughput thus improves, as the solution is then less centralized.

TL2 for Static Transactions

Consider methods that are static, for example, an n location CAS operation (NCAS). For such methods in which all locations read and written are determined in advance, the TL2 algorithm offers an especially attractive implementation.

For such static transactions the read and write sets are known in advance, and as we show, revalidation can be eliminated. We do this using the following variation of the TL2 algorithm for *static transaction*.

1. **Lock Locations to be Written:** Acquire the locks for locations ot be written in any convenient order using bounded spinning to avoid indefinite deadlock. In case not all of these locks are successfully acquired, the transaction fails.
2. **Increment Global version-clock:** Upon successful completion of lock acquisition of all locks perform a increment-and-fetch of the global version-clock recording the returned value in a thread local *write-version number* variable wv.
3. **Validate Locations to be Read:** Validate for each location in the to be read that its version number is $\leq wv$. We also verify that these memory locations have not been locked by other threads. In case the validation fails, the transaction is aborted.
4. **Commit and release the locks:** For each location in the write-set, store to the location the value from the new write-set and release the location's

lock by setting the version value to the write-version wv and clearing the write-lock bit (this is done using a simple store).

This implementation is especially efficient since a transaction makes only a single pass over each memory location and its associated lock. This is an improvement over all know algorithms including that of Saha et al. [82] which constructs a read-set and revalidates it to perform an NCAS.

5.3.3 Optional Hardware Integration

A Hardware Clock Based Global version-clock

On new multi-core systems it may make sense to provide a *coherent hardware counter* or *coherent hardware clock* across all cores. If the system provides such a coherent hardware counter readable in user-space such that all values read are monotonically increasing (i.e. no to threads can ever read the same value) then this counter could be used in place of the software-maintained global counter. Hardware would continuously increment the counter, and software would only ever read the counter. Critically, the hardware-based counter would not generate cache-coherency traffic.

Software-Hardware Inter-Operability

Though we have described TL2 as a software based scheme, it can be made inter-operable with HTM systems on several levels.

On a machine supporting dynamic hardware transactions, transactions need only verify for each location that they read or write that the associated versioned write-lock is free. There is no need for the hardware transaction to store an intermediate locked state into the lock word(s). For every write they also need to update the version number of the associated stripe lock upon completion. This suffices to provide inter-operability between hardware and software transactions. Any software read will detect concurrent modifications of locations by a hardware writes because the version number of the associated lock will have changed. Any hardware transaction will fail if a concurrent software transaction is holding the lock to write. Software transactions attempting to

write will also fail in acquiring a lock on a location since lock acquisition is done using an atomic hardware synchronization operation (such as CAS or a single location transaction) which will fail if the version number of the location was modified by the hardware transaction.

One can also think of using a static bounded size obstruction-free hardware transaction to speed up software TL2. This may be done variously by attempting to complete the entire commit operation with a single hardware transaction, or, alternatively, by using hardware transactions to acquire the write locks "in bulk". This latter approach is beneficial if bulk acquisition of the write-locks via hardware transactions is faster (has lower latency) than acquiring one write lock at a time with CAS. Since the write set is known in advance, we require only static hardware transactions. Because for many data structures the number of writes is significantly smaller than the number of reads, it may well be that in most cases these hardware transactions can be bounded in size. If all write locks do not fit in a single hardware transaction, one can apply several of them in sequence using the same scheme we currently use to acquire individual locks.

One can also use TL2 as a hybrid backup mechanism to extend bounded size dynamic hardware transactions to arbitrary size. Again, our empirical testing suggests that there is not much of a gain in this approach.

5.4 STM Robustness Issues

5.4.1 Memory Management

Many common data structures are based on dynamically allocated objects interconnected by address referencing. Most structures provide interface for removing subset of the stored data, which usually affects the total amount of (heap) memory consumed. In the sequential implementation of these dynamic structures, removal of sub-components is followed by releasing the memory they occupy to the program heap. In managed environments, the garbage collector is reponsible for disposing unused memory, while programming in unmanaged or native environments requires explicit calls to *free*.

Using transactions rather than coarse-grained locking for concurrent access imposes challenges on the timing of memory disposal. Since the heap itself does not reside on transactable memory locations, every change made to it is immediately visible to all concurrently executing threads. To prevent transactions from modifying potentially disposed and reallocated memory, one can either (1) implement a mechanism that delays memory reclamation until it is safe to do so, or (2) have threads verify that memory locations are valid when modifying them.

The first option is equivalent to implementing garbage collection for transactional objects, which means that the program's data would reside in two separate memory pools. This makes transferring objects to and from the transactional domain a non trivial task. Taking the second approach when programming under non-blocking constraints would be inefficient, since objects may be reclaimed in the short time window between validation and modification.

How, then, memory locations can be trusted to be valid? Correct programs never free objects while they are accessible, and the executing thread has reached the object in question by following a sequence of references that are included in the transaction's read-set. Therefore, validating the transaction guarantees that the object is accessible and thus has not been reclaimed. In both TL1 and TL2, transacted memory locations are always modified after the transaction is validated and before their associated lock is released. This leaves us with a short period in which the objects in the transaction's write-set must not

be freed. To prevent objects from being freed in that period, we let objects *quiesce* before freeing them. By *quiescing* we mean losing activity in their transactional fields as field locks are being released by their owners.

In STMs that use encounter-time lock acquisition and undo-logs [20, 82] it is significantly harder to efficiently protect objects from being modified after they are reclaimed, as memory locations are modified one at a time, replacing old values with the new values written by the transaction. Even with quiescing, to protect from illegal memory modifications, one would have to repeatedly validate the entire transaction before updating each location in the write-set. This *repeated* validation is inefficient in its simplest form and complex (if at all possible) if one attempts to use the compiler to eliminate unnecessary validations.

Unfortunately, we have not found an efficient scheme for using the *per-object* (PO) mode of TL1 and TL2 in C or C++ because locks reside inside the object header, and the act of acquiring a lock cannot be guaranteed to take place while the object is alive. As can be seen in the performance section, on the benchmarks/machine we tested there is a penalty, though not an unbearable one, for using PS mode instead of PO.

5.4.2 Inconsistent States

Tranasctional execution is usually associated with a single major requirement: that all successful transactions appear as if they were performed atomically. However, transactional memory implementations must also guarantee that **unsuccessful** transactions will not affect the program's state. Since STMs are designed in such way that unsuccessful transactions do not modify data (by either delaying stores or doing rollback on abort), the only way they can still affect the program's state is by misbehaving while processing data. Such scenarios would have been impossible if changes to memory by successful transactions were absolutely atomic. However, as atomicity is only emulated, partial updates can be viewed by threads that would eventually abort.

During the period between the time a thread viewed some inconsistency and the time its active transaction was aborted, the execution may encounter states that could not have been anticipated by the programmer. These unexpected

states can lead to infinite loops, illegal memory accesses, division by zero, and other exceptions. We discussed the variety of approaches for dealing with this problem in Section 5.1.2.

The TL2 algorithm avoids inconsistent states in a rather elegant way: by sampling the versioned lock of a memory location on every (read or write) access to it, the algorithm is guaranteed to operate on a snapshot of its read-set addresses. This snapshot corresponds to the time when the thread executed step 1 of the algorithm, in which the global version was saved in the thread variable rv.

5.5 Empirical Performance Evaluation

We present here a comparison of algorithms representing state-of-the-art blocking and non-blocking paradigms on two popular data structures: red-black trees and skip-lists [75]. We chose these data structures as they involve non-trivial transactions in the common case, and also have available hand-crafted algorithms which we can compare to. The red-black tree data structure has become a standard in benchmarking STMs [44].

The red-black tree and skip-list implementations expose a key-value pair interface of *put*, *delete*, and *get* operations. The put operation installs a key-value pair. If the key is not present in the data structure put will insert a new element describing the key-value pair. If the key is already present in the data structure put will simply update the value associated with the existing key. The get operation queries the value for a given key, returning an indication if the key was present in the data structure. Finally, delete removes a key from the data structure, returning an indication if the key was found to be present in the data structure. The benchmark harness calls put, get and delete to operate on the underlying data structure. The harness allows for the proportion of put, get and delete operations to be varied by way of command line arguments, as well as the number of threads, trial duration, initial number of key-value pairs to be installed in the data structure, and the key-range. The key range describes the maximum possible size (capacity) of the data structure.

The harness spawns the specified number of threads. Each of the threads loops, and in each iteration the thread first computes a random number used to select, in proportion to command line argument mentioned above, if the operation to be performed will be a put, get or delete. The thread then generates a random key within the key range, and, if the operation is a put, a random value. The thread then calls put, get or delete accordingly. All threads operate on a single shared data structure. At the end of the timing interval specified on the command line the harness reports the aggregate number of operations (iterations) completed by the set of threads.

For our experiments we used a 16-processor Sun Fire™ V890 which is a cache coherent multiprocessor with 1.35Ghz UltraSPARC-IV® processors running Solaris™ 10.

5.5.1 Red-Black Tree Experiments

The red-black tree implementation we used in our experiments was derived from the `java.util.TreeMap` implementation found in the Java 6.0 JDK. That implementation was written by Doug Lea and Josh Bloch. In turn, parts of the Java TreeMap were derived from Cormen et al [13]. We would have preferred to use the exact Fraser-Harris red-black tree [34] but that code was written to their specific transactional interface (with integrated garbage collection) and could not readily be converted to a simple form.

In the first set of graphs, we include the results of the following algorithms:

Mutex The Solaris POSIX threads library mutex as a coarse-grained locking mechanism. It uses a spin-then-block strategy. In addition, the Solaris pthreads mutex uses schedctl to avoid spinning if the owner is descheduled.

MCS A simple spin-based MCS-lock [65].

stm_fraser This is the state-of-the-art non-blocking STM of Harris and Fraser [34]. We use the name originally given to the program by its authors. It has a special record per object with a pointer to a transaction record.

stm_ennals This is the lock-based encounter order object-based STM algorithm of Ennals taken from [20] and provided in LibLTX [34]. Note that LibLTX includes the original Fraser and Harris lockfree-lib package. It uses a lock per object and an object-based interface of [34]. Though we did not have access to code for the Saha et al algorithm [82], we believe the Ennals algorithm to be a good representative this class of algorithms, with the possible benefit that the Ennals structures were written using the non-mechanical object-based interface of [34] and because unlike Saha et al, Ennals's write set is not shared among threads.

hanke This is the hand-crafted lock-based concurrent relaxed red-black tree implementation of Hanke [28] as coded by Fraser [34]. The idea of relaxed balancing is to uncouple the re-balancing from the updating in order to speed up the update operations and to allow a high degree of concurrency.

TL1/PO The TL1 algorithm described in Section 5.2, which does not use a global version clock; It collects read and write sets and validates the read-set after acquiring the locks on the memory locations. The "/PO" stands for the *per-object* variation of the algorithm.

TL2/PO The per-object variation of the new globally versioned transactional locking algorithm of Section 5.3.

TL2-IOA/PO The *Increment-On-Abort* variation of TL2, described in Section 5.3.2.

In Figure 5.1 we present six red-black tree benchmarks performed using two different key ranges and three set operation distributions. The key range of $[100, 200]$ generates a small size tree while the range $[10000, 20000]$ creates a larger tree, imposing larger transaction size for the set operations. The different operation distributions represent two types of workloads, the first with nothing but read operations, the second is dominated by reads (5% puts, 5% deletes, and 90% gets), and in the third, most operations are writes (30% puts, 30% deletes, and 40% gets).

In all six graphs, all algorithms scale quite well to 16 processors, with the exception of the two mutual exclusion based ones. Ennals' algorithm performs badly on the contended write-dominated benchmark, apparently suffering from frequent transaction collisions, which are more likely to occur on encounter-time locking based solutions.

The performance of the STM implementations usually differs by a constant factor, most of which we associate with the overheads of the algorithmic mechanisms employed (as seen in the single thread performance). The hand-crafted algorithm of Hanke provides the highest throughput in all cases because its single thread performance is superior to all STM algorithms. Among the STMs, TL1 and TL2-IOA had the best performance, their order depending on the percentage of non-modifying transactions. The TL1 algorithm, being more accurate in aborting transactions, was faster than the TL2 variations on benchmarks with significant percentage of writes. On the other hand, the TL2s performed better on read-dominated workloads, as they do not require the re-validation step for reading-only transactions.

In Figure 5.2 we demonstrate the effect of lock granularity on performance by

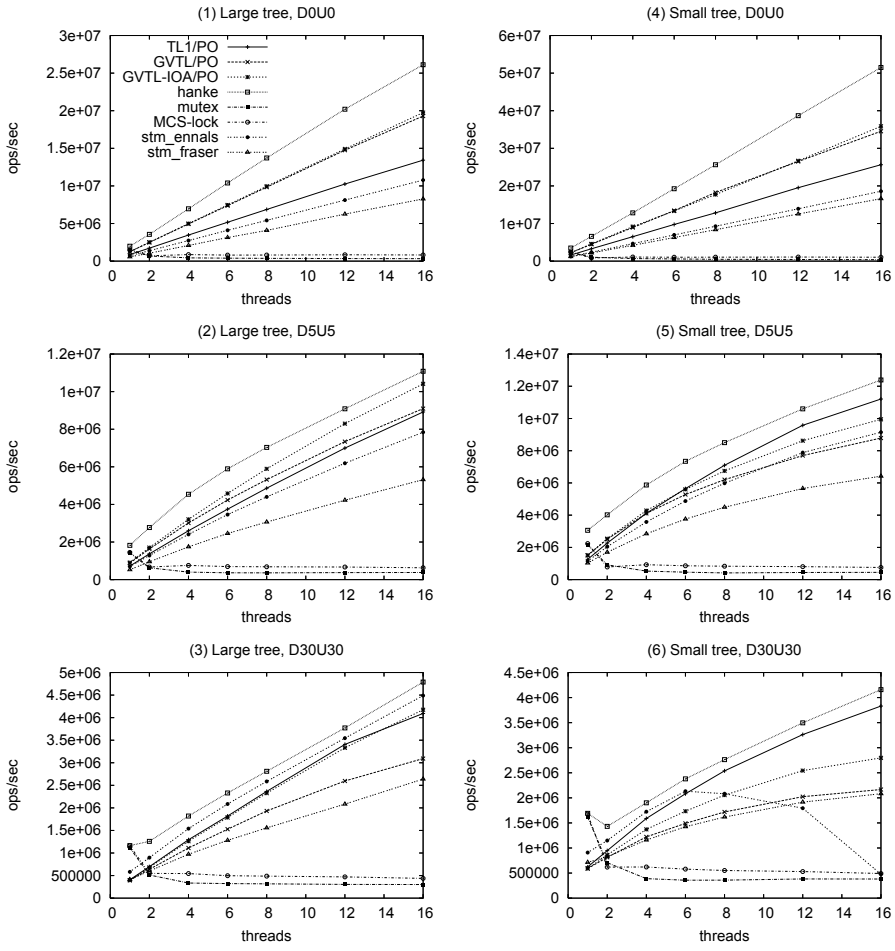


Figure 5.1: Throughput of large (left) and small (right) red-black tree with 100% gets (top), with 5% puts and 5% deletes (middle), and 30% puts and 30% deletes (bottom)

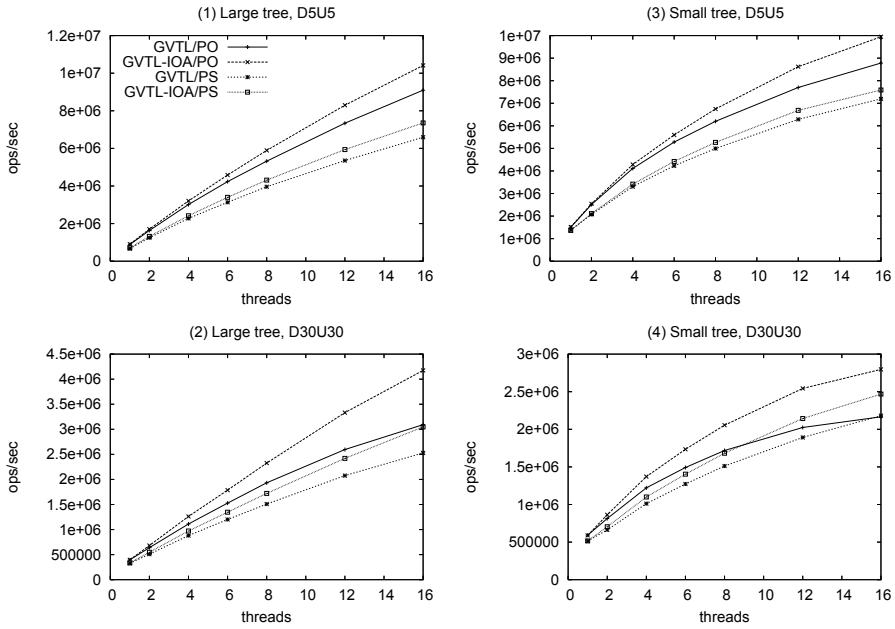


Figure 5.2: Throughput of the Per-Object and Per-Stripe TL2 variations on large (left) and small (right) red-black tree with 5% puts and 5% deletes (middle), 30% puts and 30% deletes (bottom)

comparing the per-object and per-stripe locking schemes in TL2. As expected, the per-object variations perform better due to the smaller read- and write-sets used. In 5.2(4), we observe how the per-object TL2 algorithm slightly suffers from contention in the frequently modified small red-black tree.

5.5.2 Skiplist Experiments

We compared the set of STMs with a the cas-based concurrent skiplist algorithm of Sundell and Tsigas [90] (marked "ST" in the graph legend), as well as the two coarse-grained locks. We based our code on Rickard Faith's skiplist implementation used in a number of open-source projects, which is faithful to Pugh's original design [75].

Unlike the case of red-black trees, the hand-crafted algorithm was significantly faster than any of the STMs, whose performance still differentiated by fixed fac-

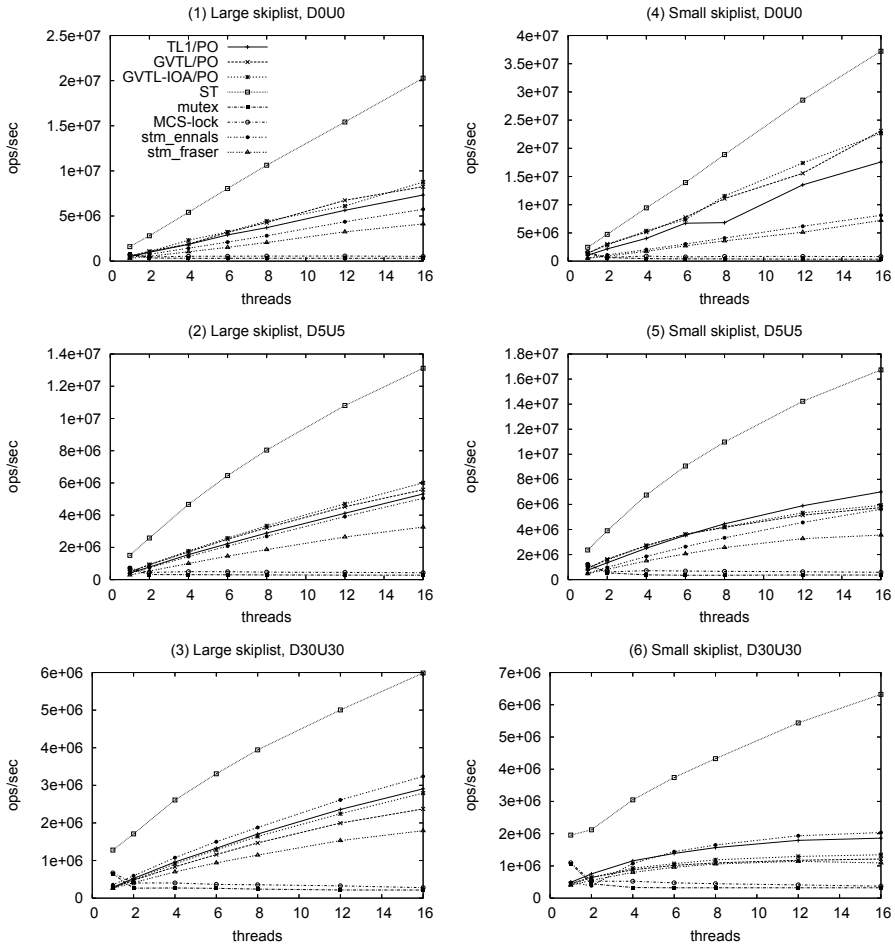


Figure 5.3: Throughput of large (left) and small (right) skiplist with 100% gets (top), with 5% puts and 5% deletes (middle), and 30% puts and 30% deletes (bottom)

tors. The results of our experiments on the skiplist data structure are presented in Figure 5.3. In most graphs, the TL1 and TL2 variations perform better than the two other STMs, while in write-dominated workloads Ennals' algorithm is slightly faster, due to its early acquire approach, preventing collisions in such scenarios.

5.6 Correctness of the Basic TL2 Algorithm

We base the correctness of the TL2 algorithm on simple claims on the linearization of transactions. In the proof, we refer to the algorithm steps as they were defined in Section 5.3. Data-modifying transactions are linearized at the exact point in time they execute the atomic memory access in step 4 of the algorithm, while reading-only transactions are linearized at step 1.

In this section, we show that successful transactions affect each other with respect to the order of their linearization points. We treat concurrently executing modifying transactions separately from concurrent read-only/modifying ones. We prove the algorithm's correctness by showing that any transaction is not affected by ones linearized later in time, and **fully** aware of earlier transactions.

5.6.1 Write-Write Conflicts

Consider op_1 and op_2, two successful modifying transactions executed on a shared data structure by separate threads t_1 and t_2. Thread t_1 obtains rv_1 and wv_1 as its reading and writing versions of the global version clock, while t_2 receives rv_2 and wv_2. We assume *w.l.o.g.* that $wv_2 > wv_1$, i.e. t_1 was the first to execute step 4. We denote the read- and write-set as constructed in step 2 of the algorithm by $rs(op)$ and $ws(op)$, and use the notation $v_i^{(s)}(x)$ as the version value associated with address x as it was read by op_i in step s of the algorithm.

In Lemma 5.6.1, we claim that the more recent transaction, op_2, sees the other's data modifications as if they happened atomically, and in Lemma 5.6.2 we show that op_1 is not aware of the modifications done by op_2.

Lemma 5.6.1. *For all addresses $x \in ws(op_1) \cap rs(op_2)$, if there was no successful transaction op' with $wv_1 < wv' < wv2$ and $x \in ws(op')$, then op_2 reads from x the same value op_2 has written in it.*

Proof. Assume by way of contradiction that op_2 reads a different value. One of the following must be true:

1. op_2 read x before op_1 wrote to it.
   Since op_1 needs x locked during steps 4 and 5 in the algorithm and since

it was the first to execute step 4, we know that op_2 was in step 5 later than the time x was locked by op_1 while executing step 3. From the success of op_2's validation in step 5, x must have been released by op_1, and $v_2^{(5)}(x) \geq wv_1$.

In step 5, op_2 verified that $v_2^{(5)}(x) \leq rv_2$. Therefore, we get that $rv_2 \geq wv_1$, which means that op_2 executed step 1 after op_1 was in step 4. When op_2 was in step 2 doing the speculative execution, it verified that while reading x, there was no active lock on it by checking its version before and after the load. As x was not locked when read by op_2, it must have been released by op_1, and its step 2 version $v_2^{(2)}(x) \geq wv_1$. This contradicts with the assumption that op_2 read from x before op_1 wrote to it.

2. There exists op' that wrote into x later than op_1 and earlier than the time op_2 started step 3 in the algorithm.
   It follows that op' executed step 4 earlier than op_2. Since both op_1 and op' needed to lock x and op_1 did it first, we have that op' executed step 4 later than op_1. Therefore, $wv_1 < wv' < wv_2$, contradicting the Lemma's predicate.

The above contradictions conclude the proof of the lemma. □

Lemma 5.6.2. *For simplicity, we will assume that stored values are unique. If $x \in rs(op_1) \cap ws(op_2)$, then op_1 reads from x a different value than the one op_2 writes there.*

Proof. If op_1 reads the value written to x by op_2, it does it while in step 2. op_2 cannot possibly execute step 6 earlier than op_1 does step 3, because $wv_1 < wv_2$, which means that op_2 gets to step 4 later than op_1. □

5.6.2 Read-Write Conflicts

We now define op_2 as a successful reading only transaction. We define op_1 to take effect before op_2 by executing step 4 before op_2 reads rv_2 in step 1. In Lemma 5.6.3 we claim that op_2 sees the modifications of op_1 as if they took place atomically.

Lemma 5.6.3. *If $x \in ws(op_1) \cap rs(op_2)$ and there does not exist a successful transaction op' with $wv' > wv_1$ that executes step 4 before op_2 executes step 1, then op_2 reads from x the same value op_1 has written to it.*

Proof. Assume by way of contradiction that op_2 returns a different value. One of the following must be true:

1. op_2 read x before op_1 wrote to it.
   Impossible since op_2 started when op_1 was in step 4, holding the lock on x at least until the time it wrote to x in step 6. op_2 checks the lock and version before and after reading from x. If it read from x before op_1 wrote there, the first check must have found an occupied lock and the transaction would have failed.

2. There exists a successful transaction op' that wrote a value val' to x before op_2 read from it.
   From the lemma, op' either executes step 4 after op_2 does step 1, or $wv' < wv_1$. In the first case, $wv' > rv_2$, so when op_2 reads the value written to x by op', it does it after the lock has been released (because of the lock validations before and after the load,) and fails as $v_2^{(2)}(x) \geq wv' > rv_2$.

   In the second case, op' executes step 4 earlier than op_1, and since both must lock x, op_1 executes step 6 later. op_2 starts after op_1 did step 4, but since it succeeds, by the time it post-validates x, op_1 has released the lock overriding val'.

□

Chapter 6

Summary

This thesis is an embodiment of the exploration process I have been through in the past few years. I started off from the observation that the field of shared-memory synchronization suffers from major practical issues that limit its popularity among practitioners. My goal during the entire process was to make concurrency easier without sacrificing performance.

The inherent problem in enabling concurrent access to shared data is the trade-off between the code complexity and parallelism. Simpler code is easy to design and maintain, but it usually means that safety is kept using less scalable methods, such as coarse-grained locking. On the other side, hand crafted concurrent algorithms are sometimes extremely difficult to design and prove correct, but the provide better performance due to the use of finer-grained synchronization. Researchers have devised generic synchronization frameworks, such as transactional memory schemes, aimed at simplifying the programming interface for fine-grained synchronization. However, they still impose non-trivial requirements on programmers, that still need skills in concurrent reasoning in order to write efficient and sometimes correct code.

I started my journey in the field of shared memory synchronization by designing an efficient and scalable lock-free hash table algorithm, based on the novel concept of *split-ordering*. Working on the algorithm and its correctness proof made me understand the need for and the challenges in designing efficient generic synchronization frameworks. I followed my intuition that non-

compromising approach of the existing non-blocking STM schemes of that time could not evolve into both efficient and programmer-friendly solutions. I figured that some sequential factor must be introduced in order to completely free the programmer from concurrency reasoning. That sequential factor started as the high-level operation log of *Predictive Log-Synchronization*, which still required some special but sequential reasoning.

To remove that requirement, it was necessary to reduce the granularity of logged elements to words. Therefore, in *Address-Based Log-Synchronization* I used a log of low-level memory accesses, which enabled fine-grained transactions with limited parallelism. Since all modifying operations were serialized, the overall throughput was bounded by the rate of modifying transactions. In the *Transactional Locking II* algorithm, the log was reduced to a global counter. The TL2 global clock was in fact the reason for some of its extremely desired properties as an STM: it sustains inconsistent states and fits mixed transactional and non-transactional memory usage. In the TL2-IOA (*Increment On Abort*) variation the global clock was no longer a burden on scalability, and the serialization problem that was present since the Predictive Log-Synchronization design has finally come to a closure.

Paradoxically, the original observation that serialization can counter-intuitively be beneficial for synchronization in highly parallel programs, eventually led me to a design which is both efficient and scalable. The way that began with the centralized design of log-synchronization ended with the efficient STM-like TL2-IOA.

Bibliography

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[2] O. Agesen, D. Detlefs, C. Flood, A. Garthwaite, P. Martin, M. Moir, N. Shavit, and G. Steele. DCAS-based concurrent deques. *Theory of Computing Systems*, 35(3):349–386, 2002.

[3] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. *ACM SIGPLAN Notices*, 34(10):207–222, 1999.

[4] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[5] C. Scott Ananian and Martin Rinard. Efficient software transactions for object-oriented languages. In *Proceedings of Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. ACM, 2005.

[6] Anderson and Moir. Universal constructions for large objects. In *WDAG: International Workshop on Distributed Algorithms*. LNCS, Springer-Verlag, 1995.

[7] Hagit Attiya and Eyal Dagan. Improved implementations of binary universal operations. *J. ACM*, 48(5):1013–1037, 2001.

[8] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[9] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Apr 2006.

[10] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*. Series: Series in Computer Science. Springer, 2005.

[11] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.

[12] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition edition, 2001.

[13] Thomas H. Cormen, E. Leiserson, Charles, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990. COR th 01:1 1.Ex.

[14] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM Press.

[15] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, September 2006.

[16] Dave Dice and Nir Shavit. What really makes transactions fast? In *TRANSACT06 ACM Workshop*, June 2006.

[17] David Dice. Implementing fast java monitors with relaxed-locks. In *Java Virtual Machine Research and Technology Symposium*, pages 79–90. USENIX, 2001.

[18] C. Ellis. Concurrency in linear hashing. *ACM Transactions on Database Systems (TODS)*, 12(2):195–217, 1987.

[19] Carla Schlatter Ellis. Concurrency in linear hashing. *ACM Trans. Database Syst.*, 12(2):195–217, 1987.

[20] Robert Ennals. Software transactional memory should not be obstruction-free. www.cambridge.intel-research.net/ rennals/notlockfree.pdf, 2005.

[21] K. Fraser. *Practical Lock-Freedom*. Ph.D. dissertation, Kings College, University of Cambridge, Cambridge, England, September 2003.

[22] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[23] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. Almost wait-free resizable hashtable. In *Proceedings of the 18th International Parallel and Distributed Processing Symopsium (IPDPS)*, 2004.

[24] M. Greenwald. *Non-Blocking Synchronization and System Design*. Phd thesis, Stanford University, Palo Alto, CA, 1999.

[25] M. Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data-structures using dcas. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 260–269, jul 2002.

[26] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 258–264, New York, NY, USA, 2005. ACM Press.

[27] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[28] S. Hanke. The performance of concurrent red-black tree algorithms. *Lecture Notes in Computer Science*, 1668:286–300, 1999.

[29] T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on DIStributed Computing*, pages 265–279, 2002.

[30] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science*, 2180:300–314, 2001.

[31] Tim Harris. Exceptions and side-effects in atomic blocks. In *Proceedings of the 2004 Workshop on Concurrency andSynchronization in Java programs*, pages 46–53, Jul 2004. Proceedings published as Memorial University of Newfoundland CS Technical Report 2004-01.

[32] Tim Harris and Keir Fraser. Harris-fraser word-based stm. `http://www.cl.cam.ac.uk/Research/SRG/netos/lock-free/src/libstm-v1.tar.gz`.

[33] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, 2003.

[34] Tim Harris and Keir Fraser. Concurrent programming without locks. www.cl.cam.ac.uk/Research/SRG/netos/papers/2004-cpwl-submission.pdf, 2004.

[35] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[36] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[37] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. In *Proceedings of the 16th International Symposium on DIStributed Computing*, volume 2508, pages 339–353. Springer-Verlag Heidelberg, January 2002. A improved version of this paper is in preparation for journal submission; please contact authors.

[38] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529. IEEE, 2003.

[39] M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[40] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[41] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[42] Maurice Herlihy. The SXM software package. `http://www.cs.brown.edu/~mph/SXM/README.doc`, 2005.

[43] Maurice Herlihy, Victor Luchangco, and Mark Moir. Dynamic software transactional memory library 2.0. `http://www.sun.com/download/products.xml?id=453fb28e`.

[44] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[45] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM Press, 2003.

[46] 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. ACM Press, 1993.

[47] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[48] W.H. Hesselink, J.F. Groote, S. Mauw, and R. Vermeulen. An algorithm for the asynchronous write-all problem based on process collision. *Distributed Computing*, 14(2):75–81, 2001.

[49] M. Hsu and W. Yang. Concurrent operations in extendible hashing. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 241–247. Morgan Kaufmann, 1986.

[50] P. C. Kanellakis and A. Shvartsman. *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers, 1997.

[51] Sanjeev Kumar, Michael Chu, Christopher Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *To appear in PPoPP 2006*, 2006.

[52] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1997.

[53] D. Lea. Personal communication, January 2003.

[54] D. Lea. Concurrent hash map in JSR166 concurrency utilities, 2004.

[55] Doug Lea. Email communication detailing frequency of hashtable downsizing in standard benchmarks., January 2005.

[56] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2007.

[57] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings*, pages 212–223. IEEE Computer Society, 1980.

[58] Joao M.S. Lourenco and Goncalo T. Cunha. Testing patterns for software transactional memory engines. In *PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 36–42, New York, NY, USA, 2007. ACM.

[59] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare single swap. In *Proc. 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 2003.

[60] Nancy Lynch and Nir Shavit. Timing-based mutual exclusion. In *Proceedings of the IEEE Real-Time Systems Symposium.*, pages 2–11, 1992.

[61] Virendra J. Marathe, William N. Scherer, and Michael L. Scott. Design tradeoffs in modern software transactional memory systems. In *In Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR'04)*, 2004.

[62] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sep 2005.

[63] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, June 2006.

[64] Paul E. Mckenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[65] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[66] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the third ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 106–113. ACM Press, 1991.

[67] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM Press, 2002.

[68] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 21–30. ACM Press, 2002.

[69] 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.

[70] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, August 1997.

[71] Kevin E. Moore, Bobba Jayaram, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th Annual Internaitonal Symposium on High Performance Computer Architecture (HPCA 2006)*, 2006. to appear.

[72] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and preliminary architecture sketches. In *Proceedings of Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. ACM, 2005.

[73] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM Press.

[74] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA)*, pages 182–204, July 1999.

[75] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *ACM Transactions on Database Systems*, 33(6):668–676, 1990.

[76] Chris Purcell and Tim Harris. Non-blocking hashtables with open addressing. In *DISC*, pages 108–121, 2005.

[77] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[78] Ravi Rajwar and Mark Hill. Transactional memory online. `http://www.cs.wisc.edu/trans-memory`, 2006.

[79] Yaron Riany, Nir Shavit, and Dan Touitou. Towards a practical snapshot algorithm. *Theoretical Computer Science*, 269(1-2):163–201, 2001.

[80] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, September 2006.

[81] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[82] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM Press.

[83] William N. Scherer and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM Press.

[84] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. In *The 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 102–111. ACM Press, 2003.

[85] Ori Shalev and Nir Shavit. Predictive log-synchronization. In *EuroSys 2006*, pages 305–315, Apr 2006.

[86] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, 2006.

[87] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 6(30):645–670, 1997.

[88] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.

[89] Robert E. Strom and Joshua S. Auerbach. The optimistic readers transformation. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 275–301, London, UK, 2001. Springer-Verlag.

[90] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65(5):609–627, 2005.

[91] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

[92] Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Preemption-based avoidance of priority inversion for Java. In *Proceedings of the International Conference on Parallel Processing*, pages 529–538. IEEE Computer Society, 2004.

[93] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 519–542. Springer-Verlag, 2004.

אוניברסיטת תל אביב

הפקולטה למדעים מדויקים על שם ריימונד ובברלי סאקלר

בית הספר למדעי המחשב

טכניקות לבניית מבני נתונים הניתנים לגישה מקבילית

מאת

אורי שלו

חבור לשם קבלת תואר "דוקטור לפילוסופיה"

הוגש לסנט של אוניברסיטת תל אביב

אוקטובר 2007

בהנחיית:

פרופ' ניר שביט

פרופ' סיון טולדו

תמצית העבודה

במסמך זה אציג מספר טכניקות חדשניות לגישה מקבילית של מבני נתונים על זיכרון משותף במחשבים מרובי-מעבדים. טכניקות אלה משפרות את מקביליות הריצה ולכן את הספק החישוב הכולל על ידי הימנעות משימוש בנעילות כוללניות. תחילה, אציג עיצוב חדש של מבנה נתונים מסוג "טבלת hash", המבוסס על עקרון חדש הנקרא "סידור מפוצל". האלגוריתם המוצע הוא הראשון אשר אינו עושה שימוש בנעילות וגם מאפשר הרחבה של הטבלה ביעילות. בהמשך, אציג שלוש פרדיגמות כלליות למימוש מקביליות במבני נתונים על זיכרון משותף.

שתי הפרדיגמות הראשונות מבוססות על עקרון חדש בשם "סינכרוניזציית-לוג". הן מיועדות לשיפור הספק החישוב באמצעות מתן היכולת לבצע פעולות קריאה בלבד במקביל לפעולות כתיבה. ביצוע נמוך-תקורה של פעולות קריאה בלבד מתאפשר בזכות שכפול של נתונים לשני עותקים.

לבסוף, אציג את פרדיגמת ה"נעילה הטרנזקציונית", המהווה גישה פרקטית לעיצוב יעיל של מערכות זיכרון טרנזקציוני בתוכנה. אציג את TL2, סכימה המשתמשת בנעילות בגרעיניות נמוכה ובנויה על עקרון מפתח של שעון גלובלי. TL2 מתמודדת בהצלחה עם היבטים פרקטיים של מימוש זיכרון טרנזקציוני בתוכנה: תקורה נמוכה, אי-פגיעות למצבים לא קונסיסטנטיים ותאימות לשיטות ניהול זיכרון סטנדרטיות.

בסיס משותף לכל הפתרונות המוצעים הוא הדגש על המימד הפרקטי שלהם. כל האלגוריתמים נוסו אמפירית ונמצאו כבעלי הספק עבודה גבוה בהשוואה לשיטות אלטרנטיביות, במגוון רחב של ניסויים.

תקציר העבודה

בשנים האחרונות, מחשבים מרובי מעבדים עם זיכרון משותף זוכים לפופולאריות בעקבות בצורך הגדל בכלים לחישובים אינטנסיביים בתעשיית התוכנה. למרות הטבע המבוזר של רשת האינטרנט, אפליקציות רבות תלויות בישויות ריכוזיות, כגון בסיסי נתונים, שעבורם הפלטפורמה הנפוצה ביותר היא מחשב מרובה מעבדים. ניצול ריבוי מעבדים לצורך הגברת הספק החישוב אינו דבר טריוויאלי. יש להתגבר על מספר צווארי בקבוק בכדי להגיע לפתרון הניתן להרחבה, ביניהם: משאבי החומרה, כגון זיכרון ויחידות חישוב, יחידות תקשורת, ואפליקציות תוכנה.

בעוד האפקטים של צווארי-בקבוק הקשורים בחומרה ניתנים למזעור באמצעות שדרוג פיסי של החומרה, צווארי-הבקבוק הקשורים בתקשורת ובאפליקציה הם אינהרנטיים. מספר המעבדים הגדול גורר זמן שירות ארוך לשאילתות זיכרון, וביצוע סדרתי של תכניות לא מנצל באופן יעיל את משאבי החישוב הזמינים. גישה נפוצה במחקרים שנושאים הגברת הספקי חישוב במחשבים מרובי מעבדים עם זיכרון משותף היא עיצוב ידני ייחודי של אלגוריתמים עבור גישה מקבילית לנתונים משותפים. אלגוריתמים אלה לרוב מאפשרים מקביליות גבוהה תוך שימוש יעיל יותר במשאבי התקשורת.

כאשר מתכנתים בסביבה מרובת מעבדים, יש להיות ערים לערכים המשתנים ללא הרף של הנתונים המשותפים. המרה של אלגוריתם סדרתי למקבילי עשויה להיות משימה קשה ביותר, בשל מגוון גורמים התלויים בתזמון מערכת ההפעלה ומנגנון קוהרנטיות זיכרון המטמון, גורמים שבדרך כלל שקופים למתכנת אך קריטיים לביצוע נכון ממהיר של התכנית. לעיתים קרובות ישנו יחס הפוך בין סיבוכיות התכנית לבין רמת המקביליות בה היא תומכת: נעילות גסות הן פשוטות ובטוחות לשימוש אך גורמות לעיתים לצירים קריטיים ארוכים שפוגעים במקביליות. הוכחה של אלגוריתמים מקביליים ייחודיים עשויה לגזול זמן רב בשל הטבע הלא דטרמיניסטי של ביצוע מקבילי.

נעילות גסות, על פי הגדרתן, הן במקרים רבים בחירה גרוע כאבן בניין לסינכרוניזציה באפליקציות מקביליות. הסיבה היא שמנעולים מספקים הפרדה בזמן, אך לא במרחב הזיכרון. שימוש בקבוצה קטנה של מנעולים עבור מבני נתונים גדולים ומורכבים מגביל את זמינותם של חלקים גדולים מהנתונים המשותפים. לרוב, אין זה יעיל שחוטי חישוב הפועלים על חלקים שונים של מבנה הנתונים יחסמו עצמם הדדית בשל העובדה שהם מבצעים פעולה דומה בו זמנית. נעילות בגרעיניות גסה כמעט ולא משקללות את הלוקאליות של הפעולות, רק את התזמון שלהן. על ידי הורדת הגרנולריות של הנעילות (והגדל מספר המנעולים), עבור מבני נתונים מסוימים, אפשר להוריד את גודל אפקט ההתחרות בין חוטי חישוב. עם זאת, נעילות בגרנולריות נמוכה עשויות להיות מסובכות מאוד למימוש.

בעוד מספר מבני נתונים, כגון מחסניות ותורים, הם ריכוזיים באופן בסיסי בכך שיש להם מספר קטן של נקודות עומס, מבנים אחרים כגון עצי חיפוש וטבלאות האש מתאפיינים בגישה מבוזרת יותר. שימוש בנעילות בגרעיניות נמוכה פועל באופן חכם יותר על אותם מבני מתונים ומשפר משמעותית את הספק החישוב הכולל.

פרדיגמת התכנות ללא-נעילות צמחה כ"תרופה" לחסרונות שונים של גישת הנעילות. על פי הגדרתה, היא נמנעת מנעילות כאשר בדרך כלל התחליף הוא פקודות מכונה אטומיות כגון Compare-and-swap או Load-linked/Store-conditional. תכניות ללא נעילות אינן יכולות להיתקע ונמנעות מתסריטים שבהם חוטי-חישוב ממתינים למשאב כלשהו להתפנות. במהלך העשור האחרון, חלה ירידה הדרגתית בקשיחותם של התנאים הנחשבים כבסיסיים עבור ביצוע ללא נעילות. תכונת ה-Wait-Freedom המקורית דרשה התקדמות מכל אחד מחוטי החישוב, בעוד תכונת ה-Lock-Freedom דרשה זאת רק מחוט חישוב אחד לפחות. בהמשך, ה-Obstruction-Freedom הוצעה, והיא כבר מאפשר תקיעות חיות (livelocks) וההתקדמות תלויה במנגנוני ניהול התחרות נפרדים. קהילות המחקר והתעשייה הנמיכו את הסף למה שמקובל להיות ללא-נעילות מסיבות טובות: הורדת התקורה של ניהול ההתקדמות מאפשרת תכניות פשוטות יותר ומהירות יותר.

אחת מתכונות המפתח של סינכרוניזציה ללא נעילות היא היכולת לשרוד כשלים ארוכים או אפילו תמידיים של חלקים מהמערכת ללא השתהויות ארוכות. כדי

לספר יכולת זו, פעילות כתיבה של תהליכים על זיכרון משותף צריכות להיות מתועדות על ידי החוט הפעיל כדי שלא יתקע חוטים אחרים אם יתעכב או ייכשל. הפיכת פרטי הביצוע להיות נגישים ונראים על ידי חוטי חישוב אחרים מוסיפה לעלויות התקורה של מימושים ללא נעילות. לאחרונה אף נטען על ידי חוקרים בתחום שנעילה נמוכת גרעיניות היא אלטרנטיבה סבירה להימנעות מנעילות מכיוון שאין בה את החיסרון האינהרנטי שלהלן. בשימוש בנעילות עדיין יש לנקוט בזהירות ולמנוע תקיעות (deadlocks), וכן ישנה רגישות רבה יותר לכשלים-לצמיתות של מעבדים.

בעבר, מוקד המחקר היה על פתרונות ייחודיים עבור קבוצה קטנה של מבני נתונים: מחסניות, תורים, רשימות מקושרות, טבלאות האש ועצי חיפוש. כיום ישנה הבנה שקיומו של פתרון כללי לגישה מקבילית הוא חיוני. מקביליות תוכל להיות בכל מקום רק כאשר מתכנתים בעלי כישורים רגילים יוכלו לכתוב תכניות מקביליות יעילות ובטוחות. פרדיגמת הזיכרון הטרנזקציוני צמחה כניסיון לספק פלטפורמה קלה לשימוש ויעילה לתכנות מקבילי. במהלך השנים האחרונות הוצעו מבחר של פתרונות תוכנה, חומרה, ושילובי חומרה ותוכנה לזיכרון טרנזקציוני. בנייה זו מאפשר לבצע בלוקים של קוד כטרנזקציה, כלומר, עם הבטחה שאפקט הביצוע יהיה אטומי על הנתונים המשותפים. עקרון הזיכרון הטרנזקציוני מאפשר ריצה מקבילית של חוטי חישוב על אותו סט נתונים, בתנאי שאינם ניגשים לקבוצות נתונים חופפות בזמן ובמרחב הזיכרון. כל עוד חוטים פועלים על קבוצות זרות של כתובות זיכרון, הם לא מעכבים אחד את השני. יתרה מכך, ברוב המימושים של זיכרון טרנזקציוני, חוטים החופפים רק בכתובות מהן הם קוראים בלבד גם יכולים להתקדם במקביל. מימושים רבים מאפשרים טרנספורמציה קלה של קוד קיים עם נעילות גסות לקוד טרנזקציוני, ובכך מורידים את רמת המומחיות הנדרשת ממתכנתים של אפליקציות עשירות במקביליות.

הממשק שמספקות סכמות הזיכרון הטרנזקציוני הוא סותר באופן בסיסי את היכולות של חומרה קונוונציונלית. מבחינה פיסיקלית, אין אפשרות יעילה לבצע מספר בלתי מוגבל של עדכוני זיכרון באופן אטומי, במיוחד לאור השימוש במנגנוני זיכרון מטמון מודרניים. בשל כך, מספר סכמות של זיכרון טרנזקציוני מערבות שינויים בחומרה כאשר אחרות מבצעות אמולציה בתוכנה. נכון להיום, פתרונות חומרה טרם הבשילו ושולבו במוצר והם עדיין רחוקים מלהיות זמינים לקהילת המפתחים. לעומת זאת, מספר מימושי תוכנה של זיכרון טרנזקציוני שוחררו

לשימוש ציבורי. בכלליות, מימושים בחומרה מתמודדים עם תמיכה בטרנזקציות שאינו חסומות בזמן קצר ובגודל התלוי במשאבי חומרה, ובמימושי תוכנה מנסים למזער את התקורה הבלתי נמנעת.

עד לאחרונה, רוב פתרונות התוכנה שהוצעו פעלו בגישת הסינכרוניזציה ללא נעילות בכדי להשיג בקלות את יתרונות הגישה: הימנעות מתקיעות, המתנות ארוכות והתאמה למגוון רחב של מבני נתונים. הניסיון להשיג את אוסף המטרות האלו גבה מחיר של ביצועים: התקורה של אותם מנגנונים שהוצעו הייתה משמעותית בהשוואה לאלגוריתמים סדרתיים. בשל כך, גברו הקולות בזכות גישה הנעילות בגרנולריות נמוכה למימוש טרנזקציות בתוכנה, למרות היותה תיאורטית יותר רגישה להתנהגויות תזמון תהליכים, וזאת בשל הפוטנציאל להוריד את עלויות התקורה הטמון בפשטות היחסית של מנעולים.

המטרה העיקרית של תזה זו היא להציג אוסף של גישות חדשניות שיסייעו לקידומם של שיטות תכנות מתקדמות, כגון מבני נתונים ללא נעילות ותכנות טרנזקציוני. בכדי שכלים אלו יתקבלו על ידי קהילות המפתחים, הם חייבים (1) להציע ביצועים תחרותיים הן למקרה הסדרתי והן למקבילי, (2) להיות בלתי תלויים במיומנויות תכנות שאינן בסיסיות, ו-(3) להיות תואמים לפלטפורמות תוכנה וחומרה קיימות. מטרות אלו הן בעצם המוטיבציה של עבודה זו.

המחקר בסינכרוניזציה ללא נעילות ובזיכרון טרנזקציוני התחיל בנקודה מאוד רחוקה מהמטרות שלהלן, ומאז חלה התקדמות משמעותית לעברן. חוקרים שנקטו בגישת הסינכרוניזציה ללא נעילות ונאבקו עם מבני נתונים כמו טבלאות האש ועצים בינאריים, אשר רמת הסיבוך שלהם גבוהה משל מחסניות ותורים. בין השאר, בשל כך החלה המגמה של חיפוש פתרונות הסינכרוניזציה הכוללניים, בדמות זיכרון טרנזקציוני.

בתזה זו, תחילה אציג מימוש של טבלת האש הניתנת להרחבה – אלגוריתם מקבילי ייחודי ללא נעילות המבוסס על עקרון חדשני בשם "סידור מפוצל". ביצועי האלגוריתם מתחרים עם אלטרנטיבות מבוססות נעילות.

טכניקת הסידור המפוצל נוצרה מהצורך להתגבר על הקושי להעביר איברים מתא לתא בטבלאות האש בצורה אטומית כאשר מרחיבים את הטבלה. באופן מטאפורי, משמעו של סידור מפוצל בטבלת האש הוא לסדר את התאים בין האיברים במקום

לסדר את האיברים בתאים. שלא כמו הזזת איבר מתא לתא, הפעולה של פיצול תא יכולה כעת להיעשות על ידי פעולת מכונה אטומית אחת (CAS). מכיוון שהאיברים נשארים במקומם בעוד התאים "זזים", אין חשש שייווצר חוסר קונסיסטנטיות במצב הטבלה שהיה עשוי להיווצר אם איברים היו מועברים מתא לתא תוך כדי שינויים בתזמון תהליכים. כדי שגישה זו תעבוד, יש לשמור את כל האיברים ברשימה אחת מסודרים על פי סדר מיוחד, סדר שתואם פיצול עתידי של תאי הטבלה: כאשר הטבלה תורחב ויהיה צורך לפצל תאים, על האיברים שבתוכם להיות מוכנים לאותו פיצול מראש. כאשר האיברים מסודרים בסידור מפוצל, אפשר לפצל תא על ידי פעולה פשוטה של הפניית מצביע התא החדש למרכז התא המקורי. כל תא יכול להתפצל שוב ושוב, באופן רקורסיבי, ובכך בעצם מוגדר סדר מלא על איברי הרשימה.

העבודה הניסויית שביצעתי על טבלאות האש בהספקים גבוהים הובילה לאבחנות על עיצוב של מנגנונים כלליים לסינכרוניזציה, הדומים ביכולתם לזיכרון טרנזקציוני בתוכנה. רעיונות אלו התפתחו לכדי מספר שיטות פרקטיות לגישה כללית למבני נתונים משותפים בחישוב מרובה מעבדים.

גישה אחת שפיתחתי, "סינכרוניזציית לוג", מבחינה בין פעולות שמבצעות שינוי במבנה הנתונים לבין פעולות של קריאה בלבד, שבדרך כלל הרבה יותר נפוצות ברוב האפליקציות. סינכרוניזציית לוג מבוססת על לוג (יומן) גלובלי של פעולות רמה-גבוהה המשנות את מצב הנתונים. פעולות אלה מתבצעות אחת אחרי השנייה באופן סדרתי, אך החידוש הוא שתוך כדי פעולתן מתאפשרת עבודה כמעט חסרת תקורה של פעולות שקוראות בלבד. מכיוון שבאפליקציות רבות פעולות שקוראות בלבד הן נפוצות משמעותית, יש פוטנציאל שיפור לביצועים. אציג שתי ואריאציות של סינכרוניזציית לוג.

על פי הראשונה, "סינכרוניזציית לוג תחזיתית", הריצה היא ללא נעילות, כאשר חוטי חישוב מסוגלים לחזות את תוצאת הפעולה שלהם על פי הלוג ולהמשיך הלאה לביצוע של פעולות נוספות. החוטים משתמשים בתיאור הפעולות הרשומות בלוג כדי לחזות את התוצאה של הפעולה שלהם מעכבת אותם מלהמשיך בריצה. על פי שיטה זו, מבנה הנתונים כולו משוכפל לשני עותקים, כאשר אחד משמש לביצוע פעולות כתיבה והשני מאפשר פעולות קריאה במקביל. שני העותקים מחליפים תפקידים ומתעדכנים בתדירות גבוהה. חוט חושב אחד שתופס מנעול

גלובלי הוא זה שמבצע את פעולות הכתיבה הרשומות בלוג על העותק לכתיבה, וכאשר הוא מסיים, הוא משחרר את המנעול הגלובלי. סינכרוניזציית לוג תחזיתית עומדת בקריטריון של ללא-נעילות מכיוון שחוטי חישוב שלא זכו במנעול מסוגלים עדיין להתקדם על ידי בחינת הלוג המכיל את כל המידע הדרוש כך שיוכלו להמשיך בפעולתם. במקביל חוטי חישוב יכולים לבצע כמעט ללא תקורה פעולות קריאה בלבד מהעותק לקריאה.

השיטה השנייה לסינכרוניזציית לוג היא "סינכרוניזציית לוג מבוססת כתובת", שבה הלוג מכיל את כתובות הזיכרון המדויקות אליהן תהליכים כותבים וקוראים. גם כאן, הכתיבות עצמן מתבצעות באופן סדרתי על ידי מעבר על לוג הפעולות וביצוע הכתיבות. מכיוון שטרנזקציות שונות עשויות להתנגש אחת עם השנייה, יש צורך לזהות קונפליקטים אלה בלוג. חוטי חישוב המבצעים פעולות כתיבה, רצים תחילה באופן "וירטואלי", בו הם רק מתעדים את הכתיבות שלהם בלוג. חוט אחד בלבד יכול לתפוס מנעול גלובלי, ואז לזהות קונפליקטים בלוג, לבטל טרנזקציות מתנגשות ולבצע בפועל את הכתיבות על מבנה הנתונים. כדי לאפשר קריאה במקביל ממבנה הנתונים, כל כתובות הזיכרון הניתנות לגישה מקבילית משוכפלים לשני עותקים (בניגוד לסינכרוניזציית לוג תחזיתית שבה כל מבנה הנתונים משוכפל). גם גישה זו מיועדת להרצות בהם פעולות שקוראות בלבד הם יחסית דומיננטיות, שכן התקורה בביצוען היא אפסית.

במאמץ לשפר את יכולת ההרחבה של פרדיגמת סינכרוניזציית הלוג, ובניסיון להמיר את הלוג עצמו למונה גרסאות פשוט, נוצר הרעיון של אלגוריתם המבוסס נעילות בגרעיניות נמוכה למימוש זיכרון טרנזקציוני בתוכנה שנקרא "נעילה טרנזקציונית 2". על ידי שימוש בשעון גלובלי, קונפליקטים בין טרנזקציות מזוהים בקלות ובאופן מיידי. כמו כן, אחת הבעיות הקלאסיות במימוש זיכרון טרנזקציוני, שהיא הרגישות למצבים לא קונסיסטנטיים, נפתרת ב"נעילה טרנזקציונית 2" בצורה אלגנטית וללא תקורה גבוהה. על פי אלגוריתם זה, חוט חישוב מתחיל ביצוע טרנזקציה בדגימת השעון הגלובלי, ממשיך בריצה ספקולטיבית של קוד הטרנזקציה שבה כל גישה לזיכרון מלווה בבדיקה מול זמן תחילת הטרנזקציה שנדגם. בכדי לסיים את הטרנזקציה, יש לרכוש נעילה על כל אחד מהמקומות לכתיבה, להשיג תג-זמן ייחודי מהשעון הגלובלי, לבצע וידוא נוסף, לעדכן את מספר הגרסה במנעול ולשחררו.

בניגוד למערכות הזיכרון הטרנזקציוני בתוכנה שהוצעו בעבר, "נעילה טרנזקציונית 2" נמנעת ממצבים בלתי קונסיסטנטיים בצורה נקייה וללא תקורה גבוהה. האלגוריתם מאפשר טרנזקציות קריאה-בלבד מאוד יעילה, בכך שאין צורך לבצע כל וידוא אחרי ריצה אחת בלבד של הטרנזקציה. פעולות קריאה בלבד יכולות להסתפק בהשוואות לשעון הגלובלי המתבצעות במהלך הריצה, ועצם ההגעה לסוף הקוד מסמנת ביצוע מוצלח. כמו כן, בניגוד למערכות זיכרון טרנזקציוני אחרות בתוכנה, נעילה טרנזקציונית 2 מאפשרת ניהול זיכרון גמיש בכך שאינה מחייבת מערכת ניהול זיכרון נפרדת.

התיזה מבוססת על המאמרים הבאים:

Ori Shalev and Nir Shavit. **Split-ordered lists: Lock-free extensible hash tables**. *J. ACM*, 53(3):379–405, 2006.

Ori Shalev and Nir Shavit. **Predictive log-synchronization**. In *EuroSys 2006*, pages 305–315, Apr 2006.

Dave Dice, Ori Shalev, and Nir Shavit. **Transactional locking II**. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, September 2006.