Exploiting Parallelism of Multi-Core Architectures

Dmitri Perelman

Exploiting Parallelism of Multi-Core Architectures

Research Thesis

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Dmitri Perelman

Submitted to the Senate of the Technion – Israel Institute of Technology
Elul 5772 Haifa September 2012

The Research Was Done Under the Supervision of Prof. Idit Keidar in the Department of Electrical Engineering, Technion.

THE GENEROUS FINANCIAL HELP OF THE TECHNION — ISRAEL INSTITUTE OF TECHNOLOGY IS GRATEFULLY ACKNOWLEDGED

Publications:

- V. Bortnikov, G. Chockler, D. Perelman, A. Roytman, S. Shachor, and I. Shnayderman. Brief announcement: reconfigurable state machine replication from non-reconfigurable building blocks. In *Proceedings of the 31 ACM Symposium on Principles of Distributed Computing*, PODC '12
- E. Gidron, I. Keidar, D. Perelman, and Y. Perez. SALSA: scalable and low synchronization NUMA-aware algorithm for producer-consumer pools. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12
- D. Perelman, E. Bortnikov, R. Lempel, and R. Sandler. Lightweight automatic face annotation in media pages. In *Proceedings of the 21st World Wide Web Conference*, WWW '12
- D. Perelman, A. Byshevsky, O. Litmanovich, and I. Keidar. SMV: Selective multi-versioning STM. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, DISC '11
- D. Basin, R. Fan, I. Keidar, O. Kiselov, and D. Perelman. CAFÉ: Scalable task pools with adjustable fairness and contention. In *Proceedings of the 25th International Symposium on Principles of Distributed Computing*, DISC '11
- V. Bortnikov, G. Chockler, D. Perelman, A. Roytman, S. Shachor, and I. Shnayderman. FRAPPE: Fast replication platform for elastic services. In *Proceedings of the 5th Workshop on Large Scale Distributed Systems and Middleware*, LADIS '11, 2011
- D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, PODC '10
- D. Perelman and I. Keidar. SMV: Selective multi-versioning stm. In *5th ACM SIGPLAN Workshop on Transactional Computing*, TRANSACT '10, 2010
- I. Keidar and D. Perelman. On avoiding spare aborts in transactional memory. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 59–68, 2009

Acknowledgments

I'd like to thank Idit Keidar, my supervisor, teacher and friend (if some articles are missing in this paragraph, it is because she didn't pass over the acknowledgments section :)). Due to Idit I started the graduate studies and I feel very lucky that I had an opportunity to work with her all these years. Her passion and professionalism will always stay an example for me.

I thank my friends and colleagues for making the Technion such a great place to spend PhD years. DimaB (aka Dimanych), Ittay, Eyal, Elad, Alex, Max, Zvika, Liat, Nathaniel, Oved, Stacy, Rui, Yaniv: I'm looking forward to see you again, hummus is on me :). I thank Vita and Grisha from IBM labs and Eddie from Yahoo! labs for their great attitude. I really enjoyed working with you and I'm sure we'll stay in touch. I thank Victor Kulikov for our coffee breaks and the Greg staff for small cappuccinos.

I'm grateful to Hasso Plattner Institute for their generous financial support.

I want to thank my parents for their love, for bringing me up the way I am and for their non-stopping care. I'm also very grateful to my wife's parents for their enormous help and support.

I devote this thesis to my beautiful wife Anna whose love and patience lead me through all these years and to my dear daughters Yana and Dina.

Contents

List of Figures

List of Tables

Abstract

As multi-core computer architectures are becoming mainstream, it is widely believed that one of the biggest challenges facing computer scientists today is learning how to exploit the parallelism that such architectures can offer. This work deals with efficient synchronization and data exchange among threads.

The first part of this thesis addresses theoretical and practical issues of transactional memory (TM), a new synchronization abstraction, which allows threads to bundle multiple operations on memory objects into transactions. Similarly to database transactions, TM transactions are executed *atomically*: either all of the transaction's operations appear to take effect simultaneously (in this case, we say that the transaction *commits*), or none of transaction's operations are seen (in this case, we say that transaction *aborts*).

Existing TMs may abort many transactions that could, in fact, commit without violating correctness. We call such unnecessary aborts *spurious aborts*. We classify what kinds of spurious aborts can be eliminated, and which cannot. We further study what kinds of spurious aborts can be avoided efficiently. Specifically, we show that some unnecessary aborts cannot be avoided, and that there is an inherent tradeoff between the overhead of a TM and the extent to which it reduces the number of spurious aborts. We also present a polynomial-time sample TM algorithm that avoids certain kinds of spurious aborts and analyze its properties and performance.

An effective way to reduce the number of aborts in transactional memory is to keep multiple versions of transactional objects. We therefore study inherent properties of STMs that use multiple versions to guarantee successful commits of all read-only transactions. We first show that these STMs cannot be disjoint-access parallel. We then consider the problem of garbage collecting old object versions, and show that no STM can be optimal in the number of previous versions kept. Moreover, we show that precise garbage collecting of useless versions is impossible in STMs

implemented with invisible reads. As an example, we present a theoretical sample STM algorithm that uses visible reads and efficiently removes object versions once they become useless.

We refer to the practical implications of excessive aborts and multi-versioning by developing *Selective Multi-Versioning (SMV)*, a multi-versioned STM that reduces the number of forceful aborts, especially those of long read-only transactions. SMV efficiently deals with old versions while still allowing invisible read-only transactions by relying on automatic garbage collection. It achieves $\times 7$ throughput improvement over a single-version STM and more than a two-fold improvement over an STM keeping a constant number of versions per object. Moreover, we show that the memory consumption of algorithms keeping a constant number of versions per object might grow exponentially, while SMV operates successfully even in systems with stringent memory constraints.

Another important aspect of multi-core programming is the problem of efficient data exchange among threads. We present a highly-scalable non-blocking producer-consumer task pool, designed with a special emphasis on lightweight synchronization and data locality. The core building block of our pool is *SALSA, Scalable And Low Synchronization Algorithm* for a single-consumer container with task stealing support. Each consumer operates on its own SALSA container, stealing tasks from other containers if necessary. We implement an elegant self-tuning policy for task insertion, which does not push tasks to overloaded SALSA containers, thus decreasing the likelihood of stealing. SALSA uses a novel approach for coordination among consumers, without strong atomic operations or memory barriers in the fast path. It invokes only two CAS operations during a chunk steal. Our evaluation demonstrates that a pool built using SALSA containers scales *linearly* with the number of threads and significantly outperforms other FIFO and non-FIFO alternatives.

Chapter 1

Introduction

During the past decade we could witness the crucial changes in the programming paradigms. While the hardware architectures evolved by increasing the number of computing elements and their heterogeneity, the software world had to adapt to the new demands and to face the growing complexity and the increasing parallelism. Development of scalable programs stopped being a niche of a few professionals: thousands of "mere mortal programmers" are demanded to build highly efficient parallel applications. This situation raises a need for devising new tools that help to deal with the software development in the multi-core era.

Efficient synchronization. One of the main software challenges in the multi-core world is an efficient synchronization of multithreaded programs. Conventional locking solutions introduce a host of well-known problems: coarse-grained locks are not scalable, while fine-grained locks are error-prone and hard to design. Transactional memory [48] has gained popularity as a new synchronization abstraction for multithreaded systems, which has the potential to overcome the pitfalls of traditional locking schemes. A *transactional memory* toolkit, or *TM* for short, allows threads to bundle multiple operations on memory objects into one transaction. Similar to database transactions [77], transactions are executed *atomically*: either all of the transaction's operations appear to take effect simultaneously (in this case, we say that the transaction *commits*), or none of transaction's operations are seen (in this case, we say that transaction *aborts*). The model and correctness criterion of transactional memory are formally defined in Chapter 3.

Many existing TMs abort transactions that could commit without violating correctness of the program. Such unnecessary aborts might degrade the overall performance and decrease the predictability of the program. Chapter 4 aims to advance the theoretical understanding of TM aborts, by studying what kinds of spurious aborts can or cannot be eliminated, and what kinds of spurious aborts can or cannot be avoided efficiently. Specifically, we show that some unnecessary aborts cannot be avoided, and that there is an inherent tradeoff between the overhead of a TM and the extent to which it refrains from spurious aborts. We then demonstrate a polynomial-time sample algorithm that avoids most unnecessary aborts and analyze its properties.

A necessary technique for reducing the number of aborts in STM is the usage of multiple versions for transactional objects. Multi-versioning is especially useful for read-only transactions: by keeping enough versions it is possible to assure that each read-only transaction successfully commits by reading a consistent snapshot of the object it accesses. Chapter 5 focuses on the theoretical properties of the algorithms that assure successful commit of read-only transactions. We show that such algorithms cannot be disjoint-access parallel (DAP) and study their garbage collection limitations. Amongst others, we claim that a simple approach of keeping a constant number of versions for each object can cause an exponential memory growth, and then we prove that no STM can be space optimal, i.e., no STM can ensure that it always maintains the minimum number of object versions possible.

Chapter 6 presents a practical multi-versioned STM called *Selective Multi-Versioning (SMV)*. SMV is progressive and it ensures that each read-only transaction successfully commits. Our STM keeps old object versions as long as they might be useful for a live read-only transaction to read. It is able to do so while still allowing reading transactions to be invisible by relying on automatic garbage collection to dispose of obsolete versions. SMV is most suitable for read-dominated workloads, for which it performs better than previous solutions. It has an up to $\times 7$ throughput improvement over a single-version STM and more than a two-fold improvement over an STM keeping a constant number of versions per object, while operating successfully even in systems with stringent memory constraints.

Efficient data exchange. An additional challenge posed by the emerging multi-core architectures is data exchange among threads. A fundamental data structure for transferring tasks in a parallel computation is a producer-consumer task pool. It is thus highly important to ensure that such a pool does not become a scalability bottleneck when concurrently accessed by large number of threads.

Chapter 7 presents a scalable and highly-efficient task pool with lightweight synchronization-free operations in the common case. The data allocation scheme of our pool is cache-friendly and especially suitable for NUMA architectures. The core building block of the pool is *SALSA, Scalable And Low Synchronization Algorithm* for a single-consumer container with stealing support. SALSA's chunk-based stealing algorithm allows consume operations to be synchronization-free when no stealing occurs, s.t. neither producers nor consumers invoke strong atomic operations in the common case.

In many-core machines running multiple applications, system behavior becomes less predictable. Unexpected thread stalls may lead to an asymmetric load on consumers, which may in turn lead to high stealing rates, hampering performance. SALSA employs a novel auto-balancing mechanism that has producers insert tasks to less loaded consumers, and is thus robust to spurious load fluctuations.

Our evaluation demonstrates that SALSA-based pools significantly outperform state-of-the-art FIFO and non-FIFO alternatives.

Chapter 2

Related Work

In Section 2.1 we review previous work on using multiple versions for avoiding aborts in STM. Section 2.2 describes previous works on concurrent task pools.

2.1 Transactional Memory

The transactional memory programming paradigm was first introduced by Herlihy and Moss [48] as a hardware proposal for combining sequences of concurrent operations into atomic transactions. Since then, the idea has gained a lot of popularity and currently there exists multiple software [29, 69, 30, 22], hardware [27, 76, 56], and combined [75] TM implementations.

Multi-Versioning in TM. Most existing TM implementations, e.g., [47, 35, 32, 29] abort one transaction whenever two overlapping transactions access the same object and at least one access is a write. While easy to implement, this approach may lead to high abort rates, especially in situations with long-running transactions and contended shared objects.

Historically, one of the commonly used methods for reducing the number of aborts was maintaining multiple object versions. Multiversion concurrency control is a classical approach for providing concurrent access to the database in database management systems [16, 63]. Its idea is to let a reading transaction obtain a consistent snapshot corresponding to an arbitrary point in time (typically defined at the beginning of a transaction) – concurrent updates are isolated through maintaining old versions rather than through a process of locks or mutexes.

Multi-versioning technique was adopted by transactional memory algorithms as well. Aydonat and Abdelrahman [11] proposed a solution based on a conflict serializability graph and multi-versioned objects in order to reduce the number of unnecessary aborts. However, their solution still induces spurious aborts, and does not characterize exactly when such aborts are avoided. Moreover, they implement a stricter correctness criterion than opacity, which inherently requires more aborts. Napper and Alvisi [61] described a serializable TM, based upon multi-versioned objects, which used cycle detection in the precedence graph when validating the correctness criterion. The focus of the paper was providing a lock-free solution. The authors did not refer to the aspect of spurious aborts and, in fact, their TM did lead to spurious aborts due to a limitation on write operation, which had to insert the new version after the latest one. In addition, Napper and Alvisi did not refer to the problems of garbage collection and computational complexity of operations.

Formal notions for quantifying aborts. Gramoli et al. [40] referred to the problem of spurious aborts and introduced the notion of *commit-abort ratio*, which is the ratio between the number of committed transactions and the overall number of transactions in the run. Clearly, the commit-abort ratio depends on the choice of the transaction that should be aborted in case of a conflict. This decision is the prerogative of a contention manager [47]. Attiya et al. [6] showed a $\Omega(s)$ lower bound for the competitive ratio for transactions' makespan of any online deterministic contention manager, where s is the number of shared objects. Their proof, however, does not apply to our model, because it is based upon the assumption that whenever multiple transactions need exclusive access to the same shared object, only one of these transactions may continue, while others should be immediately aborted. In contrast, our model allows the TM to postpone the decision regarding which transaction should be aborted till the commit, thus introducing additional knowledge and improving the competitive ratio. In this paper, we show that no TM can obtain a commit-abort ratio achieved by an optimal offline algorithm. This result suggests that it is not interesting to compare (online) TMs by their commit-abort ratio, as the distance from the optimal result turns out to be an artifact of the workload rather than the algorithm, and every TM has a workload on which it performs poorly by this measure.

Input acceptance is also a notion presented by Gramoli et al. [40] — a TM *accepts* a certain input pattern (sequence of operational invocations) if it commits all of its transactions. The authors compared different TMs according to their input acceptance patterns. Guerraoui et al. [41]

introduced the related notion of π-*permissiveness*. Informally, a TM satisfies π-permissiveness for a correctness criterion π, if every history that does not violate π is accepted by the TM. Thus, π-*permissiveness* can be seen as optimal input acceptance. However, Guerraoui et al. focused on a model with single-version objects, and their correctness criterion was based upon conflict serializability, which is stronger than opacity and thus allows more aborts. They ruled out the idea of ensuring permissiveness deterministically, and instead provide a randomized solution, which is always correct and avoids spurious aborts with some positive probability. In contrast, we do not limit the model to include single-version objects only, our correctness criterion is a generalization of *opacity* [43], and we focus on deterministic guarantees. Although permissiveness does not try to regulate the decisions of the contention manager, we show that no online TM may achieve permissiveness. Intuitively, this results from the freedom of choice for returning the object value during the read operation — returning the wrong value might cause an abort in subsequent operations, which is avoided by a clairvoyant (offline) algorithm.

Garbage collection. Any practical multi-versioned STM has to address the problem of removing old object versions. Some earlier STMs, such as LSA [69], keep a fixed number of old object versions. However, this approach leads to storing versions that are too old to be of use to any transaction on the one hand, and to aborting transactions because they need older versions than the ones stored on the other. In contrast, SMV keeps versions as long as they might be useful for ongoing transactions, and makes them GCable by an automatic garbage collector as soon as they are not. For infrequently updated objects, SMV typically keeps a single version.

Another multi-versioned STM, JVSTM [22], maintains a priority queue of all active transactions, sorted by their start time. A cleanup thread waits until the transaction at the head of the queue (the oldest transaction) is finished. When that happens, the cleanup process iterates over the objects overwritten by the committed transaction and discards their previous versions. Thus, while also keeping versions only as long as active transactions might read them, the GC mechanism of JVSTM imposes an additional overhead for transaction startup and termination (including both update and read-only transactions).

In a recent paper [34], the authors improved the GC mechanism of JVSTM by maintaining a global list of per-thread transactional contexts, each keeping information about the latest needed versions. A special cleanup thread iterates periodically over this list and thus finds the versions that can be discarded. This improvement, however, does not eliminate the need for a special cleanup thread, which should run in addition to Java GC threads. JVSTM read-only transactions still need to write to the global memory. In contrast, this thesis presents a simple algorithm with invisible read-only transactions, which exploits the automatic GC available in languages with managed memory.

Multi-versioning alternatives. Instead of multi-versioning, STMs can avoid aborts by reading uncommitted values and then having the reader block until the writer commits [68], or by using read-write locks to block in case of concurrency [30, 9]. These approaches differ from the algorithms proposed in this thesis, where transactions never block and may always progress independently. Moreover, reads, which are invisible in SMV, must be visible in these "blocking" approaches. In addition, reading the values of uncommitted transactions might lead to consistency violations during transactions.

Transactional mutex locks (TML) [25], have been shown to be very efficient for read-dominated workloads due to their simplicity and low overhead. Unlike the multi-versioned algorithms presented in this thesis, TML do not allow concurrency between update transactions and thus do not exploit the parallelism in read-write or write-dominated workloads.

Another technique for reducing the number of aborts is timestamp extension [69, 33]. This mechanism requires maintaining a read-set and therefore is usually not used by read-only transactions. Timestamp extension is applicable for SMV's update transactions as well, hence this improvement is orthogonal to the multi-versioning approach presented in this work.

Impossibility of disjoint-access parallelism. An important technique for optimizing STM performance is disjoint-access parallelism. As described earlier, this means that transactions that do not access the same objects should also not access the same memory locations, thereby avoiding memory contention. Guerraoui and Kapalka [42] show that a single-versioned, obstruction-free [47] STM cannot be strictly DAP. However, their proof does not apply in the multi-versioned setting we consider.

Attiya *et al.* [10] show that there is no STM implementing DAP that uses invisible reads, in which read-only transactions always terminate. In Section 5.4.1, we show that no responsive MV-permissive STM can be DAP. As stated earlier, MV-permissiveness ensures all read-only transactions commit, and update transactions abort only when they conflict with other update transactions. Thus, our results show that the requirement of invisible reads in [10] can be replaced by precluding update transactions from aborting when they conflict with read-only transactions.

MV-permissiveness without responsiveness. Attiya and Hillel [8] present PermiSTM algorithm, which provides MV-permissiveness but is not responsive — every read-only transaction commits successfully; a transaction that tries to update the object, which has been read by an active reader, is blocked until the termination of this reader. By forfeiting the responsiveness PermiSTM succeeds to obtain the properties that cannot be achieved by UP-MV or any other responsive MV-permissive algorithm — it uses a single version per object and it is disjoint-access parallel. This way, UP-MV and PermiSTM demonstrate the degree of freedom that exists between the progress conditions and implementation overhead.

2.2 Concurrent Task Pools

Concurrent task pool is a ubiquitous data structure that has a number of important applications in multiprocessor computing, such as passing information among threads in a parallel computation. SEDA [78], a highly concurrent web server, excessively uses task pools for building its execution pipeline of requests. A key challenge is then to ensure that the pool does not become a bottleneck when it is concurrently accessed by a large number of threads.

Pool implementations. Consumer-producer pools are often implemented as FIFO queues. A widely used state-of-the-art FIFO queue is Micheal and Scott's queue [58]. This queue is implemented by a linked-list with *head* and *tail* references. The put operation adds a new node to the list and then updates the tail reference. This is done by two CAS operations; one for adding the new node and one for updating the tail reference. The get operation removes a node by moving the head reference to point to the next node. This approach is not scalable under high contention as only one contending operation may succeed.

Moir et al. [59] suggest using elimination to reduce the contention on the queue. Whereby put and get operations can eliminate each other during the back-off after an unsuccessful operation. However, due to the FIFO property, those eliminations can only be done when the queue is empty, making this approach useful only when the queue is close to empty.

Hoffman et al. [50] try to reduce the contention of the put operation by allowing concurrent put operations to add tasks to the same "basket". This is done by detecting contention on the tail, which is indicated by a failed CAS operation when trying to update the tail. This reduces the contention on the tail, but not on adding the node to the "basket", which still requires a CAS operation. Therefore, this approach, while more efficient than Micheal and Scott's queue, is still not scalable under high contention.

Gidenstam et al. [36] use a similar approach to Micheal and Scott's, but, in order to improve locality and decrease the contention on the head and tail, the data is stored in chunks, and the head and tail points to a chunk rather than single nodes. This allows updating these references only once per-chunk rather than on every operation. However, this solution still requires at least one CAS per operation, rendering it non-scalable under high contention.

A number of previous works have recognized this limitation of FIFO queues, and observed that strict FIFO order is seldom needed in multi-core systems.

Afek et al. [3] implemented a non-FIFO pool using diffraction trees with elimination (ED-pools). An ED-pool is a tree of queues, which contains elimination arrays that reduce contention. While ED-pools scale better than FIFO based solutions, they do not scale on multi-chip architectures [12].

Basin et al. [14] suggest a wait-free task-pool that allows relaxing FIFO. This pool is more scalable than previous solutions, but, since it still has some ordering (fairness) requirements, there is contention among both producers and consumers.

The closest non-FIFO pool to the work presented in Chapter 7 is the Concurrent Bags of Sundell et al. [74], which, like SALSA, uses per-producer chunk lists. This work is optimized for the case that the same threads are both consumers and producers, and typically consume from themselves, while SALSA improves the performance of such a task pool in NUMA environments where producers and consumers are separate threads. Unlike our pool, the Concurrent Bags algorithm uses strong atomic operations upon each consume. In addition, steals are performed in the granularity of single tasks and not whole chunks as in SALSA. Overall, their throughput does not scale linearly with the number of participating threads, as shown in [74] and in Section 7.4.

To the best of our knowledge, all previous solutions use strong atomic operations (like CAS), at least in every consume operation. Moreover, most of them [3, 4, 14] do not partition the pool among processors, and therefore do not achieve good locality and cache-friendliness, which has been shown to limit their scalability on NUMA systems [12].

Implementation techniques. Variations of techniques we employ were previously used in various contexts. Work stealing [18] is a standard way to reduce contention by using individual per-consumer pools, where tasks may be stolen from one pool to another. We improve the efficiency of stealing by transferring a chunk of tasks upon every steal operation. Hendler et al. [46] have proposed stealing of multiple items by copying a range of tasks from one dequeue to another, but this approach requires costly CAS operations on the fast-path and introduces non-negligible overhead for item copying. In contrast, our approach of chunk-based stealing coincides with our synchronization-free fast-path, and steals whole chunks in O(1) steps. Furthermore, our use of page-size chunks allows for data migration in NUMA architectures to improve locality, as done in [17].

The principle of keeping NUMA-local data structures was previously used by Dice et al. for constructing scalable NUMA locks [28]. Similarly to their work, our algorithm's data allocation scheme is designed to reduce inter-chip communication.

The concept of a synchronization-free fast-path previously appeared in works on scheduling queues, e.g., [5, 45]. However, these works assume that the same process is both the producer and the consumer, and hence the synchronization-free fast-path is actually used only when a process transfers data to *itself*. Moreover, those work assume a sequentially consistent shared-memory multiprocessor system, which requires insertion of some memory barrier instructions to the code when implemented on machine providing a weaker memory model [7]. On the other hand, our pool is synchronization-free even when tasks are transfered among multiple threads; our synchronization-free fast-path is used also when multiple producers produce data for a single consumer. We do not know of any other work that supports synchronization-free data transfer among different threads.

The idea of organizing data in chunks to preserve locality in dynamically-sized data structures was previously used in [21, 36, 45, 74]. SALSA extends on the idea of chunk-based data structures by using chunks also for efficient stealing.

Chapter 3

Model and Notations

We consider a shared memory environment where execution threads communicate with each other using *shared memory*, and each thread also has *private memory* which it alone can access. The scheduler can suspend a thread, for an arbitrary duration of time, at any moment after termination of a basic processor instruction (read, write, CAS). Threads cannot be suspended in the middle of a basic instruction. In modern architectures read and write operations may be reordered unless explicitly using a fence operation. However, in our model we assume sequential execution of instruction per-thread. The reordering problems are solved by using implicit fences when using CAS, or by the technique explained in 7.4.1.

Transactions. A *transaction* consists of a sequence of *transactional operations*, where each operation is comprised of an invocation step and a subsequent matching response step, collectively called *transactional steps*. The system contains a set of *transactional objects*. Each transactional operations either accesses a transactional object, or tries to commit or abort the transaction. More precisely, let T be a transaction, o be a transactional object, and v be a value. Then a transactional operation is one of the following. (1) An invocation step *start*(T), followed by response S, meaning T is started. (2) An invocation step *read*(T, o), followed by a response step that either gives the current value of o, or responds A, meaning that the transaction is *aborted*. (3) An invocation *write*(T, o, v), followed by a response either acknowledging the write, or responding A. (4) An invocation *Abort*(T), followed by response A (abort operation). (5) An invocation *Commit*(T), followed either by response C, meaning T committed, or A.

We say the *read set*, resp. *write set* of a transaction is the set of transactional objects read, resp. written to by T. We say T is *read-only* if its write set is empty. An update transaction is any transaction that is not read-only. We say two transactions *conflict* if they both access a common transactional object, and at least one of the accesses is a write. We assume that the steps in a transaction are not known ahead of time, but it is known a priori whether a transaction is a read-only or update transaction. Detection of read-only behavior can be done at compile time or using programmer annotations.

Transaction histories. A *transactional history* H is a sequence of transactional steps, interleaved in an arbitrary order (in the rest of the thesis we use the notion of *run* as a synonym to a transactional history). A transaction is *active* in H if it is neither committed nor aborted, it is *complete* otherwise. A transaction can perform operations as long as it is active. Each transaction has a unique identifier (id). Retrying an aborted transaction is interpreted as creating a new transaction with a new id.

Two histories H_1 and H_2 are *equivalent* if they contain the same transactions and each transaction T_i issues the same operations in the same order with the same responses in both. A history H is *complete* if it does not contain active transactions. If history H is not complete, we may build from it a complete history *Complete(H)* by adding an abort operation for every active transaction. We define *committed(H)* to be the subsequence of H consisting of all the operations of all the committed transactions in H.

The real-time order on transactions is as follows: if the first operation of transaction T_i is issued after the last response of transaction T_j in H, then T_j precedes T_i in H, denoted $T_j \preceq_H T_i$. Transactions T_i and T_j are *concurrent* if neither $T_j \preceq_H T_i$, nor $T_i \preceq_H T_j$. A transactional history S is *sequential* if it has no concurrent transactions. S is *legal* if it respects the sequential specification of each transactional object accessed in S. Transaction T_i is legal in S if the largest subsequence S' of S, such that for every transaction $T_k \in S'$, either (1) $k = i$, or (2) T_k is committed and $T_k \prec_S T_i$, is a legal history.

Transactional memory. A *transactional memory (TM)* is an algorithm for running transactions. We do not consider any kind of transactional nesting. Each transaction is *run* by a thread, and each thread runs at most one transaction at a time. To run a transaction T, a thread runs each of T's transactional operations, as follows. (1) Take as input an invocation step of T. (2) Perform a sequence of private and shared memory steps, which are determined by the input and the memory. (3) Return as output a response step to T. We write $thr(T)$ for the thread running T. A transactional memory can forcefully abort transaction T_j as a result of invocation step of another transaction T_i. In this case we say that T_j is aborted and the next operation invocation of T_j returns A.

We call the memory objects accessed by the threads *base objects*. Note that these are conceptually distinct from the transactional objects accessed by the transactions. We also call the steps performed by the threads *base steps*. We assume that all the base steps for running a transactional step appear to execute atomically. The means by which such linearizability of transactional steps is achieved lies beyond the focus of this thesis. In practice, it can be achieved using locks (like the two-phase locking mechanism used in commit operations by TL2 [29]), or by lock-free algorithms [35]. Due to the assumption of atomicity of transactional steps we consider only the well-formed histories in which an invocation of transactional operation is followed by the corresponding response.

We say that a TM is *responsive* if it guarantees that each operation invocation eventually gets a response, even if all other threads do not invoke new transactional operations. This limits the responsive TM's behavior upon operation invocation, so that it may either return an operation response, or abort a transaction, but cannot wait for other transactions to invoke new transactional operations. Note that we do allow for a responsive TM to wait for concurrent transactional operations to complete, for example TL2 [29] is responsive in spite of the use of locks. One may say that a responsive TM provides lock-freedom at the level of transactional operations.

A *configuration* of a TM consists of the states of the shared memory, private memory, and threads. An *execution* of a TM is an alternating sequence of configurations and base steps, starting with a configuration in which the memory and threads are all in their initial states. Two executions are *indistinguishable* to a thread if it performs the same sequence of state changes in both executions. Given a configuration C and a transaction T, we let the *configuration external to* T in C consist of the state of the shared memory and the states and private memories of all threads other than $thr(T)$ in C.

Given a set of transactions $\mathcal{T}$ and an execution α, the *execution interval* of $\mathcal{T}$ in α, written $interval(\alpha, \mathcal{T})$, is the smallest subsequence of α containing all the base steps for the transactions in $\mathcal{T}$.

Correctness. Our correctness criterion resembles the *opacity* condition of Guerraoui and Kapalka [43]. For a history H, and a partial order P on the transactions that appear in H, we say that H satisfies P-opacity if there exists a sequential history S such that:

- S is equivalent to *Complete(H)*.
- Every transaction $T_i \in S$ is legal in S.
- If $(T_i, T_j) \in P$, then $T_i \prec_S T_j$.

Given a function Γ that maps histories to partial orders of transactions that appear in those histories, we say a TM satisfies Γ-opacity if every history H generated by the TM satisfies $\Gamma(H)$-opacity.

When $\Gamma(H)$ is the real-time order on all the ordered pairs of non-concurrent transactions in H, the history S should preserve the real-time order of H as in the original definition of opacity. On the other hand, when $\Gamma(H)$ is empty, the correctness criterion is a serializability with consideration of aborted transactions. The use of Γ makes it possible to require a transactional ordering that lies between serializability and strict serializability according to any arbitrary rule (e.g., Riegel et al. [70] considered demanding real-time order only from transactions belonging to the same thread). We define a more general criterion in order to broaden the scope of our results. In the rest of this thesis, we will assume that $\Gamma(H)$ is a subset of the real-time order on transactions, unless stated otherwise.

DAP. We define the notion of *weak disjoint-access parallelism*, following [10]. Let T_1, T_2 be transactions, and let α be an execution. Let $\mathcal{T}$ be the set of all transactions whose execution interval overlaps with the execution interval of $\{T_1, T_2\}$ in α. Let X be the set of transactional objects accessed by $\mathcal{T}$. Let $G(T_1, T_2, \alpha)$ be an undirected graph with vertex set X, and an edge between vertices $x_1, x_2 \in X$ whenever there is a transaction $T \in \mathcal{T}$ accessing both x_1 and x_2. We say T_1, T_2 are *disjoint-access* in α if there is no path between T_1 and T_2 in $G(T_1, T_2, \alpha)$. Given two sets of base steps, we say they *contend* if there is a base object that is accessed by both sets of steps, and at least one of the accesses changes the state of the object.

Definition 1. An STM is *weakly disjoint-access parallel (weakly DAP)* if, given any execution α, and transactions T_1, T_2 that are disjoint-access in α, the base steps for T_1 and T_2 in α do not contend.

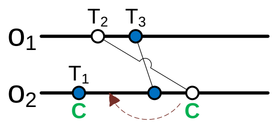


Figure 3.1: Example transactional run, transaction T_2 reads version o_2^1.

Depicting transactional runs. We depict transactional histories in the style of [70] (see Figure 3.1). An object o_i's state in time is represented as a horizontal line, with time proceeding left to right. Transactions are drawn as polylines, with circles representing accesses to objects. Filled circles indicate writes, and empty circles indicate reads. A commit is indicated by the letter **C**, and an abort by the letter **A**. A read operation returning an old value of an object is indicated by a dotted arc line. The initial value of object o_i is denoted by o_i^0, and the value written to o_i by the j'th write is denoted by o_i^j.

Chapter 4

On Avoiding Spurious Aborts in TM

This chapter deals with a theory for understanding aborts in transactional memory systems. Existing TMs may abort many transactions that could, in fact, commit without violating correctness. We call such unnecessary aborts *spurious aborts*. We classify what kinds of spurious aborts can be eliminated, and which cannot. We further study what kinds of spurious aborts can be avoided efficiently. Specifically, we show that some unnecessary aborts cannot be avoided, and that there is an inherent tradeoff between the overhead of a TM and the extent to which it reduces the number of spurious aborts. We also present an efficient example TM algorithm that avoids certain kinds of spurious aborts, and analyze its properties and performance.

A preliminary version of the work presented in this chapter appears in proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2009).

4.1 Introduction

A transaction's abort may be initiated by a programmer or may be the result of a TM decision. In the latter case, we say that the transaction is *forcefully aborted* by the TM. For example, when one transaction reads some object A and then writes to some object B, while another transaction reads the old value of B and then attempts to write A, one of the transactions must be aborted in order to ensure atomicity. Most existing TMs perform unnecessary (*spurious*) aborts, i.e., aborts of transactions that could have committed without violating correctness; see Section 2.1. Spurious aborts have several drawbacks: work done by the aborted transaction is lost, computer resources are wasted, and the overall throughput decreases. Moreover, after the aborted transactions restart, they may conflict again, leading to livelock and degrading performance even further.

This chapter aims to advance the theoretical understanding of TM aborts, by studying what kinds of spurious aborts can or cannot be eliminated, and what kinds of spurious aborts can or cannot be avoided efficiently. Specifically, we show that some unnecessary aborts cannot be avoided, and that there is an inherent tradeoff between the overhead of a TM and the extent to which it refrains from spurious aborts.

Previous works introduced two related notions: commit-abort ratio [40] and permissiveness [41]. The latter stipulates that if it is possible to proceed without aborts and still not violate correctness, no aborts should happen. However, while shedding insight on the inherent limitations of online TMs, these notions do not provide an interesting yardstick for comparing TMs. This is because under these measures, all online TMs inherently perform poorly for some worst-case workloads, as we show in Section 4.2.

In Section 4.3, we then define measures of spurious aborts that are appropriate for online TMs. Intuitively, our *strict online permissiveness* property allows a TM to abort some transaction only if not aborting any transaction would violate correctness. Unlike earlier notions, strict online permissiveness does not prevent the TM from taking an action that might lead to an abort in the future. Thus, the information available to the TM at every given moment suffices to implement strict online permissiveness. Clearly, this property depends on the correctness criterion the TM needs to satisfy. In this thesis, we consider opacity [43] or slight variants thereof (see Chapter 3). In this context, strict online permissiveness prohibits aborting a transaction whenever the execution history is equivalent to some sequential one. We prove that strict online permissiveness cannot be satisfied efficiently by showing a reduction from the NP-hard *view serializability* [62] problem. We then define a more relaxed property, *online permissiveness*, which allows the TM to abort transactions if otherwise it would have to change the serialization order between already committed transactions. We show that online permissiveness also has inherent costs — it cannot be satisfied by a TM using *invisible* reads. Moreover, the information about a read should be exposed in shared memory immediately after the read operation returns.

In Section 4.4, we show a polynomial time TM protocol satisfying online permissiveness. The protocol maintains a precedence graph of transactions and keeps it acyclic. Unfortunately, we show that the graph must contain some committed transactions. But without removing any committed transactions, detecting cycles in the precedence graph would be impractical as it would induce a high runtime complexity. Hence, we define precise garbage collection rules for removing transactions from the graph. Even so, a naïve traversal of the graph would be costly; we further introduce optimization techniques that decrease the number of nodes traversed during the acyclity check.

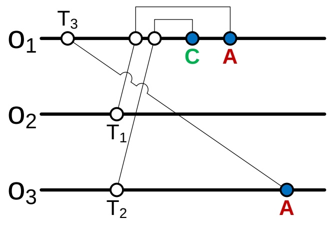
4.2 Limitations of Previous Measures

4.2.1 Commit-Abort Ratio

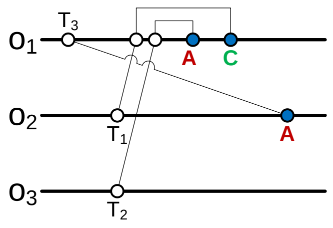
The *commit-abort ratio*, τ, [40] is the ratio between the number of committed transactions and the overall number of transactions in the history. Unfortunately, we now show that no online TM may guarantee an optimal commit-abort ratio.

As we said earlier, a TM does not know read and write accesses in advance, i.e., a TM is *online*. As opposed to this, we say that an *offline* algorithm knows the sequence of accesses of the transaction beforehand.

Lemma 1. *No TM can achieve the commit-abort ratio of an optimal offline algorithm.*



(a) Run r_1: T_2 commits, all other transactions abort: $\tau = \frac{1}{3}$



(b) Run r_2: T_1 commits, all other transactions abort: $\tau = \frac{1}{3}$

Figure 4.1: No online TM may know whether to abort T_1 or T_2 in order to obtain an optimal commit-abort ratio.

Proof. Consider the scenarios depicted in Figure 4.1. We show that no TM can achieve commit-abort ratio better than $\frac{1}{3}$ in both runs, while an optimal offline algorithm achieves $\tau = \frac{2}{3}$. Transactions T_1 and T_2 cannot both commit because they both write to o_1 after reading its previous value.

There are three possible scenarios for a TM algorithm: 1) abort T_1, 2) abort T_2, or 3) abort both T_1 and T_2. Clearly, in the third case τ cannot be better than $\frac{1}{3}$.

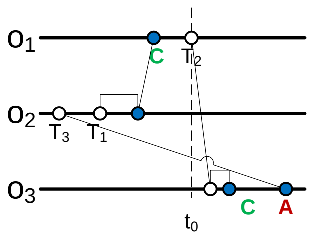
In run r_1 (Figure 4.1(a)), the TM commits T_2 and T_1 is aborted. Then the adversary causes transaction T_3 to read o_3 — it must be aborted because it conflicts with T_2, resulting in $\tau = \frac{1}{3}$.

In run r_2 (Figure 4.1(b)), the TM commits T_1 and T_2 is aborted. In this case the adversary causes T_3 to read o_2, T_3 must be aborted because of its conflict with T_1, resulting again in $\tau = \frac{1}{3}$.

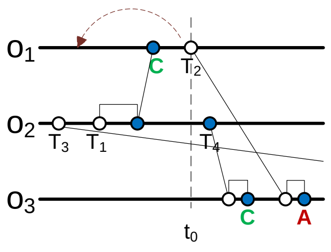
Note that the optimal offline TM in these cases would abort only one transaction, yielding $\tau = \frac{2}{3}$. □

4.2.2 Permissiveness

Since requiring an optimal commit-abort ratio is too restrictive, we consider a weaker notion that limits aborts only in runs where none are necessary. Recall that a TM accepts a certain history if it commits all of its transactions. A TM provides π-*permissiveness* [41] if it accepts every history satisfying π (a TM provides Γ-opacity-permissiveness if it accepts every history satisfying Γ-opacity). Gramoli et al. showed that existing TM implementations do not accept all inputs they could have, and hence are not permissive. We show that this is an inherent limitation.



(a) Run r_1: T_2 reads the value v_1



(b) Run r_2: T_2 reads the value v_0

Figure 4.2: At time t_0, no online TM knows which value should be returned to T_2 when reading o_1 in order to allow for commit in the future.

The formal impossibility illustrated in Figure 4.2 is captured in the following lemma:

Lemma 2. *For any Γ, there is no online TM implementation providing Γ-opacity-permissiveness.*

Proof. Consider the scenario depicted in Figure 4.2. All the objects have initial values, v_0. All the transactions start at the same time, and are therefore not ordered according to the real-time order, thus the third condition of our correctness criterion holds for any Γ.

T_1 writes values v_1 to o_1 and v_2 to o_2. At time t_0, there is a read operation of T_2 and the TM should decide what value should be returned. In general, the TM has four possibilities: (1) return v_1, (2) return v_0, (3) return some value v' different from v_0 and v_1, and (4) abort T_2. If the TM chooses to abort, then opacity-permissiveness is violated and we are done. (3) is not possible, for returning such a value would produce a history, for which any equivalent sequential history S would violate the sequential specification of o_1 and thus would not be legal.

Consider case (1): the TM returns v_1 for T_2 at time t_0. This serializes T_2 after T_1. Consider run r_1 depicted in Figure 4.2(a), where T_3 tries to write to o_3 and commit. In this run, the TM has to forcefully abort T_3, because not doing so would produce a history H with no equivalent sequential history: $T_1 \prec T_2 \prec T_3 \prec T_1$. However, if T_2 would read v_0 in run r_1, then T_2, T_1 and T_3 would be legal, and no transaction would have to be forcefully aborted. So Γ-opacity-permissiveness is violated.

In case (2), the TM returns v_0 for transaction T_2 at time t_0, serializing T_2 before T_1. Consider run r_2 depicted in Figure 4.2(b). Transaction T_4 writes to o_2, and afterwards reads and writes to o_3. Transaction T_4 has to be serialized after T_1, because T_1 has read v_0 from o_2. When T_2 tries to read and write to o_3 and commit, T_2 has to be serialized after T_4 because they both read and write to o_3. Therefore, the TM will have to forcefully abort some transaction, because not doing so would produce a history with no equivalent sequential history: $T_2 \prec T_1 \prec T_4 \prec T_2$. But if T_2 would read v_1 in run r_2, then no transaction would have to be forcefully aborted. So again, Γ-opacity-permissiveness is violated.

Runs r_1 and r_2 are indistinguishable to the TM at time t_0. Therefore, no online TM can accept both of the patterns, while an offline optimal TM can accept both of them. $\square$

4.3 Online permissiveness: limitations and costs

4.3.1 Strict Online Opacity-Permissiveness

Definition 2. Consider a history H, in which a transaction T receives an abort response A to one of its operations *op*. We say that H' is a live-T modification of H if H' is the same as H except that T receives a non-abort response to *op* in H'.

We now define a property that prohibits unnecessary aborts, and yet is possible to implement.

Definition 3. A responsive TM satisfies strict online Γ-opacity-permissiveness for a given Γ if the TM forcefully aborts a transaction T in a history H only if there exists no live-T modification of H that satisfies $\Gamma(H)$-opacity.

Note that this property does not define which transaction should be aborted if abort happens, and does not prohibit returning a value that will cause aborts in the future. For example, in the scenarios depicted in Figure 4.2, at time t_0, a TM satisfying this property may return either value, even though this might cause an abort in the future.

An algorithm satisfying strict online opacity-permissiveness should be able to detect whether returning a given value creates a history satisfying Γ-opacity. We show that this cannot be detected efficiently. To this end, we recall a well-known result about checking the serializability of the given history, which was proven by Papadimitriou [62].

Given history H, the *augmented* history $\bar{H}$ is the history that is identical to H, except two additional transactions: T_{init} that initializes all variables without reading any, and T_{read} that is the last transaction of $\bar{H}$, reading all variables without changing them. The set of *live* transactions in H is defined recursively in the following way: (1) T_{read} is live in H, (2) If for some live transaction T_j, T_j reads a variable from T_i, then T_i is also live in H. Note that aborted transaction cannot be live since no transaction may read a value written by an aborted one. A transaction is *dead* if it is not live. Two histories H and H' are *view equivalent* if and only if (1) they have the same sets of live transactions and (2) T_i reads from T_j in H if and only if T_i reads from T_j in H'. Note that a definition of view equivalence differs from a history equivalence defined in this thesis, which demands the same order of operations for each transaction in equivalent histories. History H is *view serializable*, if for every prefix H' of H, *complete(H')* is view equivalent to some serial history S. The following is proven in [62]:

Theorem 3 (Papadimitriou). *Testing whether the history H is view-serializable is NP-complete in the size of the history, even if H has no dead transactions.*

Lemma 4. *For any Γ, detecting whether the history H satisfies Γ-opacity is NP-complete in the size of the history.*

Proof. We first note that the problem of detecting view serializability has a trivial reduction to the problem of identifying whether a given history H is view equivalent to some serial history

S. Hence, in order to prove the claim, we need to show a reduction from the problem of detecting whether a history H is view-equivalent to some serial history S to the problem of detecting whether some history H' satisfies Γ-opacity. Consider history H with no dead transactions. Given the assumption of unique write values and in the absence of aborted transactions, the definition of view equivalence differs from the definition of opacity only in the fact that opacity refers to the partial order Γ, which is a subset of a real-time order. We construct history H', which is identical to history H except the following addition: for each T_i in H, we add *start(T_i)* at the beginning of H'. We will show that H is view equivalent to some serial history S if and only if H' satisfies Γ-opacity.

All the transactions in H' are concurrent (*start(T_i)* follows before any other operation for every T_i), therefore the third condition of Γ-opacity vacuously holds for any Γ. In the absence of aborts in H', H' satisfies Γ-opacity if and only if there exists a legal sequential history S', so that every transaction in H' issues the same invocation events and receives the same response events as in S'. Therefore, H' satisfies Γ-opacity if and only if H' is view-equivalent to some serial history S'. □

4.3.2 Online Opacity-Permissiveness

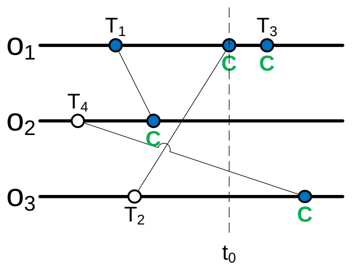


Figure 4.3: The order of transactions T_1 and T_2 is changed after their commit time.

Intuitively, the problem with strict online opacity-permissiveness lies in the fact that the order of committed transactions may be undefined and may change in the future. Consider, for example, the scenario depicted in Figure 4.3. Transactions T_1 and T_2 are not ordered according to real-time order, therefore they are not ordered by Γ. At time t_0, the serialization order is $T_1 \rightarrow T_2$, as o_1 holds the value written by T_2. When T_3 commits, the serialization order of T_1 and T_2 becomes undefined, since T_3 overwrites o_1 before any transaction reads the value written by T_2. And when T_4 commits, the serialization order becomes $T_2 \rightarrow T_4 \rightarrow T_1 \rightarrow T_3$. If the partial serialization order induced by the run cannot change after being defined, the problem becomes much easier. To capture this restriction, we extend the TM's interface so as to make the serialization decisions explicit: every commit operation returns a partial order on all committed transactions with conflicting writes. Specifically, we assume that a successful tryCommit(T_i) operation returns, instead of C, a partial order R_i on previously committed transactions.

We denote by $R(t)$ the value returned in the last commit occurring by point t in H; $R(t)$ is empty if no commit occurs by time t in H.

Note that this interface is only intended to expose the internal state of the TM, in order to facilitate reasoning, and can be filtered out before actually providing a response to the application. Using this interface, we now define the persistent ordering property, which prevents a TM from "changing its mind" about the serialization order of already committed conflicting transactions.

Definition 4 (Persistent Ordering). A history H (with the modified interface) satisfies persistent ordering if: 1) $R(t)$ orders all pairs of transactions T_i and T_j that have committed by point t and their write-sets intersect. 2) For all t' and t such that $t' < t$, $R(t') \subseteq R(t)$. 3) H satisfies $R(t)$-opacity for all t.

In other words, if committed transactions T_i and T_j both write to the same object in H, then they are explicitly ordered by the time both of them commit and their order persists thereafter. We say that a TM satisfies Persistent Ordering if every history generated by the TM satisfies Persistent Ordering. We now define our more relaxed property, online Γ-opacity-permissiveness, which may be satisfied at a polynomial cost.

Definition 5. A responsive TM satisfies online Γ-opacity-permissiveness for a given Γ if:

1. The TM satisfies Persistent Ordering.

2. The TM forcefully aborts a transaction T in a history H only if there exists no live-T modification of H that satisfies persistent ordering and $\Gamma(H)$-opacity.

Note that Definition 5 implies that each committing transactions should define its serialization order with regard to all other committed transactions that have written to the same objects. To the best of our knowing, all existing TMs do in fact define the order on two transactions that write to the object by the time the latter transaction commits. We note that this requirement might be limiting for TMs that wish to exploit the benefits of commutative or write-only operations (see [60]), and do not necessarily define the serialization point of the committed transactions. However, this limitation is essential for an effective check of the opacity criterion.

In the following sections we show a polynomial-time TM satisfying online opacity-permissiveness. We now prove that such an implementation, nevertheless, has some inherent costs.

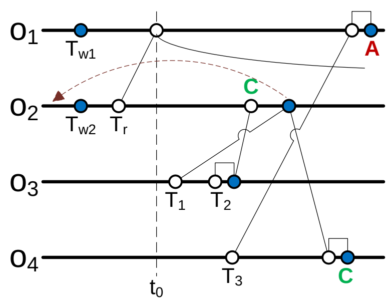
Impossibility of invisible reads. One of the basic decisions that needs to be made during the design of a TM is whether to *expose* the fact that transaction T_i has read the object o, i.e., make a change in shared memory as a result of the read, making the read *visible*. In case we expose the read, there arises another question, regarding whether we can postpone exposing the read until the commit. One of the central problems with exposing the read is that it requires writing metadata in shared memory. One typically tries to avoid writes to shared memory, because writing data that is read by different cores has a high cache penalty. Postponing exposing the read until the commit may save redundant writes in case the transaction eventually aborts.

Unfortunately, we shall now show that if the serialization order can violate the real-time order of transactions, then online opacity permissiveness requires all reads to be exposed in shared memory immediately after a read happens. To this end, we first need to rule out trivial TMs, for example, ones that always return an object's initial value in response to a read. We formally define our non-triviality requirement as follows:

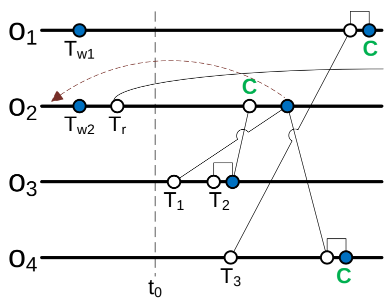
Definition 6. For a given Γ, a TM is not trivial if a read operation of object o by transaction T_i does not return an older value than the last one written to o by a committed transaction before T_i began, unless returning the last value written before T_i began generates a history H that is not $\Gamma(H)$-opaque.

In other words, read may return an old value only if there is a good reason to do so (avoiding an abort). We now show that every non-trivial TM satisfying online opacity-permissiveness with no respect to real-time order must expose all its read operations immediately as they happen:

Lemma 5. *Let $\Gamma_\emptyset$ be a function from histories to partial transactional orders such that $\Gamma_\emptyset(H) = \emptyset$. If a non-trivial responsive TM satisfies online $\Gamma_\emptyset$-opacity-permissiveness, then any active transaction T_i that has read $n \geq 2$ distinct objects must keep all its reads visible.*



(a) Run r_1: T_3 cannot commit



(b) Run r_2: T_3 commits

Figure 4.4: T_3 does not distinguish between r_1 and r_2 at time t_0 if T_r does not expose its read.

Proof. Assume by contradiction that there exists a non-trivial TM satisfying online Γ-opacity-permissiveness and there exists an active transaction T_r that reads non-initial values of objects o_1 and o_2 (and perhaps some additional objects) until time t_0 and does not expose the read of some object o_1, as depicted in the left part of Figure 4.4(a).

We now continue the run from t_0 onward as described below. We invoke transaction T_1 reading object o_3, and then transaction T_2 that reads o_3, writes to o_3 and reads o_2. By non-triviality, T_1 and T_2 read the same version of o_3, hence once T_2 writes to o_3, T_2 is serialized after T_1. Moreover, T_2 must read the version of o_2 written by T_{w2} — the same one as read by T_r. We next invoke transaction T_3, which reads o_4. We then continue transaction T_1 so that it writes to o_2, then reads o_4, writes to o_4 and commits. As mentioned earlier, T_1 is serialized before T_2 and T_2 reads the object version written by T_{w2}, therefore T_1 must be serialized before T_{w2} (and before T_r). Note that T_1 can be serialized before T_{w2} because $\Gamma_\emptyset$ does not impose a real-time order on transactions. By non-triviality, T_1 and T_3 read the same version of o_4, hence T_3 is serialized before T_1 (and before T_r).

Finally, we continue transaction T_3 so that it reads o_1, writes to o_1 and tries to commit. By non-triviality, T_3 reads the version of o_1 written by T_{w1}. The commit operation of T_3 cannot succeed: on the one hand, T_3 must be serialized after T_{w1}, and on the other hand T_3 must be serialized before T_r, but T_3 cannot be serialized between T_{w1} and T_r because T_r reads the version of o_1 written by T_{w1}. Hence T_3 aborts in r_1.

Consider run r_2 depicted in Figure 4.4(b). This run is identical to r_1 except that T_r's read of o_1 is removed. T_3 can commit successfully in r_2, with the following serialization order: $\{T_{w1}, T_3, T_1, T_{w2}, T_r, T_2\}$. Since we assume the TM satisfies online Γ-opacity-permissiveness, T_3 commits. However, since

T_r does not expose its read of o_1, T_3 cannot distinguish between r_1 and r_2, a contradiction. $\square$

4.4 The AbortsAvoider Algorithm

We now present AbortsAvoider, a TM algorithm implementing online opacity-permissiveness for any given Γ. The basic idea behind AbortsAvoider is to maintain a precedence graph of transactions, and keep it acyclic, as explained in Section 4.4.1. A simplified version of the protocol based on this graph is then presented in Section 4.4.2. A key challenge AbortsAvoider faces is that completed transactions cannot always be removed from the graph, whereas keeping all transactions forever is clearly impractical. We address this challenge in Section 4.4.3, presenting a garbage collection mechanism for removing terminated transactions from the graph. In Section 4.4.4 we present another optimization, which shortens paths in the graph to reduce the number of terminated transactions traversed during the acyclity check. Our complexity analysis appears in the same section.

4.4.1 Basic Concept: Precedence Graph

Information bookkeeping. Our protocol maintains *object version lists*. We now explain what such a TM does: (1) each object o is associated with a totally ordered set of versions, (2) a read of o returns the value of one of o's versions, and (3) a write to o adds a new version of o upon commit. For simplicity, at any given moment, we number the versions of the object in increasing order. (Note that the numbering is for analysis purposes only, and the numbers of the versions change during the run as the versions are inserted and removed from the versions list). The object version $o.v_n$ includes the data, $o.v_n$.*data*, the writer transaction, $o.v_n$.*writer*, and a set of readers, $o.v_n$.*readers*. Each transaction has a *readList* and a *writeList*. An entry in a *readList* points to the version that has been read by the transaction. A *writeList* entry points to the object that should be updated after commit, the new data, and the place to insert the new version, (which may be undefined till the commit). For the sake of simplicity we assume that the values written to transactional objects are unique.

Precedence graph. Transactions may point to one another, forming a directed labelled precedence graph, PG. PG reflects the dependencies among transactions as created during the run. We denote a precedence graph of history H as PG_H. The vertices of PG are transactions, the edges of PG are as follows (Figure 4.5):

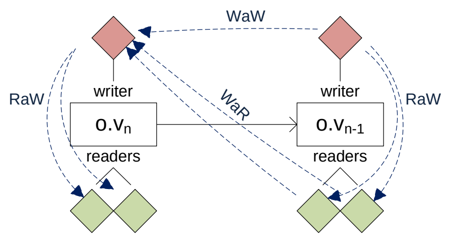


Figure 4.5: Object versions and the precedence graph, PG.

If $(T_j, T_i) \in \Gamma(H)$, then PG contains (T_j, T_i) labelled L_Γ (Γ order). If T_i reads $o.v_n$ and T_j writes $o.v_n$, then PG contains (T_j, T_i) labelled L_{RaW} (Read after Write). If transaction T_i writes $o.v_n$ and T_j writes $o.v_{n-1}$, then PG contains (T_j, T_i) (Write after Write) labelled L_{WaW}. If transaction T_i writes $o.v_n$ and T_j reads $o.v_{n-1}$, then PG contains (T_j, T_i) labelled L_{WaR} (Write after Read).

Below we present lemmas that link maintaining acyclity in PG and satisfying online-permissiveness. To this end, we restrict our discussion to non-local histories, which we now define. We say that a read operation of T_i *read$_i$(o)* in H is *local* if it is preceded in $H|T_i$ by a write operation *write$_i$(o,v)*. A write operation *write$_i$(o,v)* is *local* if it is followed in $H|T_i$ by another write operation *write$_i$(o,v')*. The *non-local* history of H is the longest subsequence of H not containing local operations [43]. Note that the precedence graph does not refer to local operations.

We denote $PG(t)$ to be the graph at time t. We define λ_{PG} to be the following binary relation: if PG contains a path from T_i to T_j consisting of L_{WaW} edges, then $T_i \prec_{\lambda_{PG}} T_j$. Note that if PG is acyclic, then λ_{PG} is reflexive, antisymmetric and transitive, and therefore λ_{PG} is a partial order.

Lemma 6. *Consider a TM maintaining object version lists. If PG is acyclic throughout some run, then the non-local history H of the run satisfies $\Gamma \cup \lambda_{PG}$-opacity.*

Proof. Let H be a history over transactions $\{T_1 \ldots T_n\}$. Let $H_C = \mathit{Complete}(H)$, i.e. H with A_i added for every active $T_i \in H$.

Since PG is acyclic, it can be topologically sorted. Let $T_{i1}, \ldots, T_{in}$ be a topological sort of PG, and let S be the sequential history $T_{i1}, \ldots, T_{in}$. Clearly, S is equivalent to H_C because both of the histories contain the same transactions and each transaction issues the same operations and receives the same responses in both of them.

We now prove that every $T_i \in S$ is legal. Assume by contradiction that there are non-legal transactions in S. Let T_i be the first such transaction. If T_i is non-legal, T_i reads a value of object o that is not the latest value written to o in S by a committed transaction. (Recall that by definition of object version lists, only values written by committed transactions can be read.) S contains only non-local operations, and therefore T_i reads the version $o.v_n$ written by another transaction T_j. Therefore, there is an edge from T_j to T_i in PG. It follows that T_j is committed in S and ordered before T_i according to the topological sort. If the value of $o.v_n$ is not the latest value written in S before T_i, then there exists another committed transaction T_j' that writes to o and is ordered between T_j and T_i in S. If T_j' writes to a version earlier than $o.v_n$, then there is a path from T_j' to T_j in PG, and therefore T_j' is ordered before T_j in S. If T_j' writes to a version later than $o.v_n$, then there is a path from T_i to T_j' in PG, and therefore T_j' is ordered after T_i in S. In any case, T_j' cannot be ordered between T_j and T_i in S, a contradiction.

For each pair $T_i \prec_\Gamma T_j$, PG contains an edge from T_i to T_j. Therefore, according to the topological sort, S preserves the partial order Γ. By definition S also preserves the order defined by λ_{PG}.

Summing up, *Complete*(H) is equivalent to a legal sequential history S, and S preserves partial order $\Gamma \cup \lambda_{PG}$. Therefore H is $\Gamma \cup \lambda_{PG}$-opaque. □

Lemma 7. *Every TM that maintains object version lists and keeps PG acyclic satisfies persistent ordering.*

Proof. In order to prove that a TM satisfies persistent ordering we need to show the following: 1) define a partial order R_i returned by a successfully committed transaction (in other words, define the way a TM exports an ordering interface); 2) show that R_i orders all pairs of committed transactions with a non-empty intersection of their write-sets; 3) show that $R(t)$ monotonically increases with t and 4) prove that $H|_t$ satisfies $R(t)$-opacity at any t.

1) We define R_i returned by a successfully committed transaction at time t to be $\lambda_{PG(t)}$; in other words R_i orders T_i and T_j if they are connected in PG by L_{WaW} edges.

2) Consider two committed transactions T_k and T_m that have a common object o in their write-sets such that T_k has written to the version $o.v_i$ and T_m has written to the version $o.v_j$, where $i < j$. In this case PG contains a path from T_k to T_m consisting of L_{WaW} edges and therefore λ_{PG} contains a pair (T_k, T_m). Hence, R_i orders all pairs of committed transactions with a non-empty intersection of their write-sets.

3) According to the rules for updating PG, L_{WaW} edges are never removed and $R(t') \subseteq R(t)$ for every $t' < t$.

4) According to Lemma 6, $H|_t$ satisfies $\lambda_{PG(t)}$-opacity and therefore $H|_t$ satisfies $R(t)$-opacity. □

Lemma 8. *Consider a responsive TM maintaining object version lists and keeping PG acyclic. Consider that this TM forcefully aborts a transaction T in a history H only if there exists no live-T H's modification H', such that PG'_H contains no cycles. Then this TM satisfies online Γ-opacity-permissiveness.*

Proof. As shown in Lemma 7, the TM satisfies persistent ordering. We need to show that if there is a cycle in PG, then the run violates $(\Gamma \cup \lambda_{PG})$-opacity.

We show first that if there is an edge (T_i, T_j) in PG, then every legal sequential history S preserving $\Gamma \cup \lambda_{PG}$ and equivalent to *Complete(H)* orders T_i before T_j. Consider two transactions T_i and T_j such that there is an edge (T_i, T_j) in PG. If the edge is labeled L_Γ, then $(T_i, T_j) \in \Gamma$, and S orders T_i before T_j. If the edge is labeled L_{RaW}, then T_j reads a value written by T_i and S also orders T_i before T_j. If the edge is labeled L_{WaW}, then $T_i < T_j$ according to λ_{PG}, hence S also orders T_i before T_j. If the edge is labeled L_{WaR}, then T_i reads $o.v_n$ while T_j writes $o.v_{n+1}$. On the one hand, T_j should be ordered after $o.v_n$.*writer* in S (there is an edge from $o.v_n$.*writer* to T_j labeled L_{WaW}). On the other hand, T_j cannot be ordered between $o.v_n$.*writer* and T_i, because T_i must read the value written by $o.v_n$.*writer* in S. Therefore, T_j is ordered after T_i in S in this case as well.

Summing up, an edge (T_i, T_j) in the precedence graph induces the order of T_i before T_j in any legal sequential history S preserving $\Gamma \cup \lambda_{PG}$ and equivalent to *Complete(H)*. Therefore, if PG contains a cycle, no such sequential history exists, and the TM cannot satisfy $\Gamma \cup \lambda$-opacity. □

Corollary 9. *Consider a TM maintaining object version lists that keeps PG acyclic. Consider that this TM forcefully aborts a transaction T in a history H only if there exists no live-T H's modification H', such that PG'_H contains no cycles. Then this TM satisfies Γ-opacity and online Γ-opacity-permissiveness.*

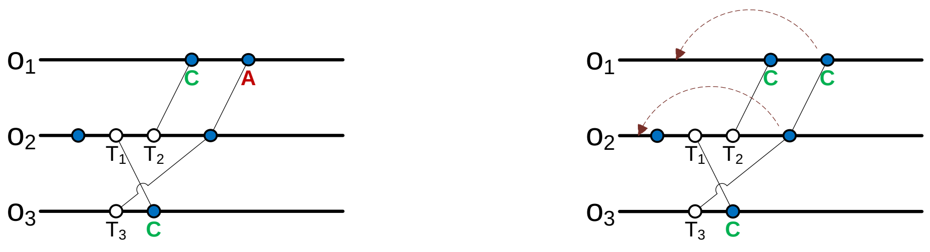
4.4.2 Simplified Γ-AbortsAvoider Algorithm

AbortsAvoider algorithm maintains object version lists as explained above, keeps PG acyclic and forcefully aborts a transaction only if not aborting any transaction would create a cycle in PG. Read and write operations are straightforward, they are depicted in Algorithm 1. A *read* operation (line 4) looks for the latest possible object version to read without creating a cycle in PG. *Write* operations (line 13) postpone the actual work till the commit.

Algorithm 1 Γ-AbortsAvoider for T_i - Read/Write.

```
 1: procedure START()
 2:     $prev = \{T_j : (T_j, T_i) \in \Gamma(H) \wedge \nexists T_k \in PG : (T_j, T_k) \in \Gamma(H)\}$
 3:     $\forall T_{prev} \in prev : PG.\text{ADDEDGES}(\{(T_{prev}, T_i)\})$
 4: procedure READ(Object $o$)
 5:     if $o \in T_i.writeList$ then return $T_i.writeList[o].data$
 6:     if $o \in T_i.readList$ then return $T_i.readList[o].data$
 7:     $n \leftarrow$ the latest version that can be read without creating a cycle in $PG$
 8:     if $n = \perp$ then return abort event $A_i$
 9:     $PG.\text{ADDEDGES}(\{(o.v_n.writer, T_i),(T_i, o.v_{n+1}.writer)\})$
10:     $o.v_n.readers.\text{ADD}(T_i)$
11:     $T_i.readList.\text{ADD}(\langle o.v_n \rangle)$
12:     return $o.v_n.data$
13: procedure WRITE(Object $o$, ObjectData $val$)
14:     if $o \in T_i.writeList$ then
15:         $T_i.writeList[o].data \leftarrow val$; return
16:     if $o \in T_i.readList$ then
17:         ▷ non-blind write, victim version is read version
18:         $writeNode \leftarrow \langle o, readList[o].version, val \rangle$
19:     else
20:         ▷ blind write, victim version is not known
21:         $writeNode \leftarrow \langle o, \perp, val \rangle$
22:     $T_i.writeList.\text{ADD}(writeNode)$
```

The *commit* operation is more complicated. Intuitively, for each object written during transaction, the algorithm should find a place in the object's version list to insert the new version without creating a cycle. Unfortunately, checking the objects one after another in a greedy way can lead to spurious aborts, as we illustrate in Figure 4.6(a). Committing T_3 first seeks for a place to install the new version of o_1 and decides to install it after the last one (serializing T_3 after T_2). When T_3 considers o_2, it discovers that the new version cannot be installed after the last one, because T_3 should precede T_1, but it also cannot be installed before the last one, because that would make T_3 precede T_2, so T_3 is aborted. However, installing the new version of o_1 before the last one would have allowed T_3 to commit, as depicted in Figure 4.6(b), that is why aborting T_3 violates online Γ-



(a) Run with greedy check (b) Run with no spurious aborts

Figure 4.6: Checking the written objects in a greedy way during the commit may lead to a spurious abort.

opacity-permissiveness.

Our commit operation (Algorithm 2, line 23) is divided to two phases. We call the object version after which the new version is to be installed a *victim version*. The victim version is known only for the non-blind writes (that is version, which has been read before the write, line 18). In the first phase the algorithm tries to install the non-blind writes (lines 27–33). In the second phase (lines 35–48) the algorithm tries to find the vicim versions for the blind writes in iterations. Initially, the victim is the object's latest version. In each iteration, the algorithm traverses the objects and for each one searches for the latest possible victim to install the new version without creating a cycle in PG (line 40). When victim $o.v_n$ is found, an edge from T_i to the writer of $o.v_{n+1}$ is added to PG (line 46). We add only the outgoing edges at this point, because changing the victim from $o.v_n$ to $o.v_{n-1}$ may remove some incoming edges to T_i but cannot remove outgoing ones. Meanwhile, incoming edges are kept in *inEdges*. After each iteration, there are possibly new outgoing edges added to PG, that would mean that the previously found victim versions might not suit anymore and a new iteration should be run. Once there is an iteration when no new edges are added, the algorithm commits — it installs the new versions after their victims and adds all the edges, including *inEdges* from the latest iteration, to the PG.

The following lemma immediately follows from the protocol.

Lemma 10. *Γ-AbortsAvoider maintains PG acyclic.*

Proof. The edges added to the graph are defined in functions READ (line 7) and VALIDATEWRITE (line 63). Both functions validate that adding the new edges preserves PG acyclity. □

We now want to show that the algorithm does not introduce unnecessary aborts.

Algorithm 2 Γ-AbortsAvoider for T_i - Commit.

```
procedure COMMIT
    newEdges ← ∅                                              ▷ edges added upon commit
    blinds ← ∅                                                ▷ the set of blind writes
    ▷ Phase I — install the non-blind writes
    for each n in $T_i$.writeList do
        if n.victim ≠ ⊥ then
            (v,edgs) ← VALIDATEWRITE(newEdges,n.victim)
            if v = FALSE then return abort event $A_i$
            newEdges ← newEdges ∪ edgs
        else
            blinds ← blinds ∪ {n}
    ▷ Phase II — install the blind writes
    repeat
        foundOutEdges ← FALSE
        inEdges ← ∅
        for each n in blinds do
            ▷ find the latest possible victim
            (victim,edges) ← FINDVICTIM(newEdges,n)
            if victim = ⊥ then return abort event $A_i$
            for each e in edges do
                if e is incoming to $T_i$ then
                    inEdges ← inEdges ∪ e
                else if e ∉ newEdges then
                    newEdges ← newEdges ∪ {e}
                    foundOutEdges ← TRUE
    until foundOutEdges = FALSE
    ▷ commit point
    for each n in $T_i$.writeList do
        install the new version right after n.victim
        $PG$.ADDEDGES(newEdges ∪ inEdges)

procedure FINDVICTIM(List⟨Edge⟩ newEdges, WriteNode wn) : (ObjectVersion, List⟨Edge⟩)
    ▷ find the latest possible victim
    if wn.victim = ⊥ then vctm ← wn.latestVersion
    else vctm ← wn.victim
    while vctm ≠ ⊥ do
        ▷ check installing the new version after vctm
        (valid, edges) ← VALIDATEWRITE(newEdges,vctm)
        if valid = TRUE then return (vctm,edges)
        vctm ← vctm.prev                                      ▷ go to the previous version
    return (⊥, ⊥)                                             ▷ no suitable victim found

procedure VALIDATEWRITE(List⟨Edge⟩ edges, ObjectVersion $o.v_n$) : (boolean, List⟨Edge⟩)
    added ← {($o.v_n$.writer, $T_i$), ($o.v_n$.readers, $T_i$), ($T_i$, $o.v_n$.next.writer)}
    valid ← acyclity of $PG$ after adding edges ∪ added
    return (valid, added)
```

Theorem 11. *Γ-AbortsAvoider forcefully aborts a transaction T in a history H only if there exists no live-T H's modification H', such that PG'_H contains no cycles.*

Proof. We first note that in Γ-AbortsAvoider no transaction can abort other transactions – the only transaction that can be aborted as a result of T_i's operation invocation is T_i by itself. Hence, if Γ-AbortsAvoider aborts a set S of transactions, then $|S| = 1$. Therefore, it is sufficient to prove the following: Γ-AbortsAvoider forcefully aborts a transaction only if not aborting any transaction would create a cycle in PG.

The read operation of object o (line 4) returns A_i only if there is no object version to read without introducing a cycle in PG. Write operation (line 13) does not abort any transaction — it postpones all the work till the commit.

Commit operation (line 23) tries to write the new versions of all the objects written during the transaction. If the object is written in the non-blind way, then the victim version is known beforehand and the new version has to be installed after the version that has been read (line 29). In this case the validation is done by *validateWrite* function (line 63), which fails if and only if adding the appropriate edges to PG creates a cycle.

It remains to show that commit function does not succeed to execute the blind writes only if that creates a cycle in PG. We will show now that if there exists a way to execute the blind writes without creating a cycle in PG, the algorithm will find it.

First of all, we will analyze the variable *newEdges* (line 24), which keeps the set of the edges added to PG upon successful commit. Edge $(T_i, T_j) \in$ *newEdges* is *compulsory*, if PG must have a path from T_i to T_j after successful commit (to that end, the edge represents a real, compulsory dependency).

Lemma 12. *During* COMMIT() *function of AbortsAvoider algorithm, newEdges set contains compulsory edges only.*

Proof. In the first phase of COMMIT(), AbortsAvoider proceeds the non-blind writes (lines 27–33). There is a single possible victim version for the non-blind write, and therefore the edges added to *newEdges* set during the first phase are compulsory.

Consider the second phase of COMMIT(), when AbortsAvoider proceeds the blind writes (lines 35–48). We will show by induction that all the edges added to *newEdges* in the second phase are compulsory.

Induction basis. At the beginning of the second phase *newEdges* set contains only the edges added by the non-blind writes, which are compulsory, as shown before.

Induction step. Let's assume, that all the edges added to *newEdges* by the algorithm so far are compulsory. Consider new edge $(T_i, o.v_{k+1}.\mathit{writer})$ added to *newEdges* by the algorithm in line 46. This happens if $o.v_k$ is chosen to be a victim version for writing to object o. According to the algorithm, $o.v_k$ is chosen to be the victim version only if all the versions $o.v_{k'}$ for $k' > k$ did not suit to be the victim versions for a given *newEdges* set. According to the induction assumption, *newEdges* set contains compulsory edges only, therefore all the versions $o.v_{k'}$ for $k' > k$ cannot be victim versions for the write operation. According to the algorithm, choosing any object version $o.v_{k'}$ for $k' \leq k$ (i.e., object version that is earlier than $o.v_k$) yields a path from T_i to $o.v_{k+1}.\mathit{writer}$ in PG, finishing the proof. □

For each object written in a blind way the algorithm checks the victim versions starting from the latest one. Victim version validation is executed in the following way: PG is checked for acyclity after inserting the edges from *newEdges* set together with the edges corresponding to adding the new version after $o.v_k$. As stated in Lemma 12, *newEdges* set contains compulsory edges only, therefore validation fail for $o.v_k$ means that neither $o.v_k$, nor any version later than $o.v_k$ can be the victim version of o. The algorithm traverses the objects in iterations, till it finds a combination of victim versions that does not create a cycle in PG (and then commits), or discovers object o such that none of o's versions can be the victim version (and then aborts). □

Corollary 13. Γ-*AbortsAvoider satisfies* Γ-*opacity and online* Γ-*opacity-permissiveness.*

We have shown that Γ-AbortsAvoider protocol is correct and avoids unnecessary aborts. In the rest of the chapter we will show the garbage collection rules and optimization techniques for the protocol.

4.4.3 Garbage Collection

A TM should garbage collect unused metadata. In our case, metadata consists of the objects' previous versions as well as terminated transactions. In this section, we describe how those may be garbage collected.

Read operations. Consider transaction T_i reading object o. The following lemma stipulates that some of the edges added to the precedence graph in the simplified protocol are redundant, and in fact, the only edges that need to be added by the protocol during read operations are incoming ones.

Lemma 14. *When T_i reads $o.v_n$, it suffices to add one edge from $o.v_n$.writer to T_i in PG.*

Proof. We say that adding an edge (v_1, v_2) is *unnecessary*, if PG already contains a path from v_1 to v_2, thus adding this edge does not influence on the cycle detection. We will show that adding the outgoing edge from T_i to $o.v_n$.*writer* during a read is unnecessary. Therefore the only edge that need to be added by the protocol is the edge from $o.v_{n-1}$.*writer* to T_i.

The protocol adds outgoing edge from T_i to $o.v_n$.*writer* if T_i reads version $o.v_{n-1}$. According to the algorithm, T_i tries first to read the latest version $o.v_{n+k}$, if this read creates a cycle, it tries to read $o.v_{n+k-1}$, $o.v_{n+k-2}$ and so on till it arrives to $o.v_{n-1}$. Note, that before starting the read, the graph PG was acyclic. If T_i does not succeed to read $o.v_{n+k}$, it means that adding an edge from $o.v_{n+k}$.*writer* to T_i would create a cycle, hence there is a path from T_i to $o.v_{n+k}$.*writer* before the start of the read. When T_i tries to read $o.v_{n+k-1}$ and does not succeed, it means that adding the edges $\{(o.v_{n+k-1}$.*writer*, $T_i)$, $(T_i$, $o.v_{n+k}$.*writer*$)\}$ creates a cycle in PG. As we have concluded, before the read, PG contained a path from T_i to $o.v_{n+k}$.*writer* and was acyclic, therefore adding the single edge $(o.v_{n+k-1}$.*writer*, $T_i)$ creates a cycle in PG, i.e. there was a path from T_i to $o.v_{n+k-1}$.*writer* before the read. Continuing in the same way, we conclude that before the read there was a path from T_i to $o.v_n$.*writer*. Therefore, adding an edge from T_i to $o.v_n$.*writer* is unnecessary. □

Using the optimization above, no incoming edge is ever added to a terminated transaction as a result of a read operation.

Write operations. We would like to know whether the new incoming edges may be added to a terminated transaction as a result of write operation. Consider committed transaction T_i that has written to o. If the new version $o.v_n$ has been written in a non-blind way (i.e. transaction T_i has read the version $o.v_{n-1}$ and then installed $o.v_n$), then no other transaction T_j will be able to install a new version between $o.v_{n-1}$ and $o.v_n$, for that would cause a cycle between T_i and T_j. Blind writes, however, are more problematic. Consider, for example, the scenario depicted in Figure 4.7. At time t_0, T_1 has no incoming edges, but we are still not allowed to garbage collect it as we now explain. There is a transaction T_2 that read object o_1 with an active preceding transaction T_3. At the time of T_3's commit, it discovers that it cannot install the last version of o_1, and tries to install the preceding version. Had we removed T_1 from PG, this would have caused a consistency violation, because we would miss the cycle between T_1 and T_3.

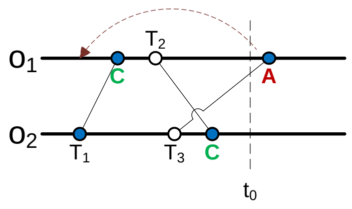


Figure 4.7: The blind write of transaction T_1 does not allow us to garbage collect it at time t_0.

The example above demonstrates the importance of knowing that from some point onward, T_i may have no new incoming edges. The lemma below shows that some edge additions can be saved:

Lemma 15. *If T_i is a terminated transaction, then no incoming edges need to be added to T_i in PG as long as for each $o.v_n$ written blindly by T_i there is no reader with an active preceding transaction.*

Proof. Consider a terminated transaction T_i satisfying conditions of the lemma. According to Lemma 14 no transaction may add incoming edges to T_i as a result of read operation. It remains to check the writes. According to the protocol, the incoming edge to T_i may be added only if transaction T_j installs the version prior to the version $o.v_n$ written by T_i. First of all we should notice that $o.v_n$ should be written in a blind way in order to make this scenario happen. Secondly, if T_j tries to insert a new version before $o.v_n$, it means that T_j failed to insert its version after $o.v_n$, i.e. adding the edges from T_i and from the readers of $o.v_n$ to T_j created a cycle. But we know that T_j cannot precede the readers of $o.v_n$ according to the condition of the lemma, that is why there was a path from T_j to T_i before the write operation of T_j. Therefore there is no need to add the edge from T_j to T_i when installing the new version. □

Garbage collection conditions. We say that a transaction is *stabilized* if no incoming edges may be added to it in the future. At the moment when T_i has no incoming edges and it is stabilized, we know that T_i will not participate in any cycle, and thus may be garbage collected.

Theorem 16. *The terminated transaction T_i is stabilized at time t_0 if either (1) T_i has not written blindly any object version $o.v_n$, or (2) all active transactions at time t_0 and all the transactions beginning after t_0 follow T_i according to Γ.*

Proof. According to Lemma 15, no incoming edges need to be added to terminated T_i in PG if T_i has no blind writes. If transaction T_j follows T_i according to Γ, then according to AbortsAvoider algorithm, PG will contain a path from T_i to T_j after `START()` operation of T_j . Therefore, T_j may not add incoming edge to T_i if $T_i \prec_\Gamma T_j$. Hence, if all active transactions at t_0 and all the transactions beginning after t_0 follow T_i according to Γ, then no new incoming edges will be added to T_i. □

For this, we deduce that terminated transactions with no incoming edges satisfying one of the conditions of Theorem 16 may be garbage collected. Note that in the runs with no blind writes, every terminated transaction is stabilized and thus the transaction may be garbage collected at the moment it has no incoming edges.

4.4.4 Path Shortening and Runtime Analysis

AbortsAvoider protocol allows adding new edges to PG only if they do not introduce cycles in PG. The straightforward cycle detection algorithm runs DFS starting from T_i, traversing a set of nodes we refer to as *ingress$_i$*. We now present an optimization that reduces the number of nodes in *ingress$_i$*.

Consider stabilized terminated T_j. The idea is to connect the ingress nodes to the egress nodes of T_j directly, thus preventing DFS from traversing T_j. This becomes possible because T_j is stabilized and thus may not have new ingress nodes, hence the egress nodes do not miss the precedence info when they lose their edges from T_j. Once a terminated transaction T_j satisfies the conditions of Lemma 15 and it can no longer have additional incoming edges, (e.g., any transaction with no blind writes), we remove all of its outgoing edges by connecting its ingress nodes directly to its egress nodes as described above, and indicate that T_j is a *sink*, i.e., cannot have outgoing edges in the future. Once a transaction is marked as a sink, any outgoing edge that should be added from it is instead added from its ingress nodes. Note that our path shortening only bypasses stabilized nodes. Had we bypassed also non-stabilized ones, we would have had to later deal with adding new incoming nodes to their egress nodes, which could require a quadratic number of operations in the number of terminated transactions. Hence, we chose not to do that.

Runtime complexity of the operations. Running DFS on *ingress*$_i$ takes $O(V^2)$, where V is the number of transactions preceding T_i, whose nodes have not been garbage collected. In the general case, $V = \textit{\#terminated} + \textit{\#active}$. But if all the transactions preceding Dsc_i had no blind writes, $V = \textit{\#active}$.

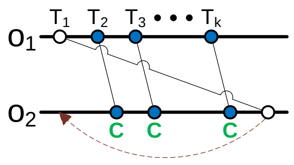


Figure 4.8: All object versions must be kept, as their writers have an active preceding transaction T_2.

The read operation seeks the proper version to read in the version list. Unfortunately, the number of versions that need to be kept is limited only by the number of terminated transactions. Consider the scenario depicted in Figure 4.8. Here, the only version of o_2 that may be read by T_1 is the first, all other versions are written by transactions that T_1 precedes. In order to find a latest suitable version, the read operation may use a binary search – $O(\log(\textit{\#terminated}))$ versions should be checked. Adding the edges takes $O(\textit{\#active})$. So altogether, the read complexity is $O(\log(\textit{\#terminated}) \cdot \max\{\textit{\#active}^2, \textit{\#terminated}^2\})$, and $O(\log(\textit{\#terminated}) \cdot \textit{\#active}^2)$ when there are no blind writes.

The write operation postpones all the work till the commit. The number of iterations in the commit phase is $O(\textit{\#writes} \cdot \textit{\#terminated})$, and in each iteration $O(\textit{\#writes})$ validate operations should be run. So the overall write cost is $O(\textit{\#writes}^2 \cdot \textit{\#terminated} \cdot \max\{\textit{\#active}^2, \textit{\#terminated}^2\})$, and $O(\textit{\#active}^2)$ when there are no blind writes.

Finally, we would like to emphasize that although in the worst-case, these costs may seem high, transactions without blind writes are garbage collected immediately upon commit. Moreover, the only nodes in *ingress*$_i$ where cycles are checked are transactions that conflict with T_i. Typically, in practice, the number of such conflicts is low, suggesting that our algorithm's common-case complexity is expected to be good. On the other hand, if the number of conflicts is high, then most TMs existing today would abort one of the transactions in each of these cases, which is not necessarily a better alternative.

Chapter 5

On Maintaining Multiple Versions in STM – Theoretical Properties

An effective way to reduce the number of aborts in software transactional memory is to keep multiple versions of transactional objects. In this chapter, we study inherent properties of STMs that use multiple versions to guarantee successful commits of all read-only transactions. We first show that these STMs cannot be disjoint-access parallel. We then consider the problem of garbage collecting old object versions. We show that the memory consumption of algorithms keeping a constant number of versions per object might grow exponentially with the number of objects, and prove that no STM can be optimal in the number of previous versions kept. Moreover, we show that garbage collecting useless versions is impossible in STMs that implement invisible reads. Finally, we present an STM algorithm using visible reads that efficiently garbage collects useless object versions.

A preliminary version of the work presented in this chapter appears in proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC 2010).

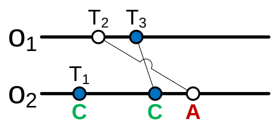
5.1 Introduction

As mentioned in Chapter 4, frequent aborts, especially in the presence of long-running transactions, may have a devastating effect on performance [11], therefore, reducing the number of aborts is an important challenge.

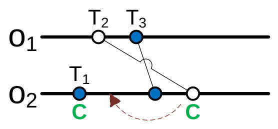
Of particular interest in this context is reducing the abort rate of read-only transactions. Read-only transactions play a significant role in various types of applications, including linearizable data structures with a strong prevalence of read-only operations [49], or client-server applications where an STM infrastructure replaces a traditional DBMS approach (e.g., FenixEDU web application [24]). Particularly long read-only transactions are employed for taking consistent snapshots of dynamically updated systems, which are then used for checkpointing, process replication, monitoring program execution, gathering system statistics, etc.

Unfortunately, long read-only transactions in current leading STMs tend to be repeatedly aborted for arbitrarily long periods of time. As we show below, the time for completing such a transaction varies significantly under contention, to the point that some read-only transactions simply cannot be executed without "stopping the world". As mentioned by Cliff Click [1], this kind of instability is one of the primary practical disadvantages of STM; Click mentions *multi-versioning* [16] (i.e., keeping multiple versions per object), as a promising way to make program performance more predictable.

Indeed, by keeping multiple versions it is possible to assure that each read-only transaction successfully commits by reading a *consistent snapshot* [15] of the objects it accesses. Consider, for example, the scenario depicted in Figure 5.1. In this run transaction T_2 reads an object o_1, then another transaction T_3 updates objects o_1 and o_2, and commits. Assume that T_2 now tries to read o_2. Reading the value o_2^2 written by T_3 would violate correctness, since T_2 does not read the value o_2^1 written by T_3. In a single-versioned STM, illustrated in Figure 5.1(a), T_2 must abort. However, a multi-versioned STM may keep both versions o_2^1 and o_2^2 of o_2, and may return o_2^1 to T_2, as illustrated in Figure 5.1(b). This allows T_2 to successfully commit, in spite of its conflict with T_3.



(a) Single-versioned TM, T_2 aborts.



(b) Multi-versioned TM, T_2 commits.

Figure 5.1: Keeping multiple versions avoids aborts, which are inevitable in STMs with only one object version.

We can capture the amount of spurious aborts that we allow using the notion of *permissive-*

ness. Some previously defined permissiveness conditions, such as *single-version* permissiveness [41], are too weak, and still allow many spurious aborts. Other permissiveness conditions, such as *online* π-permissiveness 4, prevent all spurious aborts, but require complex algorithms to implement (see Section 2.1 for details). In Section 5.2, we define the new notion of *multi-versioned (MV)* permissiveness. It ensures that read-only transactions never abort, and permits update transactions to abort only when they conflict with other update transactions. We consider a special class of responsive MV-permissive STMs, which do not allow a transaction to wait for other transactions' operations (a responsiveness notion is defined formally in Section 3). Responsive MV-permissiveness can be achieved by practical algorithms. In fact, the algorithms in [69, 22, 11] would all satisfy it if they kept enough object versions.

However, using multiple versions introduces the challenge of their efficient garbage collection. As demonstrated in Chapter 6, a simple approach of keeping a constant number of versions for each object does not provide enough of a performance benefit, and, even worse, can cause severe memory problems in long executions. Moreover, as we show below in Section 5.3, the memory consumption of algorithms keeping k versions per object might grow exponentially with the number of objects. The challenge is, therefore, to devise an approach for efficient management of old object versions. In Section 5.3, we show that this problem is inherent. We prove that no STM algorithm can be *space optimal*, i.e., ensure that it always maintains the minimum number of object versions possible. We then define an achievable GC property called *useless prefix (UP)* GC, based on maintaining object versions only when they may be needed by some existing read-only transactions.

Satisfying responsive MV-permissiveness (and UP GC) imposes costs on an STM. In Section 5.4, we show that a responsive MV-permissive STM cannot be *weakly disjoint-access parallel (DAP)*. Roughly speaking, this means that in order to ensure that read-only transactions never abort, it is necessary for transactions to communicate with each other, even when they do not access the same transactional objects. We also show that if a responsive STM is MV-permissive and satisfies UP GC, then read-only transactions must leave some trace of themselves in shared memory, even after they have committed. Note that this implies the STM cannot use *invisible reads* [32], an important technique for optimizing read-only transactions. We also note that if the UP GC requirement is omitted, then it is possible to implement an STM using invisible reads, as shown in Chapter 6, assuming there exists a garbage collection thread that sees the private ("invisible") memory of all transactions, such as the Java GC.

	Responsive MV-Permissiveness (Sec. 5.2)
Constant num of versions	Exponential memory growth (Sec. 5.3.1)
Space Optimality	Impossible (Sec. 5.3.2)
DAP	Impossible (Sec. 5.4.1)
UP GC (Sec. 5.3.3)	– Impossible when read-only transactions leave no trace after commit. (Sec. 5.4.2) – Possible: non-DAP and visible reads. (Sec. 5.5)

Table 5.1: Multi-versioning in STM: summary of limitations.

To complete our exploration of the design space of MV-permissiveness and garbage collection, we present in Section 5.5 a non-DAP algorithm using visible reads, satisfying responsive MV-permissiveness and UP GC. Our results are summarized in Table 5.1.

5.2 Multi-Versioned Permissiveness

One of the main benefits of multi-versioning is reducing the aborts rate. In order to evaluate the effectiveness of multi-versioned STMs, we need to formally define the set of aborts that are avoided. Such restrictions on aborts are captured by *permissiveness* conditions. In this section, we define a practically achievable permissiveness property that is suited for multi-versioned STMs.

Multi-versioning is particularly useful for avoiding aborts of read-only transactions. In fact, by keeping enough versions, read-only transaction can always find appropriate object versions to read, and commit successfully. Our permissiveness condition captures this property. Together with responsiveness, MV-permissiveness captures the property that read-only transactions neither abort nor block update transactions.

Definition 7. An STM satisfies multi-versioned (MV)-permissiveness if a transaction aborts only when it is an update transaction that conflicts with another update transaction.

We say that an STM satisfying MV-permissiveness is *MV-permissive*.

Some multi-versioned algorithms [69, 11] are not MV-permissive, because they do not always keep all the object versions needed to commit all read-only transactions. However, the algorithm we present in Section 5.5, as well as the algorithm presented in Chapter 6, are responsive and MV-permissive.

5.3 Garbage Collection Properties

A key aspect to maintaining multiple versions is a mechanism for garbage collecting (GC) old object versions. This section considers three sides to this problem. We first demonstrate that keeping a constant number of versions for each object can cause exponential memory growth in Section 5.3.1. In Section 5.3.2 we show that no STM can always keep the minimum number of old object versions. Then in Section 5.3.3, we define an achievable GC property that removes many old versions.

5.3.1 A Naïve Approach: Exponential Memory Growth

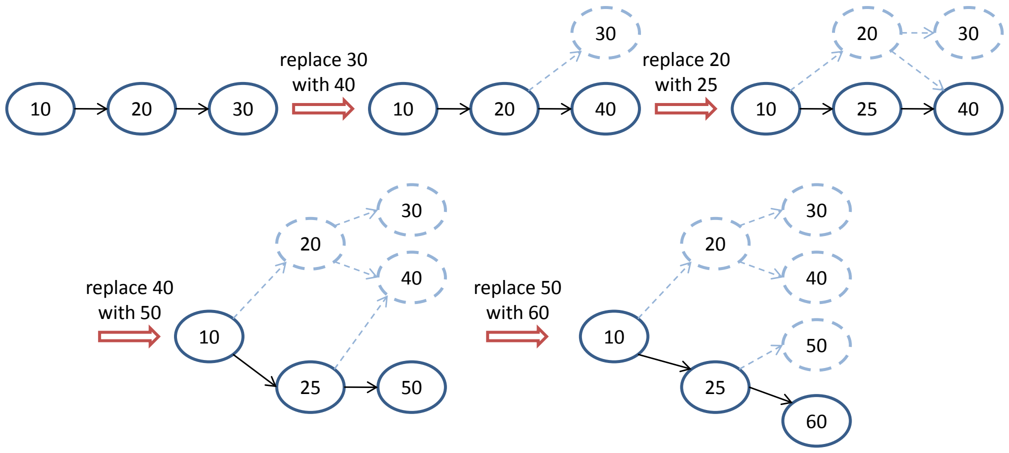


Figure 5.2: Example demonstrating exponential memory growth for an STM keeping only 2 versions of each object.

We first describe an inherent memory consumption problem of algorithms keeping a constant number of object versions. A naïve assessment of the memory consumption of a k-versioned STM would probably estimate that it takes up to k times as much more memory as a single-versioned STM.

We now illustrate that, in fact, the memory consumption of a k-versioned STM in runs with n transactional objects might grow like k^n. Intuitively, this happens because previous object versions continue to keep references to already deleted objects, which causes deleted objects to be pinned in memory.

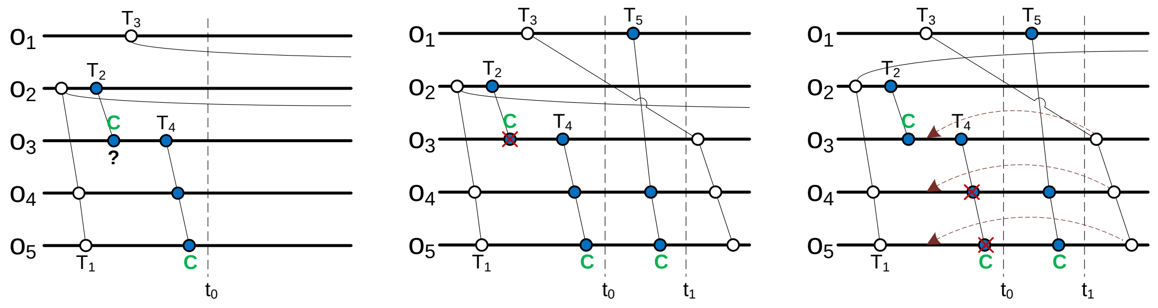
Consider, for example, a 2-versioned STM in the scenario depicted in Figure 5.2. The STM keeps a linked list of three nodes. When removing node 30 and inserting a new node 40 instead, node 30 is still kept as the previous version of $20.next$. Next, when node 20 is replaced with node 25, node 30 is still pinned in memory, as it is referenced by node 20. After several additional node replacements, we see that there is a complete binary tree in memory, although only a linked list is used in the application.

More generally, with a k-versioned STM, a linked list of length n could lead to $\Omega(k^n)$ node versions being pinned in memory (though being still linear to the number of write operations). This demonstrates an inherent limitation of keeping a constant number of versions per object. Our observation is confirmed by the empirical results shown in Section 6.3.5, where the algorithms keeping k versions cannot terminate in the runs with a limited heap size.

5.3.2 Impossibility of Space Optimal STM

Definition 8. A responsive MV-permissive STM $\mathcal{X}$ is *online space optimal*, if for any other responsive MV-permissive STM $\mathcal{X}'$ and any transactional history H, the number of versions kept by $\mathcal{X}$ at any point of time during H is less than or equal to the number of versions kept by $\mathcal{X}'$.

Theorem 17. *No responsive MV-permissive STM can be online space optimal.*



(a) An STM does not know whether to remove o_3^1.

(b) Removing o_3^1 leads to keeping the versions of o_4 and o_5 after they are overwritten.

(c) Keeping o_3^1 allows removing the versions of o_4 and o_5 after they are overwritten.

Figure 5.3: No STM can be online space optimal — it is not known at time t_0 whether to remove the version of o_3 written by T_2.

Proof. The main idea is to construct a transactional history in which any STM that keeps the minimum number of object versions at a time t_0 will keep more than the minimum number of object versions at time $t_1 > t_0$. Thus, no STM can keep the minimum number of versions at all times, and so is not online space optimal.

Formally, assume for contradiction that there exists an online space optimal STM $\mathcal{X}$ satisfying responsive MV-permissiveness. Consider the transactional history H depicted in Figure 5.3(a). At time t_0, $\mathcal{X}$ should either remove object version o_3^1 or keep it. We show that for either one of these decisions, there exists a responsive MV-permissive STM that keeps fewer versions than $\mathcal{X}$ during H or an extension of H.

Assume first that $\mathcal{X}$ keeps o_3^1 at time t_0. Consider another STM $\mathcal{X}'$ which behaves the same as $\mathcal{X}$ until time t_0, but GCs o_3^1 as soon as T_4 performs its write to o_3. Then $\mathcal{X}'$ keeps fewer object versions than $\mathcal{X}$. It remains to show that $\mathcal{X}'$ does not violate MV-permissiveness by GCing o_3^1. Notice that it suffices to show that at time t_0, all active read-only transactions, namely T_1 and T_3, can commit. Now, T_1's first read step precedes T_2's first write step. Thus, T_1 cannot read o_3^1 when invoking a read operation of o_3. $\mathcal{X}$ is MV-permissive, hence there exists a version $o_3^x \neq o_3^1$, which is kept by $\mathcal{X}$ at time t_0 and which can be read by T_1. Other than removing version o_3^1, $\mathcal{X}$ and $\mathcal{X}'$ are the same — T_1 can read o_1^x when invoking a read operation of o_3. Also, T_3 can return o_2^3, by serializing T_3 after T_4. So both T_1 and T_3 can commit after $\mathcal{X}'$ removes o_1^3, and so $\mathcal{X}'$ satisfies MV-permissiveness. Thus, $\mathcal{X}$ is not online space optimal.

Next, suppose that o_3^1 is GCed at time t_0. Consider the transactional history H_1 depicted in Figure 5.3(b), which extends H. We claim that the second step of T_3 cannot read o_3^0. Indeed, T_3 starts after T_2 finished, and T_2's second step overwrote o_3^0. So, T_3's second step must read o_3^2, and so T_4 precedes T_3 in any strict serialization. Also, T_3 precedes T_5 in any strict serialization, because the first step of T_3 does not read o_1^1. From this, we get that the third and fourth steps of T_3 must read o_4^1 and o_5^1, resp. So, these object versions cannot be GCed at time t_1. Now, to show that $\mathcal{X}$ is not online space optimal, consider another STM $\mathcal{X}'$ that keeps o_3^1 at time t_0, but GCs o_4^1 and o_5^1 at time t_0. We claim that $\mathcal{X}'$ satisfies MV-permissiveness. Again, it suffices to show the active read-transactions T_1 and T_3 at time t_0 can commmit. Indeed, T_1's second and third steps read o_4^0 and o_5^0, resp., so T_1 can commit. Also, T_3's second, third and fourth steps can read o_3^1, o_4^0 and o_5^0, resp., by serializing T_3 after T_2, and so T_3 can also commit. This is illustrated in Figure 5.3(c). Thus, $\mathcal{X}'$ satisfies MV-permissiveness. So, since $\mathcal{X}'$ keeps 6 object versions at t_1 and $\mathcal{X}$ keeps 7, $\mathcal{X}$ is not online space optimal.

□

5.3.3 Useless-Prefix GC

Though we have just seen that no responsive MV-permissive STM is online space optimal, we would still like an STM to garbage collect as many old versions as it can. To this end, we define the following.

Definition 9. An MV-permissive STM satisfies useless-prefix (UP) GC if at any point in a transactional history H, an object version o_i^j is kept only if there exists an extension of H with an active transaction T_i, such that (1) T_i can read o_i^j, and (2) T_i cannot read any version written after o_i^j.

In other words, an STM satisfying UP GC, removes the longest possible prefix of versions for each object at any point in time and keeps the shortest suffix of versions that might be needed by read-only transactions.

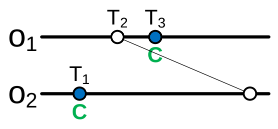
5.4 Inherent Limitations

In shared memory systems, cache contention due to concurrent memory accesses, and especially concurrent writes, is a significant performance bottleneck. Thus, it is desirable to try to separate the memory locations accessed by different transactions as much as possible. One natural requirement seems to be that transactions that access different transactional objects access only different base objects. However, we show in this section that MV-permissive STMs cannot satisfy this property.

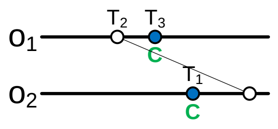
Another desirable property for an STM is not to update shared memory during read-only transactions. Such STMs are said to use *invisible reads*. It is easy to show that an STM satisfying MV-permissiveness and UP GC cannot use invisible reads. Indeed, UP GC requires knowing about existing read-only transactions, in order to determine which object versions to GC; such knowledge cannot be obtained unless read-only transactions write. In our second result in this section, we prove a stronger statement. We show that it is not possible for a responsive MV-permissive STM to perform UP GC, even when we allow read-only transactions to write, and only require that when such a transaction runs alone, the external configurations before and after the transaction are the same. This means that read-only transactions must leave some trace of their existence, even *after* they have committed. In particular, even keeping active readers lists for the objects [35], or using non-zero indicators for conflict detection [31] does not suffice.

5.4.1 Disjoint-Access Parallelism

Theorem 18. *A responsive STM satisfying MV-permissiveness cannot be weakly disjoint-access parallel.*



(a) H_1: $T_1 \preceq T_3$, T_2 must read the value written by T_1.



(b) H_2: $T_3 \preceq T_1$, T_2 cannot read the value written by T_1.

Figure 5.4: In a weakly DAP STM T_1 does not distinguish between H_1 and H_2 and cannot be MV-permissive.

Proof. Suppose for contradiction that there exists a responsive STM satisfying MV-permissiveness that is weakly DAP. Consider the transactional histories in Figure 5.4. In both H_1 and H_2, transactions T_2 and T_3 conflict on object o_1: T_3 writes to o_1 and commits, overriding the value read by an active transaction T_2. Note that since an STM is responsive and satisfies MV-permissiveness, T_3 neither aborts nor waits for T_2's termination upon a write to o_1. We claim the following. (1) The second step of T_2 returns o_2^1 in H_1. (2) The second step of T_2 returns o_2^1 in H_2. (3) The first step of T_2 returns o_1^0 in H_2. (4) H_2 is not strictly serializable if the first step of T_2 returns o_1^0, and the second step returns o_2^1. Conclusion (4) contradicts the strict serializability of the STM. So there is no responsive STM that is both MV-permissive and weakly DAP. In the following, let s_1, s_2, s_3 denote the first steps of T_1, T_2, T_3, resp., and let s_2' denote the second step of T_2.

To show (1), note that T_1 performs the last write on o_2 before the start of T_2 in H_1. So by strict serializability, s_2' returns o_2^1.

To show (2), we show that H_1 and H_2 are indistinguishable to $thr(T_2)$. We first claim that the base steps of s_1 and s_2 in H_1 do not contend. Indeed, consider another transactional history H_3 in which T_2 commits after its first step s_2. T_1 and T_2 are disjoint-access in H_3, so the base steps of s_1 and s_2 in H_3 do not contend. After s_2, $thr(T_1)$ and $thr(T_2)$ do not distinguish H_1 from H_3, because the steps of T_2 are not known ahead of time. Thus, the base steps of s_1 and s_2 in H_1 also do not contend. Next, we claim that the base steps of s_1 and s_3 in H_1 do not contend. This is because T_1 and T_3 are disjoint-access in H_3, so the base steps of s_1 and s_3 in H_3 do not contend. Since $thr(T_1)$

and $thr(T_2)$ do not distinguish H_1 from H_3 after s_3, then $thr(T_3)$ does not distinguish them after s_3. So, the base steps of s_1 and s_3 do not contend in H_1. Now, since the base steps for s_1, s_2 and s_1, s_3 in H_1 do not contend, then the configuration after the base steps of s_3 in H_1, and after the base steps of s_1 in H_2, are the same. Thus, $thr(T_2)$ does not distinguish between H_1 and H_2. So since s_2' returns o_2^1 in H_1, it also returns o_2^1 in H_2.

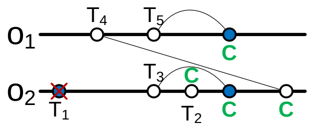
(3) is true because s_2 occurs before s_3 in H_2, and so s_2 returns o_1^0.

To show (4), let S be any legal sequential history that is equivalent to H_2. Since s_2 returns o_1^0 and s_2' returns o_2^1, then $T_2 \preceq_S T_3$ and $T_1 \preceq_S T_2$. Also, since T_1 starts after T_3 commits, then $T_3 \preceq_S T_1$. But then $T_1 \preceq_S T_2 \preceq_S T_3 \preceq_S T_1$, which is a contradiction. Thus, H_2 is not strictly serializable.

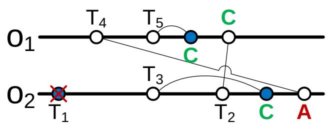
□

5.4.2 Read Visibility

Theorem 19. *Suppose a responsive STM satisfies MV permissiveness and UP GC. Consider a read-only transaction whose execution interval does not contain base steps of any other transaction. Then the configuration external to the transaction, immediately before and after the transaction, cannot be the same.*



(a) H_1: o_2^1 is GCed, T_4 can read o_2^2 and commits.



(b) H_2: o_2^1 is GCed, T_4 cannot read o_2^2 and aborts.

Figure 5.5: H_1 and H_2 are indistinguishable if a read-only transaction T_2 does not leave any trace after its execution.

Proof. Suppose for contradiction that there exists a responsive STM satisfying MV-permissiveness and UP GC, in which the external configurations before and after a read-only transaction are the same, when the transaction's interval does not overlap the steps of any other transaction. Consider the transactional histories in Figure 5.5. We claim the following. (1) o_2^1 is GCed in H_1. (2) o_2^1 is GCed in H_2. (3) T_4 aborts in H_2. Conclusion (3) is a contradiction, because T_4 is a read-only transaction, and cannot abort because of MV-permissiveness.

To show (1), first note that the second step of T_4 can read o_2^2, since this is equivalent to the legal sequential history $T_1T_2T_3T_4T_5$. Also, any read transaction that starts after H_1 follows T_3 in real-time, and so it cannot return o_2^1. Thus, in every extension of H, an active transaction can read o_2^2 or a later version. So by the definition of UP GC, o_2^1 is GCed.

We now show (2). In H_1 and H_2, T_2 is a read-only transaction, and its execution interval does not contain steps of any other transactions. So by assumption, the external configuration before and after T_2 are the same. Thus, after T_2's second step in H_2, the only thread that distinguishes between H_1 and H_2 is $thr(T_2)$. Note that $thr(T_2)$ does not GC o_2^1, since o_2^1 is the latest version of o_2 during T_2's execution interval. Then, since o_2^1 is GCed in H_1, it is also GCed in H_2.

To show (3), assume for contradiction that T_4 commits in H_2. Let S be a legal sequential history equivalent to H_2. Since o_1^2 is GCed in H_2, then T_4 must return o_2^2 in its second read step. Thus, we have $T_3 \preceq_S T_4$. Next, we have $T_4 \preceq_S T_5$, because T_4 does not read o_1^2 in its first read step. We have $T_5 \preceq_S T_2$, because T_2 starts after T_5 commits. Finally, we have $T_2 \preceq_S T_3$, because the first step of T_2 does not return o_2^2. Combining the above, we have $T_2 \preceq_S T_3 \preceq_S T_4 \preceq_S T_5 \preceq_S T_2$, which is a contradiction. Thus, T_4 does not commit in H_2, and so the lemma is proved.

□

5.5 UP Multi-Versioning Algorithm

We present *UP Multi-Versioning (UP-MV)*, a responsive STM algorithm satisfying MV-permissiveness and UP GC. Section 5.5.1 overviews the principles underlying UP-MV's design. The data structures used by UP-MV and its algorithm are described in Section 5.5.2. UP-MV's properties are analyzed in Section 5.5.3.

5.5.1 Algorithm Overview and Design Principles

First we explain how the algorithm finds the versions to read and write, and then explain the garbage collection mechanism.

Versions written and read. As UP-MV satisfies MV-permissiveness, each read-only transaction commits. Almost all responsive STMs abort an update transaction whenever its read-set is overwritten [47, 29, 69, 35]. Our first design principle mandates that we abort *only* in such situations:

Design Principle 1. Update transaction T aborts if and only if one of the objects in its read-set has been overwritten after being read by T and before T commits.

This rule is trivially checked at commit time by validating that each version in the read-set is still the latest one. To expedite these checks, we use a global version clock, as in TL2 [29] and LSA [69]. The clock is incremented by each committed transaction, and object versions are tagged with its values.

The writes to a transactional object o create a sequence of versions $o^0, o^1, \ldots$. Like [29, 35, 32], UP-MV defers the writes to commit time, and does not allow for "write reordering":

Design Principle 2. When an update transaction commits, it adds a new object version as the latest one.

Since update transactions abort whenever their read-set is overwritten, they read only the last object versions. A read-only transaction reads the latest version that it can read without violating correctness. To specify this, we define the transaction precedence relation recursively as follows: T_j *precedes* T_i if:

- T_j terminates before the start of T_i (real-time order);
- T_i reads the value written by T_j (read-after-write);
- T_i writes to object o_k, which was previously written to by T_j (write-after-write);
- T_i writes to object o_k and T_j reads the version overwritten by T_i (write-after-read); or
- $\exists T_k$ s.t. T_i precedes T_k and T_k precedes T_j.

If T_j precedes T_i, we say that T_i *follows* T_j. Note that any serialization order must respect the precedence order. We can now specify which versions are read:

Design Principle 3. Consider a transaction T_i reading object o_j. If T_i is an update transaction, it reads the latest version. Otherwise, let T_k be the earliest update transaction that follows T_i and writes to o_j. Then T_i reads the version of o_j overwritten by T_k. If no such T_k exists, T_i reads the last version of o_j.

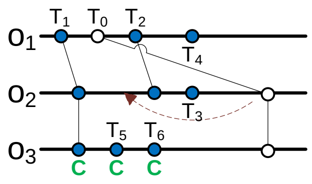


Figure 5.6: Transaction T_0 reads the latest object versions it can correctly read: when reading o_2 it accesses o_2^1, which was overwritten by T_2; when reading o_3, it accesses the last version.

For example, in Figure 5.6, when transaction T_0 reads o_2 it should read o_2^1, because this version is overwritten by T_2, which follows T_0 and writes to o_2. We say that an active transaction T_i is a *potential reader* of version o_i^j if T_i precedes o_i^{j+1}*.writer* and does not precede o_i^j*.writer*. In order to maintain the precedence information, UP-MV keeps a graph whose vertices are transactional descriptors for each transaction, and whose edges correspond to the precedence relations created by transactional steps during the run.

Note that if a read-only transaction does not conflict with any update transaction, then it has no following transactions, and therefore reads the last version of every object. Thus, by default, read-only transactions access the last object versions, which are referenced directly by *object handles*. In addition, each read-only transaction should be able to find references to relevant old object versions. But since, by UP GC, such versions may exist only as long as there are active transactions that can read them, these versions have to somehow be linked to their potential readers. This leads to the following design principle:

Design Principle 4. Every read-only transaction T has a *map* of references from objects to old versions of which T is a potential reader.

The responsibility for maintaining such maps lies on update transactions: before a committing update transaction writes to an object, it copies the reference to the overwritten version to all the maps of its active preceding transactions, (which are the potential readers of that version). The potential readers are found by traversing the precedence graph. In case the map already includes a version for this object, the version numbers are compared, and the earlier one is kept.

Garbage Collection. To satisfy the UP GC, an old object version is deleted at time t_0 if it cannot be read by any transaction after t_0. By Design Principle 3, version o_i^j may be read if and only if it has a potential reader. Version o_i^j is deleted at time t_0 if it may have no potential readers from t_0

onward. Our algorithm ensures that if there are no potential readers at time t_0, then no such readers may appear after t_0.

We deduce the following design rule for garbage collecting old object versions:

Design Principle 5. Every old object version is deleted when its last potential reader terminates.

In addition to removing old object versions, UP-MV's garbage collection should clean up transactional descriptors of terminated transactions from the precedence graph. As noted above, this graph is needed to allow committing transactions to copy overwritten versions to their active preceding transactions. Once a terminated transaction T has no active preceding transactions, its descriptor become useless. Hence:

Design Principle 6. The descriptor of terminated transaction T is deleted when the last active preceding transaction of T terminates.

5.5.2 UP-MV's Data Structures and Algorithm

Memory layout. The data structures used in the algorithm are depicted in Algorithm 3. Transactional objects are accessed via object handles, which point to the last object versions. In order to facilitate garbage collection, old versions are referenced directly by their potential readers.

Each version keeps a counter of potential readers, *potentialCount*; when this counter becomes zero the version is deleted. Additionally, each version keeps the version number, *versionNum*, as read from the global clock when the version is written. Each object version also keeps the list of its current active reading transactions, *readers*, which is used by update transactions to maintain precedence information. This is where the algorithm violates read invisibility, as required for UP GC (see Section 5.4.2).

Each transaction is represented by its transactional descriptor keeping the read-set and the write-set of the accessed objects. A data structure TxnMap keeps pointers to all the non-GCed transactions' descriptors. Some of the transactional descriptors point to each other, forming a subgraph of the precedence graph. Transactional steps add edges according to read-after-write, write-after-write, and write-after-read relations. Edges reflecting real-time precedence are added at startup, as we explain below. The transactional descriptor of a terminated transaction is GCed

Algorithm 3 UP-MV algorithm data structures.

```
1: Object Handle $o_j$:
2:     Version: latest ▷ latest version of the object
3: Version $o_j^k$:
4:     Data: data ▷ actual data
5:     Tid: writerId ▷ Id of the version's writer
6:     int: versionNum ▷ ordered version number of $o_j^k$
7:     TxnDsc[]: readers ▷ current active readers
8:     int: potentialCount ▷ the number of active read-only transactions that might need the version in future
9: TxnDsc $T_i$:
10:    {Active, Terminated}: status
11:    int: clockVal ▷ global clock at the beginning of transaction
12:    ⟨Object, Version⟩[]: readSet
13:    ⟨Object, Version⟩[]: writeSet
14:    TxnDsc[]: prev ▷ immediate predecessors of $T_i$
15:    TxnDsc[]: next ▷ immediate successors of $T_i$
16:    ⟨Object, Version⟩[]: toRead ▷ if $T_i$ cannot read the latest version of $o_j$, then the legal version is kept in
       $T_i$.toRead[$o_j$]
17: Global Variables:
18:    int: globalClock ▷ incremented by committing update txn
19:    TxnDsc[]: finished ▷ finished txns that have not been GCed
20:    ⟨Tid, TxnDsc⟩[]: txnMap
```

once it has no incoming edges. If transaction T_i has no active preceding transactions at the end of its run, T_i's descriptor is deleted by T_i itself. Otherwise, T_i's descriptor is deleted by the last active transaction preceding T_i when it terminates.

In order to track real-time order, the algorithm maintains a global transaction set *finished*, which holds the descriptors of all the terminated transactions that have not been GCed. A transaction T that cannot GC its descriptor inserts it to this set upon termination, and the descriptor is removed from *finished* when it is GCed. Note that *finished* is always empty in runs without conflicts. When a new transaction starts, it adds edges from every transaction in *finished* to itself. The use of this set is where the algorithm violates the DAP property, as necessary for responsive MV-permissiveness (see Section 5.4.1). Although the use of a global clock, which is incremented by each committing transaction, and copied to every written version, also violates DAP, we use it only to optimize consistency checks, and it is not needed for correctness.

In Figure 5.7, we see the memory layout for the scenario depicted in Figure 5.6: an active read-only transaction T_0 precedes committed transactions $T_2 \ldots T_4$, so these transactions are not

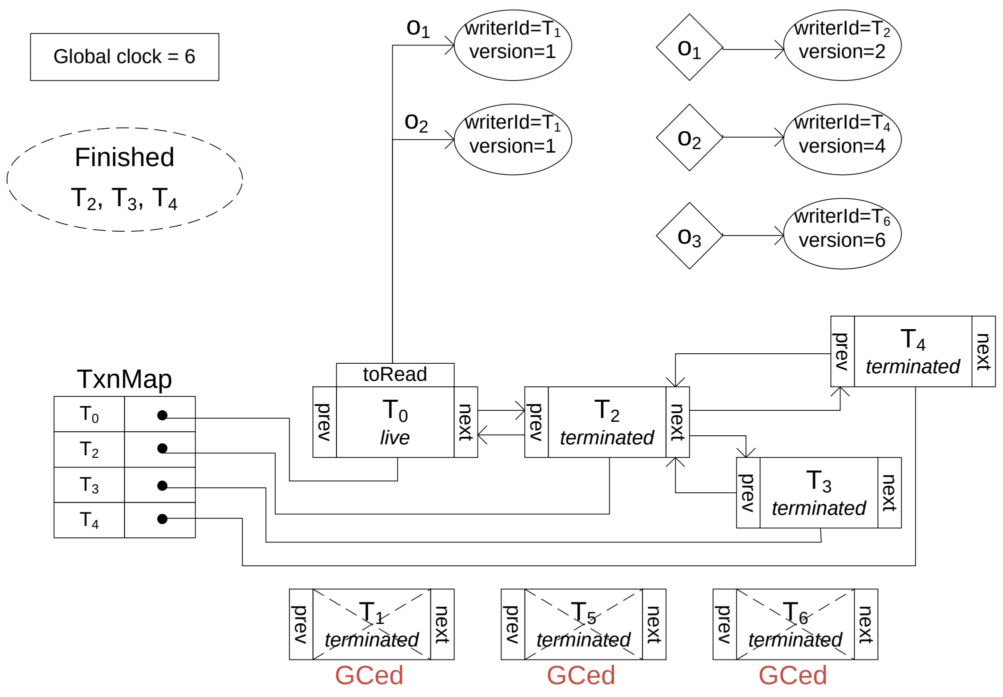


Figure 5.7: An example of memory layout: object handles keep last versions only, old versions are kept as long as they have potential readers, terminated transactions are GCed once they have no active preceding transactions.

GCed, whereas committed transactions T_1, T_5, T_6, which have no active preceding transactions, are deleted.

The map of old object versions T_i may read is stored in T_i*.toRead*. As we show in Section 5.5.3, if a read-only transaction T_i cannot read the last version of object o_j, then T_i.toRead contains a mapping from o_j to the old version that should be read by T_i. In Figure 5.7, the object versions overwritten by T_1 are referenced by its active preceding transaction T_0. All other old object versions are GCed because they have no potential readers.

We now describe an UP-MV algorithm. The description is simplified by the model's assumption that all the base steps for running a transactional operation appear to execute atomically. In practice, this atomicity can be achieved by using locks, as is done in TL2 [29], or by lock-free algorithms [35]. This issue is out of the scope of the paper.

Handling update transactions. The pseudo-code for update transaction T_i is depicted in Algorithms 4 and 5. At startup, transaction T_i saves the value of the global clock in its local variable

Algorithm 4 UP-MV algorithm for update transaction T_i.

```
1:  Write to $o_j$:
2:     if ($o_j \in T_i$.writeSet) then update $T_i$.writeSet[$o_j$]; return
3:     localCopy $\leftarrow o_j$.latest.clone()
4:     writeSet[$o_j$] $\leftarrow$ localCopy
5:     update localCopy
6:  Read $o_j$:
7:     if ($o_j \in T_i$.writeSet) then return $T_i$.writeSet[$o_j$]
8:     version $\leftarrow o_j$.latest
9:     if (version.versionNum $> T_i$.clockVal) then
10:       if $\neg$validateReadSet() then abort
11:       clockVal $\leftarrow$ version.versionNum
       ▷ update precedence information
12:    lastWriter $\leftarrow$ txnRepository.get(version.writerId)
13:    if (lastWriter $\neq \perp$) then addEdge(lastWriter, $T_i$)
14:    version.readers $\leftarrow$ version.readers $\cup T_i$
15:    readSet[$o_j$] $\leftarrow$ version
16:    return version.data
17: Commit:
18:    if $\neg$validateReadSet() then abort
19:    overwritten $\leftarrow \emptyset$                    ▷ keep the versions overwritten by $T_i$
20:    globalClock $\leftarrow$ globalClock + 1
21:    foreach $o_j \in T_i$.writeSet do:
          ▷ update precedence info
22:       prevWriter $\leftarrow$ txnRepository.get($o_j$.latest.writerId)
23:       if (prevWriter $\neq \perp$) then addEdge(prevWriter, $T_i$)
24:       foreach $T_j \in o_j$.latest.readers do: addEdge($T_j, T_i$)
          ▷ install the new version
25:       $o_j$.latest.potentialReadersCount $\leftarrow 0$
26:       overwritten[$o_j$] $\leftarrow o_j$.latest
27:       localCopy.versionNum $\leftarrow$ globalClock
28:       $o_j$.latest $\leftarrow$ localCopy
       ▷ pass the overwritten versions to the txns preceding $T_i$
29:    foreach $T_j \in T_i$.prev do:
30:       overwrittenVersions($T_j$, overwritten)
       ▷ delete the unnecessary overwritten versions (have no potential readers)
31:    foreach $\langle o_i, \text{ver}_i \rangle \in$ overwritten do:
32:       if (ver$_i$.potentialCount $= 0$) then delete ver$_i$
```

clockVal and adds edges from all the descriptors in *finished* to itself (line 37).

Write operations postpone most of the work till the commit phase; a write operation merely

Algorithm 5 UP-MV algorithm for update transaction T_i.

```
33: Startup:
34:    $T_i$.status ← Active
35:    $T_i$.clockVal ← globalClock
36:    foreach $T_j \in$ finished do:
37:       addEdge($T_j, T_i$)                                    ▷ RTO dependence
38: Termination:
39:    $T_i$.status ← Terminated
40:    finished ← finished $\cup T_i$
41:    GC($T_i$)
42: Function GC($T_i$)
       ▷ remove the transactions with no active preceding transactions
43:    if ($T_i$.prev $= \emptyset$) then
44:       txnRepository ← txnRepository $\setminus T_i$
45:       finished ← finished $\setminus T_i$
46:       foreach $\langle o_j$, version$\rangle \in T_i$.readSet do:
47:          version.readers ← version.readers $\setminus T_i$
48:       foreach $T_j \in T_i$.next do:
49:          $T_j$.prev ← $T_j$.prev $\setminus T_i$
50:          GC($T_j$)
51:       delete $T_i$'s descriptor
52: Function validateReadSet()
53:    foreach $\langle o_j$, version$\rangle \in T_i$.readSet do:
54:       if $o_j$.latest $\neq$ version then return false
55:    return true
56: Function overwrittenVersions($T_j$, overwritten)
57:    if ($T_j$.status $=$ Active) then
58:       foreach $\langle o_i$, ver$_i\rangle \in$ overwritten do:
59:          curVer ← $T_j$.toRead$[o_i]$
60:          if (curVer $= \perp \vee$ curVer.versionNum $>$ ver$_i$.versionNum)
61:             ver$_i$.potentialCount++
62:             $T_j$.toRead$[o_i]$ ← ver$_i$
63:    foreach $T_k \in T_j$.prev do:
64:       overwrittenVersions($T_k$, overwritten)
```

updates the local copy of the object and puts it in its write-set. A read operation may only return the last version of the object. To that end, the last version's number is validated. If a read operation succeeds, T_i updates the precedence information: if the last version's writer T_j was not GCed, then T_i adds an edge from T_j to itself.

Transaction T_i commits successfully if and only if no object in its read-set is overwritten after being read by T_i and before T_i commits. This is checked similarly to TL2 [29], using the global clock, and without using precedence information. A commit operation starts by revalidating T_i's read-set (line 18). If the validation fails, T_i aborts. Otherwise, T_i executes the following: 1) increments the global clock; 2) for each $o_j \in T_i$.writeSet, T_i adds edges from o_j's writer and from o_j's readers to itself, and then installs the new version (lines 22–28); and 3) calls the function overwrittenVersions to update potential readers' maps with the versions overwritten by T_i (line 30).

The process of updating potential readers with overwritten versions (lines 56–64) is executed recursively for every preceding transaction. For an active transaction T_j, the overwritten versions are inserted to its *toRead* map. If for some object o_i, *toRead* already contains a version of o_i, the version with the smaller versionNum is chosen (lines 59–62). This way, the algorithm guarantees that a read-only transaction that reads o_i accesses the version overwritten by the *earliest* following transaction.

When T_i terminates, it adds its descriptor to *finished* and starts the GC procedure (lines 42–51). The transactional descriptor may be deleted if it has no incoming edges. Since deleting one transactional descriptor decreases the number of incoming edges in its successors, the GC continues recursively with them.

Handling read-only transactions. The pseudo-code for read-only transactions appears in Algorithm 6. To read object o_j (lines 3–6), T_i checks whether the object is in *toRead*. If not, then T_i reads the last version of o_j. Otherwise, T_i reads the version from its *toRead* list.

When a read-only transaction T_i terminates, it decrements the counter of potential readers for all the versions in its *toRead* list. If a version's number of potential readers becomes zero, the old object version is deleted (lines 82–84).

5.5.3 Properties

We first show that UP-MV algorithm satisfies the invariant, which describes the contents of *toRead* map in the following way:

Invariant 1. Transaction T_i has o_i^j in its *toRead* map if and only if o_i^j is not o_i's last version and o_i^j is the latest version that T_i can read without violating correctness.

Algorithm 6 UP-MV algorithm for read-only transaction T_i.

```
Read $o_j$:
    if ($o_j \in T_i$.readSet) then return readSet[$o_j$].data
    ▷ find the version to read
    if ($o_j \in T_i$.toRead) then
        verToRead $\leftarrow T_i$.toRead[$o_j$]
    else
        verToRead $\leftarrow o_j$.latest
    ▷ update precedence information
    writer $\leftarrow$ txnRepository.get(verToRead.writerId)
    if (writer $\neq \perp$) then
        addEdge(writer, $T_i$)
    ▷ pass the overwritten versions to the preceding transactions
    foreach $T_j \in T_i$.prev do:
        overwrittenVersions($T_j, T_i$.toRead)
    verToRead.readers $\leftarrow$ verToRead.readers $\cup T_i$
    readSet[$o_j$] $\leftarrow$ verToRead
    return verToRead.data
Termination:
    $T_i$.status $\leftarrow$ Terminated
    finished $\leftarrow$ finished $\cup\ T_i$
    foreach $\langle o_j$, oldVersion$\rangle \in T_i$.toRead do:
        oldVer.potentialCount $\leftarrow$ oldVer.potentialCount $- 1$
        if (oldVer.potentialCount $= 0$) then delete oldVersion
    GC($T_i$)
```

Proof. We proof consists of the following steps: (1) we first show that transaction T_i can read object version o_i^j without violating correctness if and only if T_i does not precede o_i^j.*writer*; (2) we then show that if S_i^j is the set of committed update transactions following T_i that write to o_j, then T_i's map contains the first version of o_j that is overwritten by a transaction in S_i^j. Invariant proof follows directly from step (1) and (2).

In the previous chapter we shown that history H has a legal serialization if and only if its precedence graph is acyclic. We use this property to prove the following lemma:

Lemma 20. *Transaction T_i can read object version o_i^j without violating correctness if and only if T_i does not precede o_i^j.writer.*

Proof. T_i can correctly read o_i^j if and only if the read operation does not create a cycle in the precedence graph. When T_i reads o_i^j, two new precedence relations are added: $(o_i^j.\mathit{writer}, T_i)$ and

$(T_i, o_i^{j+1} \textit{writer})$.

($\Rightarrow$:) If T_i already precedes $o_i^j.\textit{writer}$, then adding relation $(o_i^j.\textit{writer}, T_i)$ creates a cycle in the precedence graph, so T_i cannot read o_i^j.

($\Leftarrow$:) The algorithm always installs new versions at the end, so $o_i^j.\textit{writer}$ precedes $o_i^{j+1}.\textit{writer}$. If T_i does not precede $o_i^j.\textit{writer}$, then adding relations $(o_i^j.\textit{writer}, T_i)$ and $(T_i, o_i^{j+1}.\textit{writer})$ cannot create a cycle in the precedence graph. Therefore, T_i can read o_i^j if it does not precede $o_i^j.\textit{writer}$. □

The following lemma can be proven by easy induction on the steps of the algorithm:

Lemma 21. *The transactional descriptor graph of UP-MV is at any given time a subgraph of the precedence graph, which includes a path from every active transaction T_i to each of its followers.*

Invariant 2. Let S_i^j be the set of committed update transactions following T_i that write to o_j. If S_i^j is empty then T_i's map contains no mapping for o_j. Otherwise, T_i's map contains the first version of o_j that is overwritten by a transaction in S_i^j.

Proof. We prove the invariant by showing that it is correct at the beginning of each transaction and is preserved after each algorithm step. Upon startup, T_i's map is empty, and T_i does not precede any other transaction, so $S_i^j = \emptyset$. Hence, the invariant holds.

In order to show that the invariant is preserved after each algorithm's operation, we show the following: (1) each change in T_i's mapping for o_j corresponds to a change in S_i^j, (2) S_i^j may only grow during the lifetime of T_i, (3) the invariant is preserved when a new transaction joins S_i^j. These three steps together complete the proof.

To show (1), observe that T_i's mapping for o_j changes only in the function *overwrittenVersions* (line 62), which operates on T_i's descriptor as a result of one of two events. First, a transaction T_k that follows T_i writes to o_j and commits (line 30, and recursively, line 62). In this case, $T_k \in S_i^j$. Second, an active read-only transaction T_k, which precedes some T_l that writes o_j, reads the value written by one of T_i's followers (line 75). In this case, T_i starts preceding T_l via T_k, and therefore $T_j \in S_i^j$. Each member of S_i^j writes to o_j once, Therefore, every change in T_i's mapping for o_j corresponds to a change of S_i^j.

Claim (2) follows directly from the observation that if T_i precedes T_j at time t_0, this relation persists in every extension of the history.

To show (3), we examine all the possible ways for a new transaction to join S_i^j:

A transaction T_l that follows T_i writes to o_j and commits. In this case, T_l calls overwrittenVersions() (line 30), which traverses recursively all the predecessors of T_l's descriptor. By Lemma 21, T_i's descriptor is a predecessor of T_l's descriptor, hence overwrittenVersions() is executed with T_i, and in line 62, it compares T_i's mapping for o_j with the version overwritten by T_j, and chooses the version with the earlier version number. The invariant is preserved.

A (committed) writer of o_j has a new preceding transaction T_l. This happens only when an active read-only transaction T_k, which precedes T_l, reads a value written by one of T_i's followers. In this case, T_k calls overwrittenVersions() (line 75), which traverses recursively all the predecessors of T_k's descriptor, including T_i's descriptor (by Lemma 21). According to the invariant assumption, T_k's mapping for o_j contains the version of o_j that is overwritten by the earliest transaction in S_k^j. This version is compared with T_i's current mapping for o_j in line 62, and the version with the earlier version number is chosen. Note that if an update transaction $T_m \in S_k^j$ is not the earliest one in S_k^j, then it cannot be the earliest one in S_i^j, because $S_k^j \subseteq S_i^j$. Therefore, the invariant is preserved.

□

Invariant 1 follows directly from Lemma 20 and Invariant 2. □

Lemma 22. *UP-MV is a responsive algorithm that satisfies MV-permissiveness.*

Proof. According to the algorithm, neither read-only nor update transactions never wait for other transactions' operations meaning that UP-MV is a responsive algorithm.

Update transaction T_i aborts only if the function *validateReadSet()* returns false, which happens if an object version in T_i's read-set is not the latest version of the corresponding object (line 54). In other words, T_i aborts only if another transaction writes to an object from T_i's read-set after T_i's start — T_i aborts only upon a conflict with another update transaction. Read-only transaction never aborts. Hence, UP-MV satisfies MV-permissiveness. □

Lemma 23. *UP-MV satisfies useless prefix GC.*

Proof. According to the algorithm a version is deleted if it is overwritten and its counter of potential readers arrives to 0 (lines 32 and 84). Thus, the overwritten version is kept only if it belongs to the map of some read-only transaction.

By Invariant 1, T_i has o_i^j in its *toRead* map only if o_i^j is the last version T_i can read without violating correctness. Therefore, o_i^j is kept in memory as long as there exists a transaction that can read o_i^j and cannot read any other version written after o_i^j. □

Lemma 24. *UP-MV satisfies strict serializability.*

Proof. In order to show that the algorithm satisfies strict serializability we need to show that any history H of UP-MV has an equivalent sequential history. In order to satisfy this property it is enough to preserve the precedence graph acyclic (see Chapter 4).

We first show that the transactional descriptors form a correct precedence graph: (1) the edges corresponding to the real-time order are added in line 37; (2) read-after-write edges are added in line 13; (3) write-after-read edges are added in line 73; and (4) write-after-write edges are added in line 23.

It now remains to show that the graph formed by the transactional descriptors remains acyclic throughout the run. Update transactions have no followers as long as they are active because they abort on every conflict (lines 10 and 18), and so their steps cannot create a cycle. According to Invariant 1 and lines 67–70, a read-only transaction reads the latest possible version that does not create a cycle in the precedence graph.

Therefore, UP-MV algorithm maintains a precedence graph acyclic and satisfies strict serializability. □

Chapter 6

SMV: Selective Multi-Versioning STM

In this chapter we present *Selective Multi-Versioning (SMV)*, a new STM that reduces the number of aborts, especially those of long read-only transactions. SMV keeps old object versions as long as they might be useful for some transaction to read. It is able to do so while still allowing reading transactions to be invisible by relying on automatic garbage collection to dispose of obsolete versions.

SMV is most suitable for read-dominated workloads, for which it performs better than previous solutions. It has an up to $\times 7$ throughput improvement over a single-version STM and more than a two-fold improvement over an STM keeping a constant number of versions per object, while operating successfully even in systems with stringent memory constraints.

A preliminary version of the work presented in this chapter appears in proceedings of the 25th International Symposium on Distributed Computing (DISC 2011).

6.1 Introduction

As demonstrated in Chapter 5, maintaining multiple versions in an STM is a challenging task. While no STM can be space optimal (Section 5.3.2), we proposed a relaxed GC condition called useless-prefix GC, and developed a theoretical MV-permissive algorithm satisfying this property (Section 5.5). Unfortunately, useless-prefix GC demands that every read-only transaction must leave some trace of its existence even after it has committed (Section 5.4.2), which devastates

STM performance.

In this chapter we present *Selective Multi-Versioning (SMV)*, an STM algorithm that keeps old object versions that are still useful to potential readers, while allowing read-only transactions to remain invisible, i.e., having no effect on shared memory. At first glance, combining invisible reads with effective garbage collection may seem impossible — if read-only transactions are invisible, then other transactions have no way of telling whether potential readers of an old version still exist! To circumvent this apparent paradox, we exploit separate GC threads, such as those available in managed memory systems. Such threads have access to all the threads' private memories, so that even operations that are invisible to other transactions are visible to the garbage collector. SMV ensures that old object versions become *garbage collectible* once there are no transactions that can safely read them.

In Section 6.3 we evaluate different aspects of SMV's performance. We implement SMV in Java and study its behavior for a number of benchmarks (red-black tree microbenchmark, STMBench7 [44] and Vacation [23]). We compare SMV to a TL2-style single-versioned STM [29], to a k-versioned variant of the same algorithm, which keeps k versions per object similarly to LSA [69], and to a simple global read-write lock approach.

We find that SMV is extremely efficient for read-dominated workloads with long-running transactions. For example, in STMBench7 with 64 threads, the throughput of SMV is seven times higher than that of TL2 and more than double than those of 2- and 8-versioned STMs. Furthermore, in an application with one thread constantly taking snapshots and the others running update transactions, neither TL2 nor the k-versioned STM succeeds in taking a snapshot, even when only one concurrent updater is running. The performance of SMV remains stable for any number of concurrent updaters.

We compare the memory demands of the algorithms by limiting Java heap size. Whereas k-versioned STMs crash with a Java `OutOfMemoryException`, SMV continues to run, and its throughput is degraded by less than 25% even under stringent memory constraints.

SMV presents the new approach for keeping multiple versions, which allows read-only transactions to stay invisible and delegates the cleanup task to the already existing GC mechanisms.

6.2 SMV Algorithm

We present Selective Multi-Versioning, a new object-based STM. The data structures used by SMV are described in Section 6.2.1 and Section 6.2.2 depicts the algorithm.

6.2.1 Overview of Data Structures

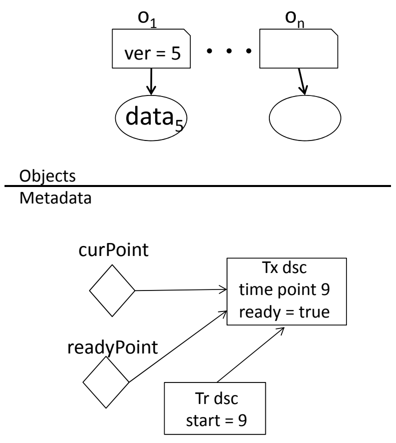
SMV's main goal is to reduce aborts in workloads with read-only transactions, without introducing high space or computational overheads. SMV is based on the following design choices: 1) Read-only transactions do not affect the memory that can be accessed by other transactions. This property is important for performance in multi-core systems, as it avoids cache thrashing issues [32, 69]. 2) Read-only transactions always commit. A read-only transaction T_i observes a consistent snapshot corresponding to T_i's start time — when T_i reads object o_j, it finds the latest version of o_j that has been written before T_i's start. 3) Old object versions are removed once there are no live read-only transactions that can consistently read them. To achieve this with invisible reads, SMV relies on the omniscient GC mechanism available in managed memory systems.

We now give a brief reminder of such a mechanism. An object can be reclaimed by the garbage collector once it becomes unreachable from the call stack or global variables. Reachability is a transitive closure over *strong* memory references: if a reachable object o_1 has a strong reference to o_2, then o_2 is reachable as well (strong references are the default ones in Java). In contrast, *weak references* [39] do not protect the referenced object from being GCed; an object referenced by weak references only is considered unreachable and may be removed.

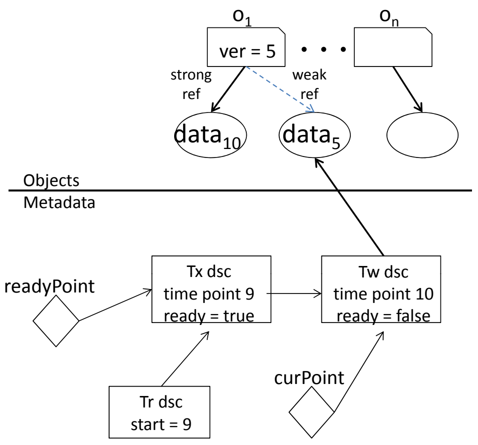
As in other object-based STMs, transactional objects in SMV are accessed via *object handles*. An object handle includes a history of object values, where each value keeps a *versioned lock* [29] – data structure with a version number and a lock bit. In order to facilitate automatic garbage collection, object handles in SMV keep strong references only to the latest (current) versions of each object, and use weak references to point to other versions.

Each transaction is associated with a *transactional descriptor*, which holds the relevant transactional data, including a read-set, a write-set, status, etc. In addition, transactional descriptors play an important role in keeping strong references to old object versions, as we explain below.

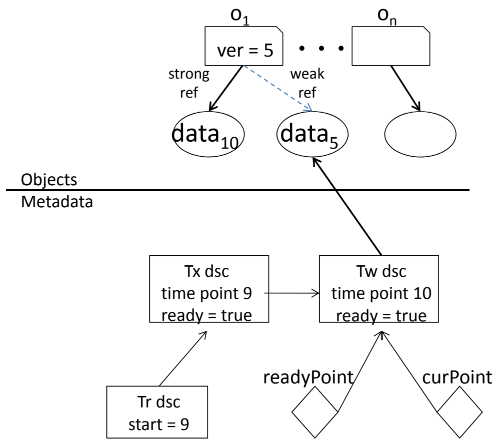
Version numbers are generated using a global version clock, where transactional descriptors act as "time points" organized in a one-directional linked list. Upon commit, an update transaction



(a) T_r's descriptor points to the latest committed transaction.



(b) T_w commits and begins write-back.



(c) T_w's write-back is finished.

Figure 6.1: Transactional descriptor of T_w references the over-written version of o_1 (data$_5$). This way, read-only transaction T_r keeps a reference chain to the versions that have been overwritten after T_r's start.

appends its transactional descriptor to the end of the list (a special global variable *curPoint* points to the latest descriptor in this list). For example, if the current global version is 100, a committing update transaction sets the time point value in its transactional descriptor to 101 and adds a pointer to this descriptor from the descriptor holding 100.

Version management is based on the idea that old object versions are pointed to by the descriptors of transactions that over-wrote these versions (see Figure 6.1). A committing transaction T_w includes in its transactional descriptor a strong reference to the previous version of every object in its write set before diverting the respective object handle to the new version.

When a read-only transaction T_i begins, it keeps (in its local variable *startTP*) a pointer to the latest transactional descriptor in the list of committed transactions. This pointer is cleared upon commit, making old transactional descriptors at the head of the list GCable.

This way, active read-only transaction T_r keeps a reference chain to version o_i^j if this version was over-written after T_r's start, thus preventing o_i^j's garbage collection. Once there are no active read-only transactions that started before o_i^j was over-written, this version stops being referenced and thus becomes GCable .

Figure 6.1 illustrates the commit of an update transaction T_w that writes to object o_1 (the use of *readyPoint* variable will be explained in Section 6.2.3). In this example, T_w and a read-only transaction T_r both start at time 9, and hence T_r references the transactional descriptor of time point 9. The previous update of o_1 was associated with version 5. When T_w commits, it inserts its transactional descriptor at the end of the time points list with value 10. T_w's descriptor references the previous value of o_1. This way, the algorithm creates a reference chain from T_r to the previous version of o_1 via T_w's descriptor, which ensures that the needed version will not be GCed as long as T_r is active.

6.2.2 Basic Algorithm

We now describe the SMV algorithm. For the sake of simplicity, we present the algorithm in this section using a global lock for treating concurrency on commit — in Section 6.2.3 we show how to remove this lock.

SMV handles read-only and update transactions differently. We assume that transaction's type can be provided to the algorithm beforehand by a compiler or via special program annotations. If not, each transaction can be started as read-only and then restarted as update upon the first occurrence of a write operation.

Handling update transactions. The protocol for update transaction T_i is depicted in Algorithm 7. The general idea is similar to the one used in TL2 [29]. An update transaction T_i aborts if some object o_j read by T_i is over-written after T_i begins and before T_i commits. Upon starting, T_i saves the value of the latest time point in a local variable *startTime*, which holds the latest time at which an object in T_i's read-set is allowed to be over-written.

A read operation of object o_j reads the latest value of o_j, and then post-validates its version (function *validateRead*. The validation procedure checks that the version is not locked and it is not

Algorithm 7 SMV algorithm for update transaction T_i.

```
1:  **Upon Startup:**
2:      $T_i$.startTime $\leftarrow$ curPoint.commitTime
3:  **Read** $o_j$:
4:      **if** ($o_j \in T_i$.writeSet)
5:          **then return** $T_i$.writeSet[$o_j$]
6:      data $\leftarrow o_j$.latest
7:      **if** $\neg$validateRead($o_j$) **then abort**
8:      readSet.put($o_j$)
9:      **return** data
10: **Write to** $o_j$:
11:     **if** ($o_j \in T_i$.writeSet)
12:         **then update** $T_i$.writeSet.get($o_j$); **return**
13:     localCopy $\leftarrow o_j$.latest.clone()
14:     update localCopy; writeSet[$o_j$] $\leftarrow$ localCopy
15: **Function validateReadSet**
16:     **foreach** $o_j \in T_i$.readSet **do:**
17:         **if** $\neg$validateRead($o_j$) **then return false**
18:     **return true**
19: **Commit:**
20:     **foreach** $o_j \in T_i$.writeSet **do:** $o_j$.lock()
21:     **if** $\neg$validateReadSet() **then abort**
        $\triangleright$ txn dsc should reference the over-written data
22:     **foreach** $o_j \in T_i$.writeSet **do:**
23:         $T_i$.prevVersions.put($\langle o_j$, $o_j$.latest$\rangle$)
24:     timeLock.lock()
25:     $T_i$.commitTime $\leftarrow$ curPoint.commitTime $+ 1$
        $\triangleright$ update and unlock the objects
26:     **foreach** $\langle o_j$, data$\rangle \in T_i$.writeSet **do:**
27:         $o_j$.version $\leftarrow T_i$.commitTime
28:         $o_j$.weak_references.append($o_j$.latest)
29:         $o_j$.latest $\leftarrow$ data; $o_j$.unlock()
30:     curPoint.next $\leftarrow T_i$; curPoint $\leftarrow T_i$
31:     timeLock.unlock()
32: **Function validateRead**(Object $o_j$)
33:     **return** ($\neg o_j$.isLocked $\wedge$ $o_j$.version $\leq T_i$.startTime)
```

greater than T_i.startTime, otherwise the transaction is aborted.

A write operation (lines 12–14) creates a copy of the object's latest version and adds it to T_i's local write set.

Commit (lines 20–31) consists of the following steps:

1. Lock the objects in the write set (line 20). Deadlocks can be detected using standard mechanisms (e.g., timeouts or Dreadlocks [53]), or may be avoided if acquired in the same order by every transaction.

2. Validate the read set (function *validateReadSet*).

3. Insert strong references to the over-written versions to T_i's descriptor (line 23). This way the algorithm guarantees that the over-written versions stay in the memory as long as T_i's descriptor is referenced by some read-only transaction.

4. Lock the time points list (line 24). Recall that this is a simplification; in Section 6.2.3 we show how to avoid such locking.

5. Set the commit time of T_i to one plus the value of the commit time of the descriptor referenced by *curPoint*.

6. Update and unlock the objects in the write set (lines 26–29). Set their new version numbers to the value of T_i.commitTime. Keep weak references to old versions.

7. Insert T_i's descriptor to the end of the time points list and unlock the list (line 30).

Algorithm 8 SMV algorithm for read-only transaction T_i.

```
1: Upon Startup:
2:     $T_i$.startTP $\leftarrow$ curPoint
3: Read $o_j$:
4:     latestData $\leftarrow$ $o_j$.latest
5:     if ($o_j$.version $\leq$ $T_i$.startTP.commitTime) then return latestData
6:     return the latest version ver in $o_j$.weak_references, s.t.
7:         ver.version $\leq$ $T_i$.startTP.commitTime
8: Commit:
9:     $T_i$.startTP $\leftarrow$ $\perp$
```

Handling read-only transactions. The pseudo-code for read-only transactions appears in Algorithm 8. Such transactions always commit without waiting for other transactions to invoke any operations. The general idea is to construct a consistent snapshot based on the start time of T_i. At startup, T_i.*startTP* points to the latest installed transactional descriptor (line 2); we refer to the time value of startTP as *T_i's start time*.

For each object o_j, T_i reads the latest version of o_j written before T_i's start time. When T_i reads an object o_j whose latest version is greater than its start time, it continues to read older versions until it finds one with a version number older than its start time. Some old enough version is guaranteed to be found, because the updating transaction T_w that over-wrote o_j has added T_w's descriptor referencing the over-written version somewhere after T_i's starting point, preventing GC.

The commit procedure for read-only transactions merely removes the pointer to the starting time point, in order to make it GCable, and always commits.

6.2.3 Allowing Concurrent Access to the Time Points List

We show now how to avoid locking the time points list (lines 24, 31 in Algorithm 7), so that update transactions with disjoint write-sets may commit concurrently.

We first explain the reason for using the lock. In order to update the objects in the write-set, the updating transaction has to know the new version number to use. However, if a transaction exposes its descriptor before it finishes updating the write-set, then some read-only transaction might observe an inconsistent state. Consider, for example, transaction T_w that updates objects o_1 and o_2. The value of *curPoint* at the beginning of T_w's commit is 9. Assume T_w first inserts its descriptor with value 10 to the list, then updates object o_1 and pauses. At this point, $o_1.\mathit{version} = 10$, $o_2.\mathit{version} < 10$ and $\mathit{curPoint} \rightarrow \mathit{commitTime} = 10$. If a new read-only transaction starts with time 10, it can successfully read the new value of o_1 and the old value of o_2, because they are both less than or equal to 10. Intuitively, the problem is that the new time point becomes available to the readers as a potential starting time before all the objects of the committing transaction are updated.

To preserve consistency without locking the time points list, we add an additional boolean field *ready* to the descriptor's structure, which becomes *true* only after the committing transaction finishes updating all objects in its write-set. In addition to the global *curPoint* variable referencing the latest time point, we keep a global *readyPoint* variable, which references the latest time point in the *ready prefix* of the list (see Figure 6.1).

When a new read-only transaction starts, its *startTP* variable references *readyPoint*. In the example above, a new transaction T_r begins with a start time equal to 9, because the new time point with value 10 is still not ready. Generally, the use of *readyPoint* guarantees that if a transaction reads an object version written by T_w, then T_w and all its preceding transactions had finished writing their write-sets.

Note, however, that when using ready points we should not violate the real time order — if a read-only transaction T_r starts after T_w terminates, then T_r must have a start time value not less than T_w's commit time. This property might be violated if update transactions become ready in an order that differs from their time points order, thus leaving an unready transaction between ready ones in the list.

We have implemented two approaches to enforce real-time order: 1) An update transaction does not terminate until the ready point reaches its descriptor. A similar approach was previously used by RingSTM [73] and JVSTM [34]. 2) A new read-only transaction notes the time point of the latest terminated transaction and then waits until the *readyPoint* reaches this point before starting. Note that unlike the first alternative, read-only transactions in the second approach are not wait-free.

According to our evaluation, both techniques demonstrate similar results. The waiting period remains negligible as long as the number of transactional threads does not exceed the number of available cores; when the number of threads is two times the number of cores, waiting causes a $10 - 15\%$ throughput degradation (depending on the workload) — this is the cost we pay for maintaining real-time order.

6.3 Implementation and Evaluation

6.3.1 Compared Algorithms

Our evaluation aims to check the aspect of keeping and garbage collecting multiple versions. Direct comparison was difficult because of different frameworks the algorithms are implemented in[1]. We implement the following algorithms:

- **SMV**– The algorithm described in Section 6.2.

[1]DeuceSTM [52] framework comes with TL2 and LSA built-in, however, its LSA implementation is single-versioned.

- **TL2**– Java implementation of TL2 [29] with a single central global version clock. We use a standard optimization of not keeping a read-set for read-only transactions. The code follows the style of TL2 implementation in Deuce framework [52].

- **TL2 with time points**– A variant of TL2, which implements the time points mechanism described in Section 6.2.1. This way, we check the influence of the use of time points on overall performance and separate it from the impact of multi-versioning techniques used in SMV.

- k**-versioned**– an STM based on a TL2-style's logic and code, in which each object keeps a constant k, number of versions (this approach resembles LSA [69]). Reads operate as in SMV, except that if no adequate version is found, the transaction aborts. Updates operate as in TL2.

- **Read-Write lock (RWLock)**– a simple global read-write lock. The lock is acquired at the beginning of an atomic section and is released at its end.

We use the Polite contention manager with exponential backoff [71] for all the algorithms: aborted transactions spin for a period of time proportional to 2^n, where n is the number of retries of the transaction.

It is worth noting that our implementation of the TL2-style algorithm does not use all the software optimizations that might be used in the original one. Our aim is to create for all the algorithms as similar a starting point as possible. This way, the tests evaluate the algorithmic differences, while minimizing the influence of the engineering optimizations and tweaks. These may certainly be applicable to each of the compared alternatives.

6.3.2 Experiment Setup

All algorithms are implemented in Java. We use the following benchmarks for performance evaluation: 1) a red-black tree microbenchmark; 2) the Java version of STMBench7 [44]; and 3) Vacation, which is part of the STAMP [23] benchmark suite.

Red-black tree microbenchmark. The red-black tree supports insertion, deletion, query and range query operations. The initial size of the tree is 400000 nodes. It is checked both for read-dominated workloads ($80/20$ ratio of read-only to update operations) and for workloads with update operations only.

STMBench7. STMBench7 aims to simulate different behaviors of real-world programs by invoking both read-only and update transactions of different lengths over large data structures, typically graphs. Workload types differ in their ratio of read-only to update transactions: $90/10$ for *read-dominated* workloads, $60/40$ for *read-write* workloads, and $10/90$ for *write-dominated* workloads. Operation types for both read-only and update transactions include graph traversals of different lengths, structural modifications, and single access operations. When running STMBench7 workloads, we bound the length of each benchmark by the number of transactions performed by each thread (2000 transactions per thread unless stated otherwise). We manually disabled long update traversals because they inherently eliminate any potential for scalability.

Vacation (Java port). Vacation [23], emulates a travel reservation system, which is implemented as a set of trees. In our experiments it is run with the standard parameters corresponding to `vacation-high++`. Note that STAMP benchmarks are not suitable for evaluating techniques that optimize read-only transactions, because these benchmarks do not have read-only transactions at all. We use Vacation as one exemplary STAMP application to evaluate SMV's overhead.

Setup. In all our benchmarks, we defined transactional objects at the granularity of graph/data structure nodes. This provides a reasonable compromise between the cost of copy-on-write and the overhead of algorithmic bookkeeping. To support this, we re-implemented collections based on `java.util`.

The benchmarks are run on a dedicated shared-memory NUMA server with 8 Quad Core AMD 2.3GHz processors and 16GB of memory attached to each processor. The system runs Linux 2.6.22.5-31 with swap turned off. For all tests but those with limited memory, JVM is run with the `AggressiveHeap` flag on. Thread scheduling is left entirely to the OS. We run up to 64 threads on the 32 cores.

Our evaluation study is organized as follows: in Section 6.3.3, we show system performance measurements. Section 6.3.4 considers the latency and predictability of long read-only operations, and in Section 6.3.5, we analyze the memory demands of the algorithms.

6.3.3 Performance Measurements

SMV overhead. As we mentioned earlier, the use of multiple versions in our algorithm can be exploited by read-only transactions only. However, before evaluating the performance of SMV with read-only transactions, we first want to understand its behavior in programs with update transactions only. In these programs, SMV can hardly be expected to outperform its single-versioned counterparts. For update transactions, SMV resembles the behavior of TL2, with the additional overhead of maintaining previous object versions. Thus, measuring throughput in programs without read-only transactions quantifies the cost of this additional overhead.

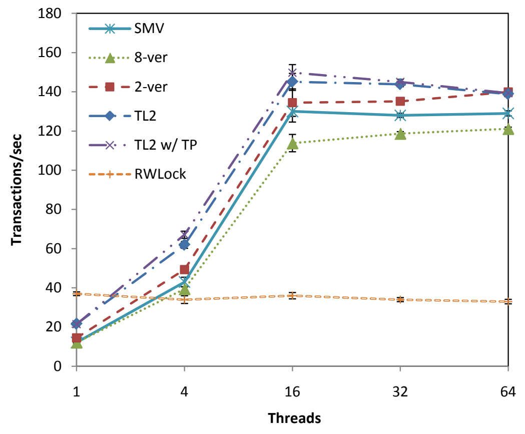
In Figure 6.2, we show throughput measurements for write-dominated benchmarks: Red-black tree (Figure 6.2(a)) and Vacation (Figure 6.2(c)) do not contain read-only transactions at all. The write-dominated STMBench7 workload shown in Figure 6.2(b) runs 90% of its operations as update transactions, and therefore the influence of read-only ones is negligible.

All compared STM algorithms show similar behavior in all three benchmarks. This emphasizes the fact that the algorithms take the same approach when executing update transactions and that they all have a common underlying code platform. The differences in the behavior of RWLock are explained by different contention levels of the benchmarks. While the contention level in Vacation remains moderate even for 64 threads, contention in the write-dominated STMBench7 is extremely high, so that RWLock outperforms the other alternatives.

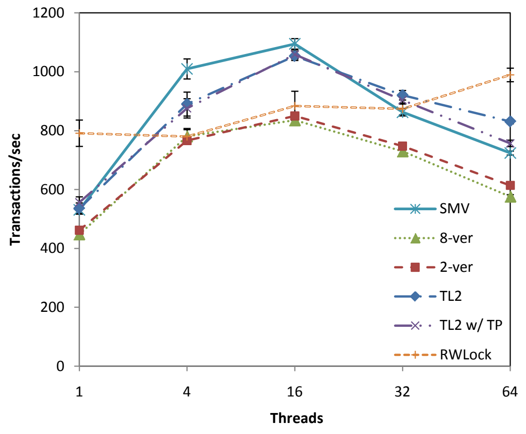
Figure 6.2 demonstrates low overhead of SMV when the number of threads does not exceed 32; for 64 threads this overhead causes a 15% throughput drop. This is the cost we pay for maintaining multiple versions when these versions are not actually used.

Throughput. We next run workloads that include read-only transactions, in order to assess whether the overhead of SMV is offset by its smaller abort rate. In Figure 6.3 we depict throughput measurements of the algorithms in STMBench7's read-dominated and read-write workloads, as well as the throughput of the red-black tree. We see that in the read-dominated STMBench7 workload, SMV's throughput is seven times higher than that of TL2. Despite keeping as many as 8 versions, the k-versioned STM cannot keep up, and SMV outperforms it by more than twice.

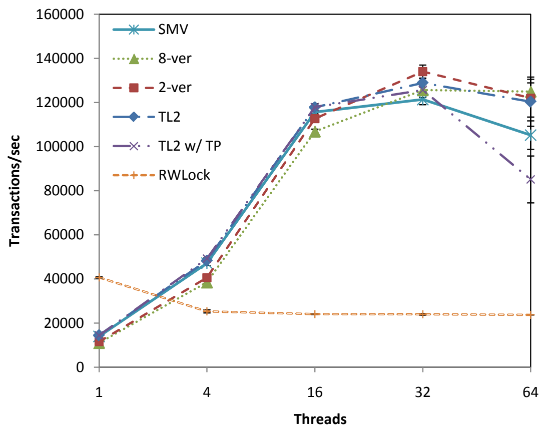
What is the reason for 8 versions not being enough? The results presented in Figure 6.4 give the explanation. In this figure we compare the average probability of accessing an old object version in a read-write workload of STMBench7 with the work that might be lost because some k^{th} version



(a) Throughput in red-black tree write-only workload.



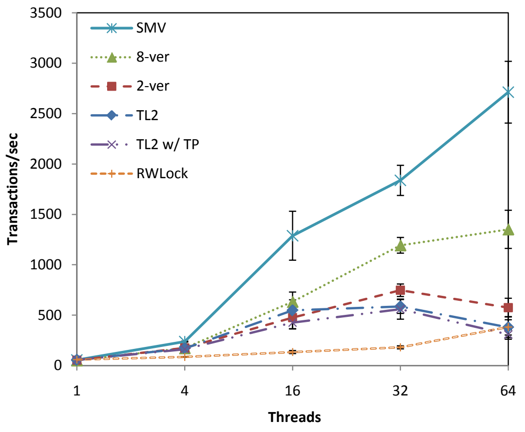
(b) Throughput in STMBench7's write-dominated workload.



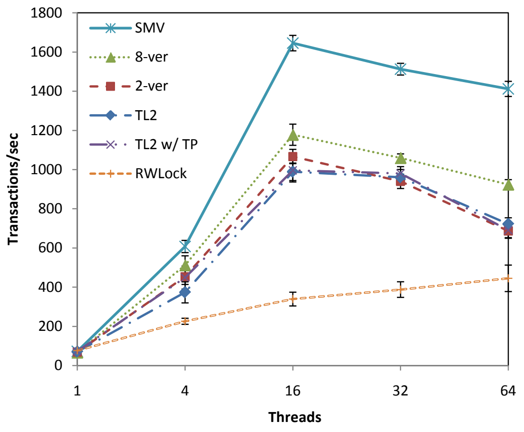
(c) Throughput in Vacation benchmark.

Figure 6.2: In the absence of read-only transactions multi-versioning cannot be exploited. The overhead of SMV degrades throughput by up to 15%.

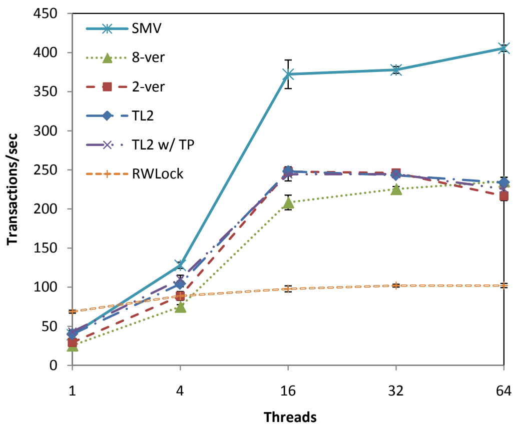
is absent. We can see that the probability of accessing an old version of some random object is extremely small (less than 1.2% even for the second version). Therefore, keeping k versions for *each object* can be wasteful. However, the amount of work lost because the k^{th} version is absent, is surprisingly high even for large k values. Intuitively, this occurs since a transaction that needs to access the k^{th} version of an object must have been running for a long time, and the price of aborting such a transaction is high. Hence, keeping previous versions is important, especially for the frequently updated objects; keeping a constant number of versions per object will typically not be enough for reducing the amount of wasted work.



(a) Throughput in STMBench7's read-dominated workload.



(b) Throughput in STMBench7's read-write workload.

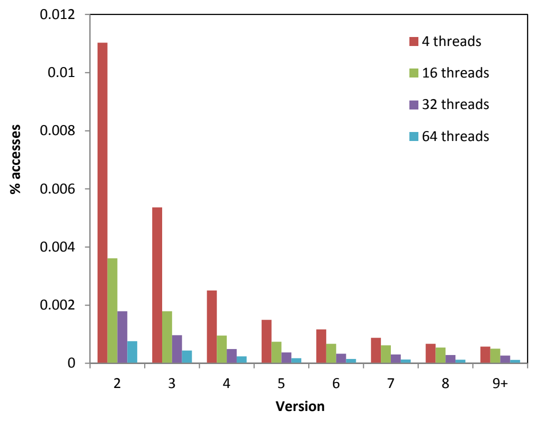


(c) Throughput in the RBTree read-dominated workload.

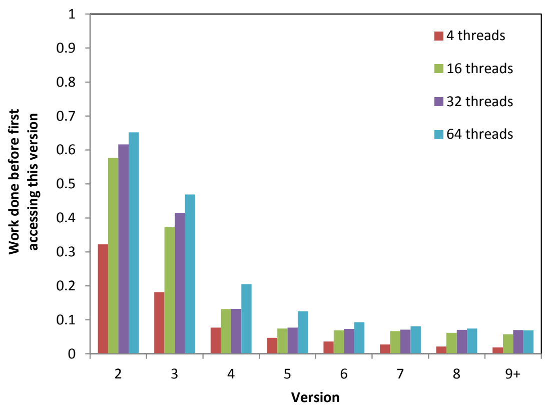
Figure 6.3: By reducing aborts of read-only transactions, SMV presents a substantially higher throughput than TL2 and the k-versioned STM. In read-dominated workloads, its throughput is $\times 7$ higher than that of TL2 and more than twice those of the k-versioned STM with $k = 2$ or $k = 8$. In read-write workloads its advantage decreases because of update transactions, but SMV still clearly outperforms its competitors.

We further note that SMV is scalable, and its advantage over a single-version STM becomes more pronounced as the number of threads rises. In the read-write workload, the number of read-only transactions that can use multiple versions decreases, and the throughput gain becomes 95% over TL2 and 52% over the 8-versioned STM.

We conclude that in the presence of read-only transactions the benefit of SMV significantly



(a) Probability of accessing an old version in a read-write workload.



(b) Work lost because the k^{th} version is absent in a read-write workload.

Figure 6.4: The average probability of accessing an old version is small – keeping old versions for *each object* is wasteful. The amount of work lost because of an absent k^{th} version is high – keeping old versions of "hot spots" is useful.

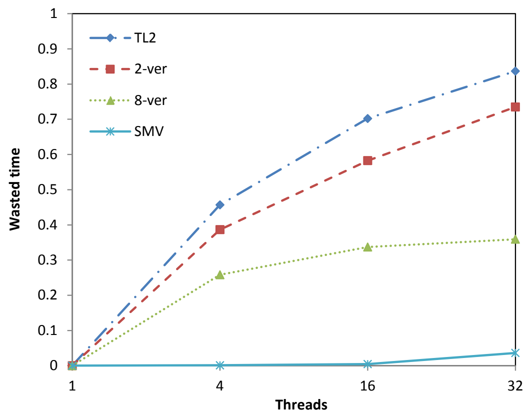


Figure 6.5: Time spent in aborted transactions in STMBench7's read-dominated workload.

outweighs its overhead. This benefit can be explained by looking at the amount of time wasted on eventually aborted transactions (Figure 6.5). This approach approximates net CPU utilization and hence explains throughput results. We note that this approximation works well only if the number of threads is less than or equal to the number of available cores (otherwise, time measurements also count intervals in which the threads are suspended). We see that in the read-dominated workload, TL2 spends more than 80% of its time running aborted transactions! Interestingly, k-versioned STMs cannot fully alleviate this effect either, succeeding to reduce the amount of wasted time to

	Number of threads				
	1	4	8	16	32
TL2	1.3	21.6	68.5	103.6	358.5
SMV	1.3	1.4	2.4	3.6	11.9
2-versioned	1.3	4.1	22.9	45.2	204.5
8-versioned	1.3	6.8	10.6	22.2	79.4

Table 6.1: Maximum time (sec) for completing a long read-only operation in STMBench7 is hardly predictable for TL2 and k-versioned STMs: it might arrive to hundreds of seconds under high loads. SMV presents stable performance unaffected by the level of contention.

36% only. In contrast, SMV's wastage does not rise above 3%.

It is possible to employ timestamp extension [69, 33] to reduce the amount of wasted work in both TL2 and SMV. However, this approach requires read-only transactions to maintain read-sets. The overhead of keeping a read-set is significant for long read-only transactions. We implemented timestamp extension in both TL2 and SMV, and our experiments showed that it did not improve the performance of either algorithm, although it did reduce the amount of wasted work. For space imitations, we omit these results.

6.3.4 Latency and Predictability of Long Read-Only Operations

In the previous section we concentrated on overall system performance without considering specific transactions. However, in real-life applications the completion time of individual operations is important as well. In this section we consider two examples: taking system snapshots of a running application and STMBench7's long traversals.

Taking a full-system snapshot is important in various fields: it is used in client-server finance applications to provide clients with consistent views of the state, for checkpointing in high-performance computing, for creating new replicas, for application monitoring and gathering statistics, etc. Predictability of the time it takes to complete the snapshot is important, both for program stability and for usability.

We first show the maximum time for completing a long read-only traversal, which is already built-in in STMBench7 (see Table 6.1). As we can see from the table, this operation takes only several seconds when run without contention. However, when the number of threads increases, completing the traversal might take more than 100 seconds in TL2 and k-versioned STMs. Un-

	Number of threads				
	1	4	8	16	32
TL2	—	—	—	—	—
SMV	1.4	1.3	1.2	1.4	1.5
2-versioned	—	—	—	—	—
8-versioned	—	—	—	—	—

Table 6.2: Maximum time (sec) to take a snapshot in Vacation benchmark. Vacation snapshot operation run by TL2 or k-versioned algorithms cannot terminate even when there is only a single application thread, while SMV presents stable performance.

like those algorithms, SMV is less impacted by the level of contention and it always succeeds to complete the traversal in several seconds.

Next, we added the option of taking a system snapshot in Vacation. In addition to the original application threads, we run a special thread that repeatedly tries to take a snapshot. We are interested in the maximum time it takes to complete the snapshot operation. The results appear in Table 6.2. We see that neither TL2 nor the k-versioned STM can successfully take a snapshot even when only a single application thread runs updates in parallel with the snapshot operation. Surprisingly, even 8 versions do not suffice to allow snapshots to complete, this is because within the one and a half seconds it takes the snapshot to complete some objects are overwritten more than 8 times.

On the other hand, the performance of SMV remains stable and unaffected by the number of application threads in the system. We conclude that SMV successfully keeps the needed versions. In Section 6.3.5, we show that it does so with smaller memory requirements than the k-versioned STM.

We would like to note that while taking a snapshot is also possible by pausing mutator threads, this approach is much less efficient, as it requires quiescence periods and thus reduces the overall throughput.

6.3.5 Memory Demands

One of the potential issues with multi-versioned STMs is their high memory consumption. In this section we compare memory demands of the different algorithms. To this end, we execute long-running write-dominated STMBench7 benchmarks (64 threads, each thread running 40000

	Memory limit				
	2GB	4GB	8GB	12GB	16GB
TL2	606.89	631.56	630.3	674.96	647.17
SMV	450.12	543.04	563.74	595.78	602.01
2-versioned	—	515.32	532.7	550.61	533.01
4-versioned	—	—	—	—	281.98
8-versioned	—	—	—	—	—

Table 6.3: Throughput (txn/sec) in limited memory systems: k-versioned STMs do not succeed to complete the benchmark.

operations) with different limitations on the Java memory heap. Such runs present a challenge for the multi-versioned STMs because of their high update rate and limited memory resources. As we recall from Section 6.3.3, multi-versioned STMs cannot outperform TL2 in a write-dominated workload. Hence, the goal of the current experiment is to study the impact of the limited memory availability on the algorithms' behaviors.

Figure 6.3 shows how the algorithms' throughput depends on the Java heap size. A "—" sign corresponds to runs in which the algorithm did not succeed to complete the benchmark due to a Java `OutOfMemoryException`. Notice that the 8-versioned STM is unable to successfully complete a run even given a 16GB Java heap size. Decreasing k to 4, and then 2, makes it possible to finish the runs under stricter constraints. However, none of the k-versioned STMs succeed under the limitation of 2GB. Unlike k-versioned STMs, SMV continues to function under these constraints. Furthermore, SMV's throughput does not change drastically — the maximum decrease is 25% when Java heap size shrinks 8-fold.

The collapse of the k-versioned STM confirms the observation from Section 5.3.1, where we have illustrated that its memory consumption can become exponential rather than linear in the number of transactional objects.

In Table 6.4 we show the relative amount of time spent garbage collecting unreferenced data (we use a standard throughput "stop the world" garbage collector, hence, the application threads do not run during this time). As we mentioned above, k-versioned STMs were simply unable to complete the benchmark. As expected, the GC share when running SMV is clearly higher than that of TL2 — 8% versus 2% in the less constrained runs. Under the stringent constraints, the GC takes a significant share of the time (30%). This is the cost we pay for collecting old object versions in a sophisticated way: transactional descriptors might form long chains of time points that complicate

	Memory limit				
	2GB	4GB	8GB	12GB	16GB
TL2	0.10	0.04	0.03	0.02	0.02
SMV	0.30	0.13	0.09	0.08	0.08
2-versioned	—	0.06	0.03	0.03	0.02
4-versioned	—	—	—	—	0.46
8-versioned	—	—	—	—	—

Table 6.4: Time spent in GC: k-versioned STMs do not succeed to complete the benchmark, SMV spends $\approx 10\%$ of the time in GC when the memory limitation is above 2GB.

the task of the garbage collector.

Chapter 7

SALSA: Scalable and Low Synchronization NUMA-aware Algorithm for Producer-Consumer Pools

This chapter presents a highly-scalable producer-consumer task pool, designed with a special emphasis on lightweight synchronization and data locality. The core building block of our pool is *SALSA, Scalable And Low Synchronization Algorithm* for a single-consumer container with task stealing support. Each consumer operates on its own SALSA container, stealing tasks from other containers if necessary. We implement an elegant self-tuning policy for task insertion, which does not push tasks to overloaded SALSA containers, thus decreasing the likelihood of stealing.

SALSA manages large chunks of tasks, which improves locality and facilitates stealing. SALSA uses a novel approach for coordination among consumers, without strong atomic operations or memory barriers in the fast path. It invokes only two CAS operations during a chunk steal.

Our evaluation demonstrates that a pool built using SALSA containers scales *linearly* with the number of threads and significantly outperforms other FIFO and non-FIFO alternatives.

The task pool presented in this chapter lacks the ability of identifying the emptiness of a container: a task retrieval is blocked as long as no task can be found. The extension of the work described in this chapter, which presents the non-blocking linearizable task pool, appears in proceedings of the 24th ACM Symposium on Parallelism and Architectures (SPAA'12) and is fully described in the technical report [38].

7.1 Introduction

The producer-consumer pool is a fundamental data structure consisting of an unordered collection of objects. Pools have a number of important applications in multiprocessor computing, e.g., transferring tasks in a parallel computation. It is thus highly important to ensure that such a pool does not become a bottleneck when concurrently accessed by large number of threads.

One of the common approaches to implement a producer-consumer pool is using FIFO/LIFO queues. However, this approach inherently suffers from poor scalability and high synchronization costs [3, 14, 74]. In addition, FIFO/LIFO properties of the queues *cannot be used in practice* if multiple consumers work on the same queue simultaneously. This happens because every consumer can be suspended by the OS scheduler for an unbounded period of time after retrieving a task. This way, a task can be "bypassed" by an arbitrary number of later tasks before actually being consumed. Hence, even if a multi-consumer queue guarantees an order on task retrieval, no simple way exists to exploit such an order.

This chapter presents a non-FIFO scalable and highly-efficient task pool, with lightweight synchronization-free operations in the common case. Its data allocation scheme is cache-friendly and highly suitable for NUMA environments. Moreover, our pool is robust in the face of imbalanced loads and unexpected thread stalls.

Our system is composed of two independent logical entities: 1) *SALSA, Scalable and Low Synchronization Algorithm*, a single-consumer pool that exports a stealing operation, and 2) a work stealing framework implementing a management policy that operates multiple SALSA pools.

In order to improve locality and facilitate stealing, SALSA keeps tasks in chunks, organized in per-producer chunk lists. Only the producer mapped to a given list can insert tasks to chunks in this list, which eliminates the need for synchronization among producers.

Though each consumer has its own task pool, inter-consumer synchronization is required in order to allow stealing. The challenge is to do so without resorting to costly atomic operations (such as CAS or memory fences) upon each task retrieval. We address this challenge via a novel chunk-based stealing algorithm that allows consume operations to be synchronization-free in the common case, when no stealing occurs, which we call the *fast path*. Moreover, SALSA reduces the stealing rate by moving entire chunks of tasks in one steal operation, which requires only two CAS operations.

In order to achieve locality of memory access on a NUMA architecture, SALSA chunks are kept in the consumer's local memory. The management policy matches producers and consumers according to their proximity, which allows most task transfers to occur within a NUMA node.

In many-core machines running multiple applications, system behavior becomes less predictable. Unexpected thread stalls may lead to an asymmetric load on consumers, which may in turn lead to high stealing rates, hampering performance. SALSA employs a novel auto-balancing mechanism that has producers insert tasks to less loaded consumers, and is thus robust to spurious load fluctuations.

We have implemented SALSA in C++, and tested its performance on a 32-core NUMA machine. Our experiments show that the SALSA-based work stealing pool *scales linearly* with the number of threads; it is 20 times faster than other work-stealing alternatives, and shows a significant improvement over state-of-the-art non-FIFO alternatives. SALSA-based pools scale well even in unbalanced scenarios.

The rest of this chapter proceeds as follows. Section 7.2 describes the system overview. The SALSA single-consumer algorithm is described in Section 7.3, we then discuss our implementation and experimental results in Section 7.4.

7.2 System Overview

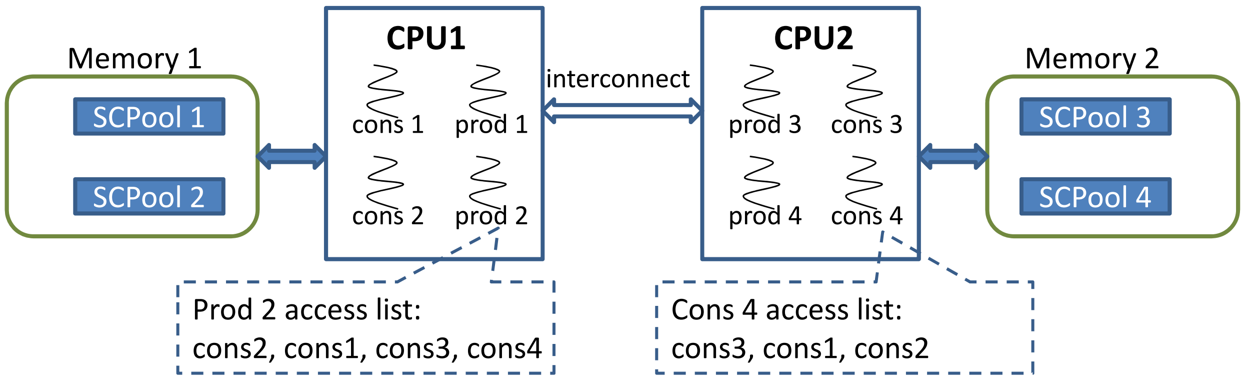


Figure 7.1: Producer-consumer framework overview. In this example, there are two processors connected to two memory banks (NUMA architecture). Two producers and two consumers running on each processor, and the data of each consumer is allocated at the closest physical memory. A producer (consumer) has a sorted access list of consumers for task insertion (respectively stealing).

In the current section we present our framework for scalable and NUMA-aware producer-consumer data exchange. Our system follows the principle of separating mechanism and policy. We therefore consider two independent logical entities:

1. *A single consumer pool (SCPool)* mechanism manages the tasks arriving to a given consumer and allows tasks stealing by other consumers.

2. A management policy operates SCPools: it routes producer requests to the appropriate consumers and initiates stealing between the pools. This way, the policy controls the system's behavior according to considerations of load-distribution, throughput, fairness, locality, etc. We are especially interested in a management policy suitable for NUMA architectures (see Figure 7.1), where each CPU has its own memory, and memories of other CPUs are accessed over an interconnect. As a high rate of remote memory accesses can decrease the performance, it is desirable for the SCPool of a consumer to reside close to its own CPU.

Algorithm 9 API for a Single Consumer Pool with stealing support.

```
1: boolean: produce(Task, SCPool)    ▷ Tries to insert the task to the pool, returns false if no space is available.
2: void: produceForce(Task, SCPool)    ▷ Insert the task to the pool, expanding the pool if necessary.
3: {Task ∪⊥}: consume()    ▷ Retrieve a task from the pool, returns ⊥ if no tasks in the pool are detected.
4: {Task ∪⊥}: steal(SCPool from)    ▷ Try to steal a number of tasks from the given pool and move them to the current pool. Return some stolen task or ⊥.
```

SCPool abstraction. The SCPool API provides the abstraction of a single consumer task pool with stealing support, see Algorithm 9. A producer invokes two operations: **produce()**, which attempts to insert a task to the given pool and fails if the pool is full, and **produceForce()**, which always succeeds by expanding the pool on demand. There are also two ways to retrieve a task from the pool: the owner of the pool (only) can call the **consume()** function; while any other thread can invoke **steal()**, which tries to transfer a number of tasks between two pools and return one of the stolen tasks.

A straightforward way to implement the above API is using dynamic-size multi-producer multi-consumer FIFO queue (e.g., Michael-Scott queue [58]). In this case, **produce()** enqueues a new task, while **consume()** and **steal()** dequeue a task. In the next section we present SALSA, a much more efficient SCPool.

Algorithm 10 Work stealing framework pseudo-code.

```
 5: Local variables:
 6:     SCPool myPool ▷ The consumer's pool
 7:     SCPool[] accessList ▷ The consumer's or producer's access list
 8: Function get():
 9:     while(true)
10:         ▷ First try to get a task from the local pool
11:         t ← myPool.consume()
12:         if (t $\neq \perp$) return t
13:         ▷ Failed to get a task from the local pool – steal
14:         foreach SCPool p in accessList in order do:
15:             t ← p.steal()
16:             if (t $\neq \perp$) return t
            ▷ No tasks found – start over again
17: Function put(Task t):
18:     ▷ Produce to the pools by the order of the access list
19:     foreach SCPool p in accessList in order do:
20:         if (p.produce(t)) return
21:     firstp ← the first entry in accessList
22:     ▷ If all pools are full, expand the closest pool
23:     produceForce(t,firstp)
24:     return
```

Management policy. A management policy defines the way in which: 1) a producer chooses an SCPool for task insertion; and 2) a consumer decides when to retrieve a task from its own pool or steal from other pools. Note that the policy is independent of the underlying SCPool implementation. We believe that the policy is a subject for engineering optimizations, based on specific workloads and demands.

In the current work, we present a NUMA-aware policy. If the individual SCPools themselves are lock-free, then our policy preserves lock-freedom at the system level. Our policy is as follows:

- **Access lists.** Each process in the system (producer or consumer) is provided with an *access list*, an ordered list of all the consumers in the system, sorted according to their distance from that process (see Figure 7.1). Intuitively, our intention is to have a producer mostly interact with the closest consumer, while stealing mainly happens inside the same processor node.

- **Producer's policy.** The producer policy is implemented in the **put**() function in Algorithm 10. The operation first calls the **produce()** of the first SCPool in its access list. Note that this operation might fail if the pool is full, (which can be seen as evidence of that the corresponding consumer is overloaded). In this case, the producer tries to insert the task into other pools, in the order defined by its access list. If all insertions fail, the producer invokes **produceForce()** on the closest SCPool, which always succeeds (expanding the pool if needed).

- **Consumer's policy.** The consumer policy is implemented in the **get**() function in Algorithm 10. A consumer takes tasks from its own SCPool. If its SCPool is empty, then the consumer tries to steal tasks from other pools in the order defined by its access list. Stealing serves two purposes: first, it is important for distributing the load among all available consumers. Second, it ensures that tasks are not lost in case they are inserted into the SCPool of a crashed (or very slow) consumer. Checking emptiness of a container is a subtle issue and we do not handle it in the current thesis (**get**() operation is blocked as long as it cannot find a task to retrieve). Emptiness detection is rigorously described in the full version of this work [38], which devises a non-blocking linearizable task pool algorithm.

7.3 Algorithm Description

In the current section we present the SALSA SCPool. We first show the data structures of SALSA in Section 7.3.1, and then present the basic algorithm without stealing support in Section 7.3.2. The stealing procedure is described in Section 7.3.3, finally, the role of chunk pools is presented in Section 7.3.4. For the simplicity of presentation, in this section we assume that the the memory accesses satisfy sequential consistency [55], we describe the ways to solve memory reordering issues in Section 7.4.1.

7.3.1 SALSA Structure

The SALSA data structure of a consumer c_i is described in Algorithm 11 and partially depicted in Figure 7.2. The tasks inserted to SALSA are kept in chunks, which are organized in per-producer chunk lists. Only the producer mapped to a given list can insert a task to any chunk in that list. Every chunk is owned by a single consumer whose id is kept in the *owner* field of the chunk. The owner is the only process that is allowed to take tasks from the chunk; if another process wants to

Algorithm 11 SALSA implementation of SCPool: Data Structures.

```
25: Chunk type
26:     Task[CHUNK_SIZE] tasks
27:     int owner ▷ owner's consumer id
28: Node type
29:     Chunk c; initially ⊥
30:     int idx; initially -1
31:     Node next;
32: SALSA per consumer data structure:
33:     int consumerId
34:     List⟨Node⟩[] chunkLists ▷ one list per producer + extra list for stealing (every list is single-writer multi-
        reader)
35:     Queue⟨Chunk⟩ chunkPool ▷ pool of spare chunks
36:     Node currentNode, initially ⊥ ▷ current node to work with
```

take a task from the chunk, it should first steal the chunk and change its ownership. A task entry in a chunk is used at most once. Its value is ⊥ before the task is inserted, and TAKEN after it has been consumed.

The per-producer chunk lists are kept in the array *chunkLists* (see Figure 7.2), where *chunkLists[j]* keeps a list of chunks with tasks inserted by producer p_j. In addition, the array has a special entry *chunkLists[steal]*, holding chunks stolen by c_i. Every list has a single writer who can modify the list structure (add or remove nodes): *chunkLists[j]*'s modifier is the producer p_j, while *chunkLists[steal]*'s modifer is the SCPool's owner. The nodes of the used chunks are lazily reclaimed and removed by the list's owner. For brevity, we omit the linked list manipulation functions from the pseudo-code bellow. Our single-writer lists can be implemented without synchronization primitives, similarly to the single-writer linked-list in [57]. In addition to holding the chunk, a node keeps the index of the latest taken task in that chunk, this index is then used for chunk stealing as we show in Section 7.3.3.

Safe memory reclamation is provided by using hazard pointers [57] both for nodes and for chunks. The free (reclaimed) chunks in SALSA are kept at per-consumer *chunkPools* implemented by lock-free Michael-Scott queues [58]. As we show in Section 7.3.4, the chunk pools serve two purposes: 1) efficient memory reuse and 2) producer-based load balancing.

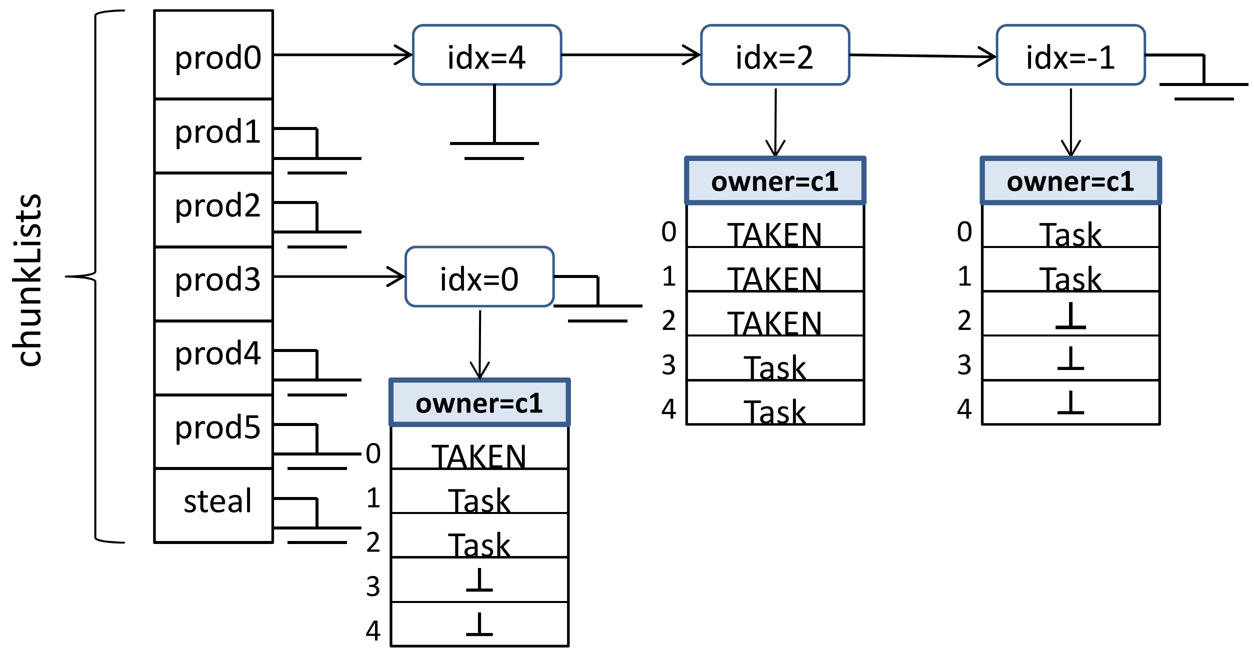


Figure 7.2: Chunk lists in SALSA single consumer pool implementation. Tasks are kept in chunks, which are organized in per-producer lists; an additional list is reserved for stealing. Each list can be modified by the corresponding producer only. The only process that is allowed to retrieve tasks from a chunk is the owner of that chunk (defined by the ownership flag). A Node's index corresponds to the latest task taken from the chunk or the task that is about to be taken by the current chunk owner.

7.3.2 Basic Algorithm

SALSA producer

The description of SALSA producer functions is presented in Algorithm 12. The insertion of a new task consists of two stages: 1) finding a chunk for task insertion (if necessary), and 2) adding a task to the chunk.

Finding a chunk The chunk for task insertions is kept in the local producer variable *chunk* (line 39 in Algorithm 12). Once a producer starts working with a chunk c, it continues inserting tasks to c until c is full – the producer is oblivious to chunk stealing. If the *chunk*'s value is $\perp$, then the producer should start a new chunk (function *getChunk*). In this case, it tries to retrieve a chunk from the chunk pool and to append it to the appropriate chunk list. If the chunk pool is empty then the producer either returns $\perp$ (if force=false), or allocates a new chunk by itself (otherwise)

Algorithm 12 SALSA implementation of SCPool: Producer Functions.

```
37: Producer local variables:
38:    int producerId
39:    Chunk chunk; initially $\perp$ ▷ the chunk to insert to
40:    int prodIdx; initially $0$ ▷ the prefix of inserted tasks
41: Function produce(Task t):
42:    return insert(t, this, false)
43: Function insert(Task t, SCPool scPool, bool force):
44:    if (chunk $= \perp$) then ▷ allocate new chunk
45:       if (getChunk(scPool, force) $=$ false) then return false
46:    chunk.tasks[prodIdx] $\leftarrow$ t; prodIdx++
47:    if(prodIdx $=$ CHUNK_SIZE) then
48:       chunk $\leftarrow \perp$ ▷ the chunk is full
49:    return true
50: Function produceForce(Task t):
51:    insert(t, this, true)
52: Function getChunk(SALSA scPool, bool force)
53:    newChunk $\leftarrow$ dequeue chunk from scPool.chunkPool
54:    if (chunk $= \perp$) ▷ no available chunks in this pool
55:       if (force $=$ false) then return false
56:       newChunk $\leftarrow$ allocate a new chunk
57:    newChunk.owner $\leftarrow$ scPool.consumerId
58:    node $\leftarrow$ new node with idx $= -1$ and c $=$ newChunk
59:    scPool.chunkLists[producerId].append(node)
60:    chunk $\leftarrow$ newChunk; prodIdx $\leftarrow 0$
61:    return true
```

(lines 54–56).

Inserting a task to the chunk As previously described in Section 7.3.1, different producers insert tasks to different chunks, which removes the need for synchronization among producers. The producer local variable *prodIdx* indicates the next free slot in the chunk. All that is left for the insertion function to do, is to put a task in that slot and to increment *prodIdx* (line 46). Once the index reaches the maximal value, the *chunk* variable is set to $\perp$, indicating that the next insertion operation should start a new chunk.

SALSA consumer without stealing

Algorithm 13 SALSA implementation of SCPool: Consumer Functions.

```
Function consume():
    if (currentNode $\neq \perp$) then ▷ common case
        t $\leftarrow$ takeTask(currentNode)
        if (t $\neq \perp$) then return t
    foreach Node $n$ in ChunkLists do: ▷ fair traversal of chunkLists
        if (n.c $\neq \perp \wedge$ n.c.owner $=$ consumerId) then
            t $\leftarrow$ takeTask(n)
            if (t $\neq \perp$) then currentNode $\leftarrow$ n; return t
    currentNode $\leftarrow \perp$; return $\perp$
Function takeTask(Node n):
    chunk $\leftarrow$ n.c
    if (chunk $= \perp$) then return $\perp$ ▷ this chunk has been stolen
    task $\leftarrow$ chunk.tasks[n.idx + 1]
    if (task $= \perp$) then return $\perp$ ▷ no inserted tasks
    if (chunk.owner $\neq$ consumerId)
        return $\perp$
    n.idx++ ▷ tell the world you're going to take a task from idx
    if (chunk.owner $=$ consumerId) then ▷ common case
        chunk.tasks[n.idx] $\leftarrow$ TAKEN
        checkLast(n)
        return task
    ▷ the chunk has been stolen, CAS the last task and go away
    success $\leftarrow$ (task $\neq$ TAKEN $\wedge$
         CAS(chunk.tasks[n.idx], task, TAKEN))
    if(success) then checkLast(n)
    currentNode $\leftarrow \perp$
    return (success) ? task : $\perp$
Function checkLast(Node n):
    if(n.idx + 1 $=$ CHUNK_SIZE) then ▷ finished the chunk
        n.c $\leftarrow \perp$; return chunk to chunkPool
        currentNode $\leftarrow \perp$
```

The consumer's algorithm without stealing is given in the Algorithm 13. The consumer first finds a nonempty chunk it owns and then invokes **takeTask()** to retrieve a task.

Unlike producers, which have exclusive access to insertions in a given chunk, a consumer must take into account the possibility of stealing. Therefore, it should notify other processes which task it is about to take.

To this end, each node in the chunk list keeps an index of the taken prefix of its chunk in the *idx* variable, which is initiated to -1. A consumer that wants to take a task T, first increments the index, then checks the chunk's ownership, and finally changes the chunk entry from T to *TAKEN* (lines 78–80). By doing so, a consumer guarantees that *idx* always points to the last taken task or to a task that is about to be taken. Hence, a process that is stealing a chunk from a node with *idx* $= i$ can assume that the tasks in the range $[0 \ldots i)$ have already been taken. The logic for dealing with stolen chunks is described in the next section.

7.3.3 Stealing

The stealing algorithm is given in the function **steal()** in Algorithm 14. We refer to the stealing consumer as c_s, the victim process whose chunk is being stolen as c_v, and the stolen chunk as C.

The idea is to turn c_s to be the exclusive owner of C, so that c_s will be able to take tasks from the chunk without synchronization. In order to do that, c_s first adds the chunk to its list (line 97), then changes the ownership of C from c_v to c_s using CAS (line 98), and then removes the chunk from c_v's list (line 114). Once c_v notices the change in the ownership it can take at most one more task from C (lines 83–86) after failing the second check of ownership in line 79 having passed the one in line 76.

When the **steal()** operation of c_s occurs simultaneously with the **takeTask()** operation of c_v, both c_s and c_v might try to retrieve the same task. We now explain why this might happen. Recall that c_v notifies potential stealers of the task it is about to take by incrementing the *idx* value in C's node (line 78). This value is copied by c_s in line 111 when creating a copy of C's node for its steal list.

Consider, for example, a scenario in which the idx is incremented by c_v from 10 to 11. If c_v checks C's ownership before it is changed by c_s, then c_v takes the task at index 11 *without synchronization* (line 80). Therefore, c_s cannot be allowed to take the task pointed by *idx* at all. Hence, c_v has to take the task at index 11 even if it does observe the ownership change. After stealing the chunk, c_s will eventually try to take the task pointed by $idx + 1$. However, if c_s copies the node before idx is incremented by c_v, c_s might think that the value of $idx + 1$ is 11. In this case, both c_s and c_v will try to retrieve the task at index 11. To ensure that the task is not retrieved

Algorithm 14 SALSA implementation of SCPool: the stealing function.

```
91:  Function steal(SCPool p):
92:    prevNode ← a node holding tasks, whose owner is $p$, from some list in $p$'s pool ▷ different policies possible
93:    if (prevNode = ⊥) return ⊥ ▷ No Chunk found
94:    c ← prevNode.c; if (c = ⊥) then return ⊥
95:    prevIdx ← prevNode.idx
96:    if (prevIdx+1 = CHUNK_SIZE ∨ c.tasks[prevIdx+1] = ⊥) return ⊥
       ▷ make it stealable from my list
97:    chunkLists[steal].append(prevNode)
98:    if (CAS(c.owner, p.consumerId, consumerId) = false)
99:      chunkLists[steal].remove(prevNode)
100:     return ⊥ ▷ failed to steal
101:   idx ← prevNode.idx
102:   if (idx+1 = CHUNK_SIZE) ▷ Chunk is empty
103:     chunkLists[steal].remove(prevNode)
104:     return ⊥
105:   task ← c.tasks[idx+1]
106:   if (task ≠ ⊥) ▷ Found task to take
107:     if (c.owner ≠ consumerId ∧ idx ≠ prevIdx)
108:       chunkLists[steal].remove(prevNode)
109:       return ⊥
110:     idx++
111:   newNode ← copy of prevNode
112:   newNode.idx = idx
113:   replace prevNode with newNode in chunkLists[steal]
114:   prevNode.c ← ⊥ ▷ remove chunk from consumer's list
       ▷ done stealing the chunk, take one task from it
115:   if (task = ⊥) then return ⊥ ▷ still no task at idx
116:   if (task = TAKEN ∨ !CAS(c.tasks[idx], task, TAKEN)) then
117:     task ← ⊥
118:   checkLast(newNode)
119:   if (c.owner = consumerId) currentNode ← newNode
120:   return task
```

twice, both functions invoke CAS in order to retrieve this task (line 116 for c_s, line 83 for c_v).

The above schematic algorithm works correctly as long as the stealing consumer can observe the node with the updated index value. This might not be the case in case the same chunk is concurrently stolen by another consumer, rendering the *idx* of the original node obsolete. In order to prevent this situation, stealing a chunk from the pool of consumer c_v is allowed only if c_v is the owner of this chunk (line 98). This approach is prone to the ABA problem: consider a scenario where consumer c_a is trying to steal from c_b, but before the execution of the CAS in line 98, the chunk is stolen by c_c and then stolen back by c_b. In this case, c_a's CAS succeeds but c_a has an old value of idx. To prevent this ABA problem, the owner field contains a tag, which is incremented on every CAS operation. For brevity, tags are omitted from the pseudo-code.

A naïve way for c_s to steal the chunk from c_v would be first to change the ownership and then to move the chunk to the steal list. However, this approach may cause the chunk to disappear when c_s stalls, because the chunk is not yet accessible via the lists of c_s and yet c_s is its owner. Therefore, SALSA first adds the original node to the steal list of c_s, then changes the ownership, and only then replaces the original node with a new one (lines 97–114).

Another issue we need to address is making sure that the idx value in nodes pointing to a given chunk increases monotonically. To this end, we make sure that when c_s creates a new node, this node's idx is greater than or equal to the idx of c_v's node. As noted before, c_v may increase the idx at most once after its chunk is stolen. Also, thanks to the ownerships checks that are done after the task was read and before the idx is incremented, we know that the idx field of c_v increases only if there is a task in the next slot after the ownership change. To ensure that idx does not decrease in this case, c_s sets the idx of the new node to be the idx of c_v plus one if the next task is not $\perp$ (line 110).

7.3.4 Chunk Pools

As described in Section 7.3.1, each consumer keeps a pool of free chunks. When a producer needs a new chunk for adding a task to consumer c_i, it tries to get a chunk from c_i's chunk pool – if no free chunks are available, the **produce()** operation fails.

As described in Section 7.2, our system-wide policy defines that if an insertion operation fails, then the producer tries to insert a task to other pools. Thus, the producer avoids adding tasks to overloaded consumers, which in turn decreases the amount of costly steal operations. We further refer to this technique as producer-based balancing.

Another SALSA property is that a chunk is returned to the pool of a consumer that retrieves the latest task of this chunk. Therefore, the size of the chunk pool of consumer c_i is proportional to the rate of c_i's task consumption. This property is especially appealing for heterogeneous systems – a faster consumer c_i, (e.g., one running on a stronger or less loaded core), will have a larger chunk pool, and so more **produce()** operations will insert tasks to c_i, automatically balancing the overall system load.

7.4 Implementation and Evaluation

In this section we evaluate the performance of our work-stealing framework built on SALSA pools. We first present the implementation details on dealing with memory reordering issues in Section 7.4.1. The experiment setup is described in Section 7.4.2, we show the overall system performance in Section 7.4.3, study the influence of various SALSA techniques in Section 7.4.4 and check the impact of memory placement and thread scheduling in Section 7.4.5.

7.4.1 Dealing with Memory Reordering

The presentation of the SALSA algorithm in Section 7.3 assumes sequential consistency [55] as the memory model. However, most existing systems relax sequential consistency to achieve better performance. Specifically, according to x86-TSO [72], memory loads can be reordered with respect to older stores to different locations. As shown by Attiya et al. [7], it is impossible to avoid both RAW and AWAR in work stealing structures, which requires using a synchronization operation, such as a fence or CAS, to ensure correctness. In SALSA, this reordering can cause an index increment to occur after the ownership validation (lines 78, 79 in Algorithm 13), which violates correctness as it may cause the same task to be taken twice, by both the original consumer and the stealing thread.

The conventional way to ensure a correct execution in such cases is to use memory fences to force a specific memory ordering. For example, adding an `mfence` instruction between lines 78 and 79 guarantees SALSA's correctness. However, memory fences are costly and their use in the common path degrades performance. Therefore, we prefer to employ a synchronization technique that does not add substantial overhead to the frequently used **takeTask()** operation. One example for such a technique is location-based memory fences, recently proposed by Ladan-Mozes et al. [54], which is unfortunately not implemented in current hardware.

In our implementation, we adopt the synchronization technique described by Dice et al. [26], where the slow thread (namely, the stealer) binds directly to the processor on which the fast thread (namely, the consumer) is currently running, preempting it from the processor, and then returns to run on its own processor. Thread displacement serves as a full memory fence, hence, a stealer that invokes the displacement binding right after updating the ownership (before line 101 in Algorithm 13) observes the updated consumer's index. On the other hand, the steal-free fast path is not affected by this change.

7.4.2 Experiment Setup

We compare the following task pool implementations:

- **SALSA** – our work-stealing framework with SCPools implemented by SALSA.

- **SALSA+CAS** – our work-stealing framework with SCPools implemented by a simplistic SALSA variation, in which every **consume()** and **steal()** operation tries to take a single task using CAS. In essence, SALSA+CAS removes the effects of SALSA's low-synchronization fast-path and per-chunk stealing. Note that disabling per-chunk stealing in SALSA annuls the idea of chunk ownership, hence, disables its low-synchronization fast-path as well.

- **ConcBag** – an algorithm similar to the lock-free Concurrent Bags algorithm [74]. It is worth noting that the original algorithm was optimized for the scenario where the same process is both a producer and a consumer (in essence producing tasks to itself), which we do not consider in this work; in our system no thread acts as both a producer and a consumer, therefore every consume operation steals a task from some producer. We did not have access to the original code, and therefore reimplemented the algorithm in our framework. Our implementation is faithful to the algorithm in the paper, except in using a simpler and faster underlined linked list algorithm. All engineering decisions were made to maximize performance.

- **WS-MSQ** – our work-stealing framework with SCPools implemented by Michael-Scott non-blocking queue [58]. Both **consume()** and **steal()** operations invoke the **dequeue()** function.

- **WS-LIFO** – our work-stealing framework with SCPool implemented by Michael's LIFO stack [57].

We did not experiment with additional FIFO and LIFO queue implementations, because, as shown in [74], their performance is of the same order of magnitude as the Michael-Scott queue. Similarly, we did not evaluate CAFÉ [14] pools because their performance is similar to that of WS-MSQ [12], or ED-Pools [3], which have been shown to scale poorly in multi-processor architectures [12, 74].

All the pools are implemented in C++ and compiled with `-O2` optimization level. In order to minimize scalability issues related to allocations, we use `jemalloc` allocator, which has been shown to be highly scalable in multi-threaded environments [2]. Chunks of SALSA and SALSA+CAS contain 1000 tasks, and chunks of ConcBag contain 128 tasks, which were the respective optimal values for each algorithm.

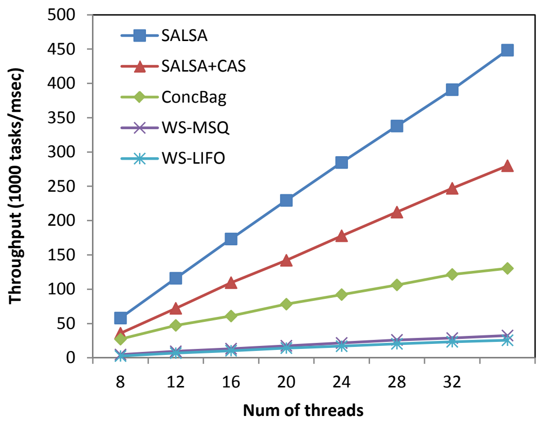
We use a synthetic benchmark where 1) each producer works in a loop of inserting dummy items; 2) each consumer works in a loop of retrieving dummy items. Each data point shown is an average of 5 runs, each with a duration of 20 seconds. The tests are run on a dedicated shared memory NUMA server with 8 Quad Core AMD 2.3GHz processors and 16GB of memory attached to each processor.

7.4.3 System Throughput

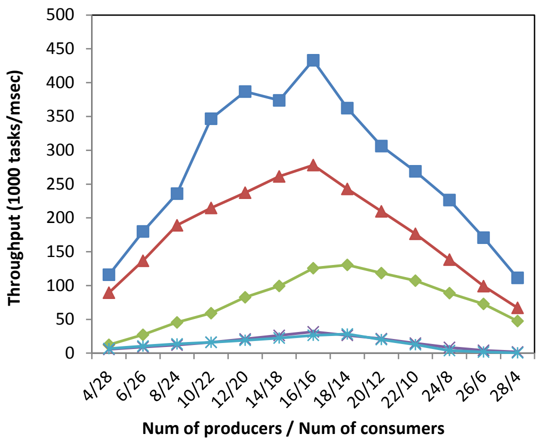
Figure 7.3(a) shows system throughput for workloads with equal number of producers and consumers. SALSA *scales linearly* as the number of threads grows to 32 (the number of physical cores in the system), and it clearly outperforms all other competitors. In the $16/16$ workload, SALSA is $\times 20$ faster than WS-MSQ and WS-LIFO, and more than $\times 3.5$ faster than Concurrent Bags.

We note that the performance trend of ConcBags in our measurements differs from the results presented by Sundell et al. [74]. While in the original paper, their throughput *drops* by a factor of 3 when the number of threads increases from 4 to 24, in our tests, the performance of ConcBags *increases* with the number of threads. The reasons for the better scalability of our implementation can be related to the use of different memory allocators, hardware architectures, and engineering optimizations.

All systems implemented by our work-stealing framework scale linearly because of the low contention between consumers. Their performance differences are therefore due to the efficiency of the **consume()** operation – for example, SALSA is $\times 1.7$ faster than SALSA+CAS thanks to its



(a) System throughput – N producers, N consumers.



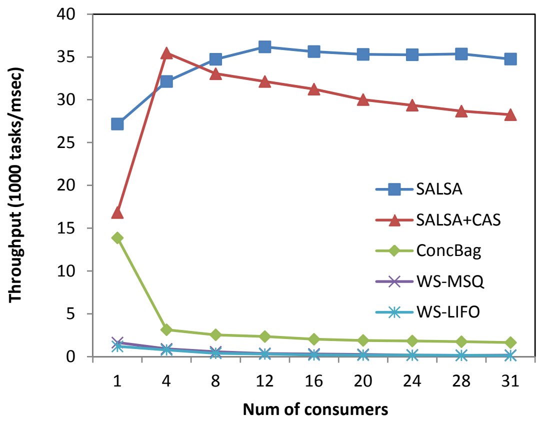
(b) System throughput – variable producers-consumers ratio.

Figure 7.3: System throughput for various ratios of producers and consumers. SALSA scales linearly with the number of threads – in the $16/16$ workload, it is $\times 20$ faster than WS-MSQ and WS-LIFO, and $\times 3.5$ faster than Concurrent Bags. In tests with equal numbers of producers and consumers, the differences among work-stealing alternatives are mainly explained by the consume operation efficiency, since stealing rate is low and hardly influences performance.

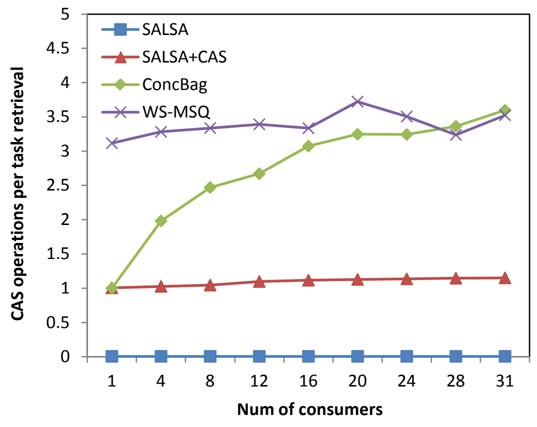
fast-path consumption technique. In contrast, in ConcBags, which is not based on per-consumer pools, every **consume()** operation implies stealing, which causes contention among consumers, leading to sub-linear scalability. The stealing policy of ConcBags algorithm plays an important role. The stealing policy described in the original paper [74] proposes to iterate over the lists using round robin. We found out that the approach in which each stealer initiates stealing attempts from the predefined consumer improves ConcBags' results by 53% in a balanced workload.

Figure 7.3(b) shows system throughput of the algorithms for various ratios of producers and consumers. SALSA outperforms other alternatives in all scenarios, achieving its maximal throughput with equal number of producers and consumers, because neither of them is a system bottleneck.

We next evaluate the behavior of the pools in scenarios with a single producer and multiple consumers. Figure 7.4(a) shows that the performance of both SALSA and SALSA+CAS does not drop as more consumers are added, while the throughput of other algorithms degrades by the factor of 10. The degradation can be explained by high contention among stealing consumers, as evident from Figure 7.4(b), which shows the average number of CAS operations per task transfer.



(a) System throughput – 1 Producer, N consumers.



(b) CAS operations per task retrieval – 1 Producer, N consumers.

Figure 7.4: System behavior in workloads with a single producer and multiple consumers. Both SALSA and SALSA+CAS efficiency balance the load in this scenario. The throughput of other algorithms drops by a factor of 10 due to increased contention among consumers trying to steal tasks from the same pool.

7.4.4 Evaluating SALSA techniques

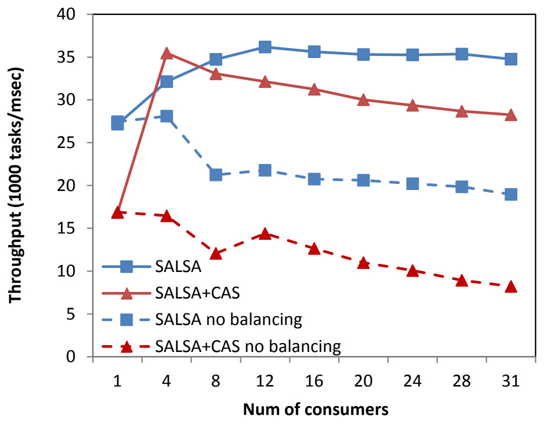


Figure 7.5: System throughput – 1 Producer, N consumers. Producer-based balancing contributes to the robustness of the framework by reducing stealing. With no balancing, chunk-based stealing becomes important.

In this section we study the influence of two of the techniques used in SALSA: 1) chunk-based-stealing with a low-synchronization fast path (Section 7.3.3), and 2) producer-based balancing (Section 7.3.4). To this end, we compare SALSA and SALSA+CAS both with and without producer-based balancing (in the latter a producer always inserts tasks to the same consumer's pool).

Figure 7.5 depicts the behavior of the four alternatives in single producer / multiple consumers workloads. We see that producer-based balancing is instrumental in redistributing the load: neither SALSA nor SALSA+CAS suffers any degradation as the load increases. When producer-based balancing is disabled, stealing becomes prevalent, and hence the stealing granularity becomes more important: SALSA's chunk based stealing clearly outperforms the naïve task-based approach of SALSA+CAS.

7.4.5 Impact of Scheduling and Allocation

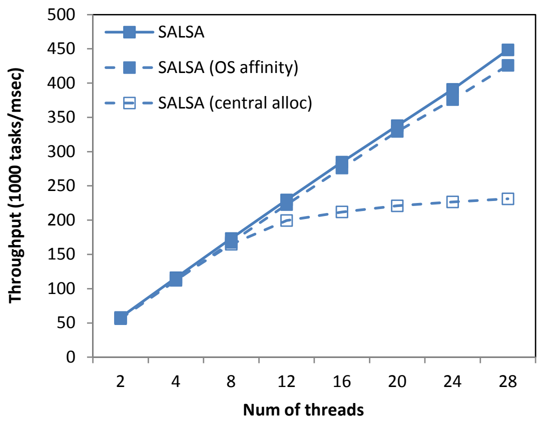


Figure 7.6: Impact of scheduling and allocation (equal number of producers and consumers). Performance decreases once the interconnect becomes saturated.

We now evaluate the impact of scheduling and allocation in our NUMA system. To this end, we compare the following three alternatives: 1) the original SALSA algorithm; 2) SALSA with no affinity enforcement for the threads s.t. producers do not necessarily work with the closest consumers; 3) SALSA with all the memory pools preallocated on a single NUMA node.

Figure 7.6 depicts the behavior of all the variants in the balanced workload. The performance of SALSA with no predefined affinities is almost identical to the performance of the standard SALSA, while the central allocation alternative looses its scalability after 12 threads.

The main reason for performance degradation in NUMA systems is bandwidth saturation of the interconnect. If all chunks are placed on a single node, every remote memory access is transfered via the interconnect of that node, which causes severe performance degradation. In case of random affinities, remote memory accesses are distributed among different memory nodes, hence their rate remains below the maximum available bandwidth of each individual channel, and the program does not reach the scalability limit.

Chapter 8

Conclusions

This thesis dealt with exploiting parallelism of multi-core architectures, and the main topics of our research were efficient synchronization and data exchange among threads.

We studied the theoretical properties of TMs avoiding unnecessary aborts, by investigating what kinds of aborts can or cannot be eliminated, and what kinds of aborts can or cannot be avoided efficiently. We have shown that some unnecessary aborts cannot be avoided, and that there is an inherent tradeoff between the overhead of a TM and the extent to which it reduces the number of spurious aborts: while strict online opacity-permissiveness is NP-hard, we presented a polynomial time algorithm AbortsAvoider, satisfying the weaker online opacity-permissiveness property.

An effective way to reduce the number of aborts in transactional memory is keeping multiple versions of transactional objects. Hence, we studied the inherent properties of STMs that use multiple versions to guarantee successful commits of all read-only transactions (we call such STMs MV-permissive). We presented the challenge of efficient garbage collection of old object versions by demonstrating that the memory consumption of algorithms keeping a constant number of versions for each object can grow exponentially. We then showed that no responsive MV-permissive STM can be optimal in the number of previous versions kept and that no responsive MV-permissive STM can be disjoint-access parallel. We defined an achievable garbage collection property, useless-prefix GC, and showed that in a responsive MV-permissive STM satisfying UP GC, even read-only transactions must make lasting changes to the system state.

Theoretical study of multi-versioning in STM is far from being complete. While we showed that no MV-permissive STM can be online space optimal, it would be interesting to consider whether there exist approximately optimal STMs. There are clear tradeoffs between the quality of garbage collection, permissiveness and the computational complexity of transactional operations: we believe that understanding these tradeoffs may be valuable to improving the performance and utility of transactional memory.

We referred to practical implications of multi-versioning by developing SMV, a multi-versioned STM that achieves high performance (high throughput, low and predictable latency, and little wasted work) in the presence of read-only transactions. Despite keeping multiple versions, SMV can work well in memory constrained environments. It keeps old object versions as long as they might be useful while still allowing read-only transactions to remain invisible by relying on automatic garbage collection to dispose of obsolete versions. SMV demonstrated up to $\times 7$ throughput improvement over a single-version STM and more than a two-fold improvement over an STM keeping a constant number of versions per object.

More generally, SMV presents the idea of keeping needed data in memory by causing potential users to keep references for preventing garbage collection. This idea is especially appealing because it delegates disposal responsibilities to the independent GC module that is being developed and upgraded by a very large community. We think that this approach can be the key to achieving good performance not only in STMs, but also in a range of concurrent data structures.

The second area of this thesis was the efficient data exchange among threads, where we presented a highly-scalable task pool framework. Our framework has employed a number of novel techniques for improving performance: 1) lightweight and synchronization-free produce and consume operations in the common case; 2) NUMA-aware memory management, which keeps most data accesses inside NUMA nodes; 3) a chunk-based stealing approach that decreases the stealing cost and suits NUMA migration schemes; and 4) elegant producer-based balancing for decreasing the likelihood of stealing.

The presented task pool scales linearly with the number of threads: it outperforms other work-stealing techniques by a factor of 20, and the state-of-the art non-FIFO pools by a factor of 3.5. We have further shown that it is highly robust to imbalances and unexpected thread stalls.

Our general approach of partitioning data structures among threads, along with chunk-based migration and an efficient synchronization-free fast-path, can be of benefit in building additional scalable high-performance services in the future.

Bibliography

[1] `http://www.azulsystems.com/blog/cliff-click/2008-05-27-clojure-stms-vs-locks`.

[2] `www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919`.

[3] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 151–162, 2010.

[4] Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Principles of Distributed Systems*, Lecture Notes in Computer Science, pages 395–410.

[5] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, 1998.

[6] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 308–315, New York, NY, USA, 2006. ACM.

[7] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. pages 487–498, 2011.

[8] H. Attiya and E. Hillel. Single-Version STMs can be Multi-Version Permissive. To appear in 12th International Conference on Distributed Computing and Networking (ICDCN11).

[9] H. Attiya and E. Hillel. Brief announcement: Single-Version STMs can be Multi-Version Permissive. In *Proceedings of the 29th symposium on Principles of Distributed Computing*, 2010.

[10] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 69–78.

[11] U. Aydonat and T. Abdelrahman. Serializability of transactions in software transactional memory. In *Second ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[12] D. Basin. Café: Scalable task pools with adjustable fairness and contention. Master's thesis, Technion, 2011.

[13] D. Basin, R. Fan, I. Keidar, O. Kiselov, and D. Perelman. CAFÉ: Scalable task pools with adjustable fairness and contention. In *Proceedings of the 25th International Symposium on Principles of Distributed Computing*, DISC '11.

[14] D. Basin, R. Fan, I. Keidar, O. Kiselov, and D. Perelman. Café: scalable task pools with adjustable fairness and contention. In *Proceedings of the 25th international conference on Distributed computing*, DISC'11, pages 475–488, 2011.

[15] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, 1995.

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

[17] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, 2011.

[18] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

[19] V. Bortnikov, G. Chockler, D. Perelman, A. Roytman, S. Shachor, and I. Shnayderman. Brief announcement: reconfigurable state machine replication from non-reconfigurable building blocks. In *Proceedings of the 31 ACM Symposium on Principles of Distributed Computing*, PODC '12.

[20] V. Bortnikov, G. Chockler, D. Perelman, A. Roytman, S. Shachor, and I. Shnayderman. FRAPPE: Fast replication platform for elastic services. In *Proceedings of the 5th Workshop on Large Scale Distributed Systems and Middleware*, LADIS '11, 2011.

[21] A. Braginsky and E. Petrank. Locality-conscious lock-free linked lists. In *Proceedings of the 12th international conference on Distributed computing and networking*, ICDCN'11, pages 107–118, 2011.

[22] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

[23] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[24] N. Carvalho, J. Cachopo, L. Rodrigues, and A. Rito-Silva. Versioned transactional shared memory for the FenixEDU web application. In *Proceedings of the 2nd workshop on Dependable distributed data management*, pages 15–18, 2008.

[25] L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear. Transactional mutex locks. In *Euro-Par'10*.

[26] D. Dice, H. Huang, and H. Yang. Asymmetric dekker synchronization. Technical report, Sun Microsystems, 2001.

[27] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. pages 157–168, mar 2009.

[28] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining numa locks. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, 2011.

[29] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 194–208, 2006.

[30] D. Dice and N. Shavit. TLRW: Return of the read-write lock. In *TRANSACT '09: 4th Workshop on Transactional Computing*, feb 2009.

[31] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. Snzi: scalable nonzero indicators. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 13–22, New York, NY, USA, 2007. ACM.

[32] R. Ennals. Cache sensitive software transactional memory. Technical report.

[33] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08*, pages 237–246, 2008.

[34] S. M. Fernandes and J. a. Cachopo. Lock-free and Scalable Multi-Version Software Transactional Memory. In *PPoPP '11*, pages 179–188, 2011.

[35] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003.

[36] A. Gidenstam, H. Sundell, and P. Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In *Proceedings of the 14th international conference on Principles of distributed systems*, OPODIS'10, pages 302–317, 2010.

[37] E. Gidron, I. Keidar, D. Perelman, and Y. Perez. SALSA: scalable and low synchronization NUMA-aware algorithm for producer-consumer pools. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12.

[38] E. Gidron, I. Keidar, D. Perelman, and Y. Perez. SALSA: Scalable and Low Synchronization NUMA-aware Algorithm for Producer-Consumer Pools. Technical report, Technion, CCIT Report 807, 2012.

[39] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, 2005.

[40] V. Gramoli, D. Harmanci, and P. Felber. Toward a Theory of Input Acceptance for Transactional Memories. Technical report, 2008.

[41] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in Transactional Memories. In *DISC 2008*, 2008.

[42] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 304–313, 2008.

[43] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, 2008.

[44] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the Second European Systems Conference*, 2007.

[45] D. Hendler, Y. Lev, M. Moir, and N. Shavit. A dynamic-sized nonblocking work stealing deque. Technical report, 2005.

[46] D. Hendler and N. Shavit. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 280–289, 2002.

[47] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC'03*, pages 92–101, 2003.

[48] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

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

[50] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In *Proceedings of the 11th international conference on Principles of distributed systems*, OPODIS'07, pages 401–414, 2007.

[51] I. Keidar and D. Perelman. On avoiding spare aborts in transactional memory. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 59–68, 2009.

[52] G. Korland, N. Shavit, and P. Felber. Noninvasive Java concurrency with Deuce STM (poster). In *SYSTOR '09*, 2009. Further details at `http://www.deucestm.org/`,.

[53] E. Koskinen and M. Herlihy. Dreadlocks: efficient deadlock detection. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 297–303, 2008.

[54] E. Ladan-Mozes, I.-T. A. Lee, and D. Vyukov. Location-based memory fences. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 75–84, 2011.

[55] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, pages 690–691, 1979.

[56] M. Lupon, G. Magklis, and A. González. FASTM: A log-based hardware transactional memory with fast abort recovery. In *PACT '09: Proc. 18th International Conference on ParallelArchitectures and Compilation Techniques*, 2009.

[57] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15:491–504, June 2004.

[58] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, 1996.

[59] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 253–262, 2005.

[60] J. E. B. Moss. Open nested transactions: Semantics and support. In *WMPI*, Austin, TX, February 2006.

[61] J. Napper and L. Alvisi. Lock-free serializable transactions. Technical report, The University of Texas at Austin, 2005.

[62] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 1979.

[63] C. H. Papadimitriou and P. C. Kanellakis. On concurrency control by multiple versions. *ACM Trans. Database Syst.*, pages 89–99, 1984.

[64] D. Perelman, E. Bortnikov, R. Lempel, and R. Sandler. Lightweight automatic face annotation in media pages. In *Proceedings of the 21st World Wide Web Conference*, WWW '12.

[65] D. Perelman, A. Byshevsky, O. Litmanovich, and I. Keidar. SMV: Selective multi-versioning STM. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, DISC '11.

[66] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, PODC '10.

[67] D. Perelman and I. Keidar. SMV: Selective multi-versioning stm. In *5th ACM SIGPLAN Workshop on Transactional Computing*, TRANSACT '10, 2010.

[68] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. *SIGPLAN Not.*, 44(4):163–172, 2009.

[69] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 284–298, 2006.

[70] T. Riegel, C. Fetzer, H. Sturzrehm, and P. Felber. From causal to z-linearizable transactional memory. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing*, pages 340–341, 2007.

[71] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC'05*, pages 240–248, 2005.

[72] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, pages 89–97, 2010.

[73] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *SPAA '08*, pages 275–284, 2008.

[74] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A lock-free algorithm for concurrent bags. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 335–344, 2011.

[75] R. Titos-Gil, M. E. Acacio, J. M. Garcia, T. Harris, A. Cristal, O. Unsal, I. Hur, and M. Valero. Hardware transactional memory with software-defined conflicts. *ACM Trans. Archit. Code Optim.*, pages 31:1–31:20, 2012.

[76] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-lazy hardware transactional memory. In *MICRO '09: Proceedings of the 2009 42nd IEEE/ACM International Symposium on Microarchitecture*, 2009.

[77] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[78] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, 2001.

במכונות מרובות ליבות שיכולות להריץ כמה אפליקציות בו זמנית, ההתנהגות של הסביבה יכולה להיות לא צפויה. העצירות הלא צפויות של החוטים עלולות לגרום לעומס לא מאוזן על הצרכנים, וזה עלול לגרום לקצב גבוה של הגניבות שיוריד את התקורה. אנחנו מממשים מדיניות אלגנטית ומשפרת עצמה עבור הכנסת המשימות, כך שהמשימות לא נדחפות בכוח למאגרי SALSA עם עומס יתר, ובכך אנחנו מורידים את ההסתברות של הגניבות.

מערכת SALSA מומשה ב-C++ והביצועים שלה נבדקו במכונת NUMA עם 32 ליבות. הבדיקות שלנו מראות שמאגר היצרנים-צרכנים מבוסס SALSA סקלבילי באופן לינארי עם כמות החוטים: הוא עובד פי 20 יותר מהר מהאלטרנטיבות של work-stealing האחרות, והוא משפר באופן משמעותי את העבודות האחרות. מאגרים מבוססי SALSA מראים סקלביליות טובה אפילו במקרים הלא מאוזנים.

הגישה הכללית שלנו לחלוקה של מבני נתונים בין החוטים, יחד עם העברת chunkים שלמים והמסלול המהיר ללא סינכרון, יכולה להיות שימושים לבניית שירותים סקלביליים אחרים במערכות מרובות ליבות.

אנחנו מסתכלים על ההיבט הפרקטי של האבורטים המיותרים ומפתחים אלגוריתם STM שנקרא Selective Multi-Versioning. האלגוריתם שלנו מוריד את כמות האבורטים המיותרים, במיוחד לטראנזקציות קוראות ארוכות. SMV מוריד את הגרסאות הישנות ביעילות, יחד עם זאת הוא מאפשר לטראנזקיות הקוראות להיות בלתי נראות ע"י השימוש ב-garbage collection אוטומתי. בנוסף, אנחנו מראים שהשימוש בזיכרון אצל אלגוריתמי STM שמתחזקים כמות קבועה של גרסאות לכל אובייקט יכול לגדול באופן אקספוננציאלי. לעומת זאת SMV ממשיך לפעול כסדרה אפילו תחת אילוצי זיכרון כבדים.

ההערכה שלנו מראה ש-SMV יעיל באופן יוצא מן הכלל בעומסי עבודה עם אחוז ניכר של טראנזקציות קוראות. למשל בבנצ'מרק STMBench7 עם 64 חוטים, התפוקה של SMV גבוהה פי 7 מהתפוקה של TL2 ויותר מפי שתיים מה-TMדים שמחזיקים שתי או שמונה גרסאות. בנוסף לזה, באפליקציות עם חוט מיוחד שמנסה לקחת snapshot קונסיסטנטי פעם אחר פעם, זיכרונות טראנזקציוניים כמו TL2 או זיכרונות עם כמות קבועה של גרסאות לכל אובייקט לא יכולים לקחת את ה-snapshot אפילו עם חוט בודד שרץ ברקע. לעומת זאת, הביצועים של SMV נשארים יציבים ללא קשר לכמות החוטים שרצים בו זמנית ברקע.

אנחנו משווים את דרישות הזיכרון של האלגוריתמים על ידי הגבלת את מרחב הזיכרון ב-Java. למרות שה-STMים שמחזיקים את כמות קבועה של הגרסאות לכל אובייקט קורסים עם OutOfMemoryException, המערכת שלנו ממשיכה לעבוד והתפוקה שלה יורדת בפחות מ-25% אפילו תחת אילוצי זיכרון כבדים.

SMV מציג גישה חדשה לשמירת גרסאות מרובות, אשר מאפשרת לטראנזקציות הקוראות להישאר בלתי נראות ומאצילה את הסמכות של מחיקת גרסאות ישנות למנגנוני GC הקיימים בשפות מנוהלות כמו Java.

עוד אספקט חשוב של מערכות מרובות ליבות שאנחנו מסתכלים עליו זה העברת מידע יעילה בין החוטים. אנחנו מציגים מאגר משימות סקאלבילי עבור צרכנים-יצרנים, שבנוי עם דגש מיוחד על הסינכרון הדל והלוקליות של המידע. הגרעין של המאגר שלנו הוא SALSA, אלגוריתם סינכרון סקאלבילי ודל תקורה שמממש את המאגר עבור צרכן בודד ותומך בפעולות גניבה. כל צרכן פועל על מאגר SALSA משלו וגונב משימות מצרכנים אחרים על פי הצורך.

האתגר של הגניבה הוא לבצע אותה בלי להידרדר לביצוע פעולות אטומיות חזקות (כמו CAS או גדר זיכרון) עבור כל הוצאת משימה. אנחנו פותרים את האתגר הזה באמצעות אלגוריתם גניבה חדשני שמבוסס על גניבת chunks. אלגוריתם הגניבה שלנו מאפשר לפעולות הצריכה להיות synchronization-free במקרה הרגיל, כאשר אין פעולות גניבה (אנחנו קוראים למקרה הזה המסלול המהיר). יותר מזה, SALSA מקטין את קצב הגניבות על ידי העברת chunkים שלמים בתוך פעולת גניבה בודדת, אשר דורשת שתי פעולות CAS בלבד.

על מנת להשיג את הלוקליות של גישות זיכרון על מכונות NUMA, ה-chunkים של SALSA נשמרים בזיכרון הלוקלי של הצרכנים. מדיניות הניהול משדכת את היצרנים והצרכנים לפי הקירבה שלהם, מדיניות כזאת מאפשרת להעביר את רוב המשימות בתוך צמתי NUMA.

תקציר

בעשור האחרון היינו עדים לשינויים מהפכניים בפרדיגמות תכנותיות. בזמן שעולם החומרה התפתח לכיוון של הגדלת אלמנטים חישוביים וההיטרוגניות שלהם, עולם התוכנה נאלץ להסתגל לדרישות החדשות ולעמוד מול האתגר של סיבוכיות החומרה והמקביליות הגדלה. הפיתוח של תוכנות סקלביליות הפסיק להיות הגומחה של המקצוענים המעטים: אלפי מתכנתים "פשוטים" נדרשים לבנות אפליקציות מקביליות יעילות. המצב הזה מעלה צורך לפיתוח כלים חדשים שיכולים לעזור לכתיבת התוכנה בעידן ריבוי הליבות. העבודה הזאת דנה במנגנוני סינכרוניזציה ובהחלפת מידע יעילים בין החוטים.

החלק הראשון של התזה עוסק בבעיות התיאורטיות והמעשיות של הזיכרון הטראנזקציוני (TM). זיכרון טראנזקציוני הינו אבסטרקצית סינכרון אשר מאפשרת לחוטים לארוז סט פעולות על האובייקטים בזיכרון לתוך טראנזקציות. בדומה לטראנזקציות במסדי נתונים, טראנזקציות בזיכרון מתבצעות באפן אטומי: או שכל הפעולות של הטראנזקציה קורות בבת אחת (במקרה כזה אומרים שהטראנזקציה עשתה commit), או שאף פעולה של הטראנזקציה לא מופיעה כלפי חוץ (במקרה כזה אומרים שהטראנזקציה עשתה abort).

מערכות TM הקיימות היום יכולות לבטל את הטראנזקציה שהייתה יכולה לעשות commit בלי להפר נכונות. אנחנו קוראים לאבורטים כאלה אבורטים מיותרים. אנחנו מתייגים איזה אבורטים אפשר למנוע ואיזה לא. בהמשך, אנחנו חוקרים איזה אבורטים מיותרים אפשר למנוע ביעילות. בין היתר, אנחנו מראים שקיימים אבורטים מיותרים שאי אפשר למנוע אותם, ושקיים האיזון בין התקורה של ה-TM לבין הכמות של האבורטים המיותרים. אנחנו גם מציגים אלגוריתם TM שעובד בזמן פולינומיאלי אשר מונע סוגים מסויימים של אבורטים מיותרים.

הדרך היעילה להקטין את כמות האבורטים המיותרים בזיכרון טראנזקציוני היא לשמור גרסאות מרובות של האובייקטים הטראנזקציוניים. שימוש בגרסאות מרובות יעיל במיוחד עבור טראנזקציות קוראות: אם נשמור מספיק גרסאות ניתן להבטיח שכל טראנזקציה קוראת מסתיימת בהצלחה על ידי קריאת ה-snapshot הקונסיסטנטי של אוסף האובייקטים הנקראים. לאור זאת אנחנו חוקרים את התכונות האינהרנטיות של אלגוריתמי TM שמשתמשים בגרסאות מרובות על מנת להבטיח סיום מוצלח של כל טראנזקציה קוראת.

קודם כל אנחנו מראים שאלגוריתמים האלה לא יכולים להיות disjoint-access parallel. אחר כך אנחנו מסתכלים על הבעיה של איסוף הגרסאות הישנות ומראים שאף STM לא יכול להיות אופטימלי במספר הגרסאות הישנות שהוא מחזיק. יותר מכך, אנחנו מראים שהאיסוף המדוייק של הגרסאות הישנות הוא בלתי אפשרי באלגוריתמי STM הממומשים עם invisible reads. אנחנו מציגים אלגוריתם STM לדוגמה, שמשתמש ב-visible reads ושאוסף את הגרסאות הישנות שלא בשימוש.

המחקר נעשה בהנחיית פרופ' עדית קידר מהפקולטה להנדסת חשמל.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

ניצול המקביליות
במערכות מרובות ליבות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

דמיטרי פרלמן

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

אלול ה'תשע"ב חיפה ספטמבר 2012

ניצול המקביליות
במערכות מרובות ליבות

דמיטרי פרלמן