

Concurrent Data Structures: Methodologies and Inherent Limitations

Eshcar Hillel

Concurrent Data Structures: Methodologies and Inherent Limitations

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Eshcar Hillel

Submitted to the Senate of
the Technion — Israel Institute of Technology
Adar aleph 5771 Haifa February 2011

Acknowledgments

The research thesis was done under the supervision of Prof. Hagit Attiya in the Computer Science Department.

I wish to thank Hagit Attiya for being my mentor throughout these years. Her professional and personal support, her time and patience, her guidance and endless knowledge are invaluable to me. I learned so much from her and I will always consider her a role model for excellence.

I thank Alessia Milani for her collaboration and friendship; I thank David Hay and Keren Censor-Hillel, my academic brother and sister, for their help and good advice whenever it was needed.

I am thankful to the committee members: Prof. Maurice Herlihy, Prof. Idit Keidar, and Prof. Erez Petrank, as well as to Prof. Nir Shavit, Prof. Yehuda Afek, Prof. Oded Shmueli, and Prof. Assaf Schuster for their helpful comments and suggestions.

Prof. Faith Ellen, Prof. Panagiota Fatourou, Vincent Gramoli, Prof. Rachid Guerraoui, Danny Hendler, Michal Kapalka, Alex Kogan, Petr Kuznetsov, Alex Shraer, Prof. Michael Spear, Prof. Gadi Taubenfeld, and Martin Vechev reviewed earlier drafts of the work presented in this dissertation and provided me with valuable comments.

This dissertation is dedicated to Dubbi Hillel, for always encouraging me to aim high and thrive whilst reminding me to enjoy the journey thereof.

My love goes to my children for having the pleasure of watching them grow; it allows everything to happen with a smile.

Special thanks to my parents, sisters, and extended family, who have been a source of inspiration; above all I thank them for being my nest.

The generous financial support of the Technion, the Israel Science Foundation, and the Gutwirth and Zef Foundations, is gratefully acknowledged.

Contents

Abstract	1
Abbreviations and Notations	3
1 Introduction	5
1.1 Multi-word Synchronization	7
1.2 Concurrent Doubly-Linked List	10
1.3 Disjoint-Access Parallel Transactional Memory	11
1.4 Transactional Memory with Nontransactional Accesses	12
1.5 Outline of the Thesis	14
I Methodologies for Highly-Concurrent Data Structures	15
2 Concurrent Data Types	17
2.1 Safety Property: Linearizability	18
2.2 Progress and Locality Properties	19
3 Multi-word Synchronization Operations	21
3.1 BLocalRMW: A Blocking Algorithm with $3k$ Failure Locality	22
3.1.1 Data Structures	23
3.1.2 Implementation	24
3.1.3 An Execution Example	26
3.1.4 Correctness Proof	26
3.2 LocalRMW: A $3k$ -Local Nonblocking Algorithm	35
3.2.1 Data Structures	35
3.2.2 Implementation	36
3.2.3 Safety Proof	39
3.2.4 Progress and Locality Proofs	43
3.3 Example: k CAS	46

4	Doubly-Linked Lists	49
4.1	CAS-Chromo: Priority Queue and Deque	50
4.1.1	Data Structures	53
4.1.2	Implementation	54
4.2	DCAS-Chromo: A Doubly-Linked List Algorithm	58
4.3	Safety Proof	61
4.4	Progress and Locality Proofs	65
4.4.1	DCAS-Chromo	65
4.4.2	CAS-Chromo	72
5	Related Work	75
II	Inherent Limitations for Transactional Memory	81
6	The Transactional Memory Approach	83
6.1	Software Implementation of TM	84
6.2	Safety Properties	85
6.3	Progress Properties	85
6.4	Disjoint-Access Parallelism	86
6.5	Invisibility of Reads	88
7	Limitations on Disjoint-Access Parallel STMs	89
7.1	Impossibility of Invisible Read-Only Transactions	90
7.2	Lower Bound for Read-Only Transactions	96
7.3	Weaker Consistency Conditions	100
7.3.1	Snapshot Isolation	100
7.3.2	Serializability	101
8	Limitations on STMs with Nontransactional Accesses	105
8.1	Eager STMs	107
8.2	Uninstrumented Access without Prior Privatization	108
8.3	Privatization with Invisible Reads	110
8.4	Privatization with Visible Reads	113
8.5	Reducing the Cost of Privatization	119
8.6	Bounds for Disjoint-Access Parallel STMs	120
9	Related Work	123
10	Summary	129
	Bibliography	133

List of Figures

2.1	The CAS and DCAS primitives	18
2.2	Operations conflict graph example	19
3.1	The k RMW and k CAS operations	21
3.2	Data structures and classes for BLocalRMW	23
3.3	Conflicting operations example	27
3.4	A recurring resets scenario	28
3.5	An increasing-decreasing chain	31
3.6	Data structures and classes for LocalRMW	36
4.1	Sequential specification of the linked list operations	51
4.2	Overlapping operations of a doubly-linked list	51
4.3	An example of a doubly-linked list with a single node	52
4.4	The state transitions of an operation	54
4.5	Data structures for CAS-Chromo and DCAS-Chromo	55
4.6	An example of a symmetric scenario of remove operations	59
4.7	An example of a <i>PushLeft</i> execution	66
4.8	An example of a <i>PopLeft</i> execution	68
4.9	An example of a <i>Remove</i> operation	69
4.10	Worst case scenario for overlapping deque operations	74
5.1	A recursive helping chain	77
6.1	An example of transactions execution	84
6.2	Transactions conflict graph example	87
7.1	A flippable execution of length k with two updaters	91
7.2	Illustration for the proof of Lemma 7.2	93
7.3	Illustration for the proof of Lemma 7.5	97
7.4	An augmented flippable execution \widehat{E}_k	102
8.1	Privatization example of a linked list	106

8.2	The execution used in the proof of Theorem 8.2	109
8.3	The executions used in the proof of Theorem 8.4	112
8.4	An example workload for the proof of Theorem 8.7	113
8.5	The executions used in the proof of Theorem 8.7	115

List of Tables

5.1	Comparison of linked list algorithms	76
5.2	Comparison of multi-word synchronization implementations	77
9.1	Comparison of strict serializable STMs	125
9.2	Comparison of STMs supporting privatization	126

Abstract

No matter how fast processors get in recent decades, applications find new ways to consume the extra speed. While the trend of increasing software demands continues consistently, the traditional approach of faster processes comes to an end, forcing major processor manufacturers to turn to multi-threading and multi-core architectures, in what is called the *concurrency revolution*. This means putting several cores on one chip, each running several threads in parallel. Although driven by hardware changes, it calls for a software revolution, and a fundamental turn toward concurrency in applications is needed to fully exploit the throughput gains of multi-core processors.

At the heart of many concurrent applications lie *concurrent data structures*, the subject of this thesis. Concurrent data structures coordinate access to shared resources; implementing them is hard, and it is a great challenge to make concurrent programming easier for the average programmer. The main goal of this thesis is to provide programmers with tools to enable them writing, understanding, and maintaining concurrent data structures, and concurrent applications in general, more easily.

We provide an algorithm for a multi-word synchronization operation, which facilitates the design of concurrent data structures by allowing access to several data items in one atomic operation. The additional amount of information stored in each data item is constant. In addition, we present a new approach in which implementations are aware of the semantics of the data structure yet they follow a general methodology. In this *built-in coloring* scheme, colors are integrated into items, and the implementation maintains a legal coloring while applying operations to data items. We demonstrate this approach with lock-free algorithms for list-based data structures such as double-ended queue, priority queue and doubly-linked list.

These algorithms are the first to provably decrease interference among operations, thereby increasing the throughput of the execution. In particular, our list-based data structures guarantee that only operations on adjacent items in the list interfere with each other; in our implementations of multi-word synchronization, operations are effectively isolated from other operations that are not nearby, hence, distant operations may proceed concurrently. We suggest measures making it easier to evaluate the interference diameter of the algorithms, and prove their correctness.

Transactional memory (TM), in which concurrent processes synchronize via in-memory transactions, has been proposed as an alternative synchronization mechanism. It promises to alleviate many of the challenges of concurrent programming. In its simplest form, the programmer defines a transaction by enclosing a set of statements in an atomic block; the underlying run time system is responsible for executing the transaction, while handling any contention issues raised due to concurrency.

Transactional memory raises a lot of hope for mastering the complexity of concurrent programming. To guide algorithm designers in their attempt to find better and more efficient implementations and to demonstrate which directions are futile, we explore the boundaries and tradeoffs of TM.

We show that there is an inherent tradeoff: no TM implementation can avoid interference on disjoint data and have read-only transactions that always complete successfully while never writing to the memory. In fact, we prove that read-only transactions in such implementations, reading t items, must write to at least $t - 1$ memory locations.

In practice, atomic blocks that often constitute the transactions may contain statements with irreversible effects that cannot be executed within the context of a transaction. Therefore, TM must allow accessing the same items from inside and outside a transaction; this is crucial both for interoperability with legacy code and in order to improve the performance of the TM. Supporting *privatization*, allows the programmer to “isolate” some items making them private to a process; the process can thereafter access them nontransactionally, without interference by other processes. We study the theoretical complexity of privatization, and show an inherent cost, linear in the number of privatized items. The assumptions needed to prove the bounds indicate that limiting the parallelism of the TM or tracking the items of other transactions are the price to pay for efficient privatization.

Abbreviations and Notations

n	—	The number of processes
p, q, p_i	—	A process
op, op_i	—	An operation
T_i	—	A transaction
U_i	—	An update transaction
v_i, t_i	—	A data item
m_i	—	A node
C	—	A configuration
$\alpha, \alpha_i, \beta_i, \gamma_i$	—	An execution interval
I_i, J_i, \hat{I}_i	—	A part of an execution interval of a transaction
d	—	The neighborhood of an operation
k	—	The number of items accessed by a multi-word operation
CAS	—	Compare and swap
DCAS	—	Double compare and swap
RMW	—	Read-modify-write operation
ADT	—	Abstract data type
TM	—	Transactional memory
STM	—	Software transactional memory

Chapter 1

Introduction

Clock frequency is hardly advancing in recent years; performance gains of processing units can no longer stem from increasing the clock speed. Instead, major chip manufacturers are shifting the focus from improving the speed of individual processors into packing more execution cores onto a single chip [68, 75]. *Multi-core* technology refers to a processor with more than one engine, allowing for greater efficiency since the processor workload is basically shared. Including but not exclusively high-end machines, multi-cores are already common in desktop computers and laptops, and are well on their way into mobile phones and other small devices.

As multi-core and multiprocessing architectures are becoming commonplace, modern applications require *concurrent data structures* for their computations. Concurrent data structures can be accessed simultaneously by multiple threads running on several cores. Important examples are *doubly-linked lists*, used for building data structures such as stacks, queues, hash tables and skip lists; *b-trees*, which are often used to index the data in large databases; and *AVL trees*—self-balancing binary trees, which offer improved performance for searching over linear data structure such as lists.

Designing concurrent data structures and ensuring their correctness is a difficult task, significantly more challenging than doing so for their sequential counterparts. The difficulty of concurrency is aggravated by the fact that threads are asynchronous since they are subject to page faults, interrupts, and so on. To manage the difficulty of concurrent programming, multithreaded applications need *synchronization* to ensure thread-safety by coordinating the concurrent accesses of the threads. At the same time, it is crucial to allow many operations to make progress concurrently and complete without interference in order to utilize the parallel processing capabilities of contemporary architectures.

The traditional approach that helps maintaining data integrity among threads is to use *lock* primitives. Mutexes, semaphores, and critical sections are used to ensure that certain sections of code are executed in exclusion.

Locks, however, have many pitfalls. The first and problem with locks is the fact that they are *blocking*. If one thread attempts to acquire a lock that is already held by another thread, the first thread blocks until the lock is free. This may result in *convoying* when a blocked thread has to wait for the lock held by a stalled thread. If the thread owning the lock goes into any sort of infinite loop, the blocked thread may wait forever. In applications with priority levels, locks may induce *priority inversion*, where a high-priority thread has to wait until a lock held by a low-priority thread is released; such scenarios are undesirable especially when performing real-time tasks. Moreover, lock-based implementations are prone to bugs since, in most cases, the programmer must follow a complicated locking policy; not doing so may result in errors such as holding the wrong lock, or not holding a lock altogether, which are very hard to debug.

Other problems are subject to the locking granularity. While *coarse-grained* locking, in which one lock protects a large portion of the data structure, if not all of it, are simple, they can significantly reduce the scalability of the application due to lock contention. On the other hand, *fine-grained* locking uses many locks to protect small units of the data structure. Acquiring and releasing many locks introduces significant overhead, and requires more careful design. In particular, avoiding *deadlocks*, in which threads block each other in a cycle, and *livelocks*, in which threads are repeatedly forced to release all the locks they have acquired and restart, increases the complexity of fine-grained algorithms. In many cases, non-expert programmers lack the skills necessary to build correct and efficient applications that harness the computational power offered by multi-cores.

Lock-free implementations do not rely on mutual exclusion, thereby avoiding some of these inherent problems. Most lock-free implementations guarantee that in any infinite execution, some pending operation completes within a finite number of steps. These implementations must rely on strong primitives [53], e.g., *compare&swap* (CAS), which atomically updates a memory location if its content is some expected value.

Designing lock-free algorithms is often complex and hard to get right just as fine-grained locking implementations. Even for relatively simple, key data structures, like *double-ended queue* (*deque*)—a queue for which items can be added to or removed from both head and tail—lock-free implementations make significant compromises. Some implementations may contain garbage nodes [51], others statically limit the size of the data structure [54] or do not allow operations on both ends of the queue to proceed concurrently [74]. Even when *double compare&swap* (DCAS)—allowing atomic update of two memory locations—is used, some implementations either are inherently sequential [42, 43] or allow access to chains of garbage nodes [35].

Transactional memory (TM) [56, 91] is another popular approach for alleviating the difficulty of programming concurrent applications for multi-core and multiprocessing systems. Inspired by classical database transactions [100], TM enables the systematic translation of sequential data structures into correct concurrent implementations,

offering flexibility in the design of concurrent applications.

A *transaction* encapsulates a sequence of operations, and it is guaranteed that if any operation takes place, they all do, and that if they do, they appear to other threads to do so atomically, as one indivisible operation. A *software implementation* of transactional memory (STM) translates high-level transaction operations on data items to low-level primitive operations on memory locations containing the data and the meta-data needed for the implementation.

Aiming at making concurrent applications as easy to write as sequential code, TM hides sophisticated synchronization mechanisms under a simple veil. The programmer only needs to declare the boundaries of the transaction, and the TM implementation is responsible for executing the annotated code, such that each transaction appears to be executed atomically.

This thesis investigates implementations of concurrent data structures, addressing mostly lock-free implementations but also fine-grained lock-based implementations. It provides methodologies that facilitate the task of implementing concurrent data structures, and suggests tools to prove their correctness and tools to predict their scalability by measuring their locality. Our research further explores inherent limitations on concurrent programming, specifically addressing implementations of transactional memory. TM is seriously considered as part of software solutions and as a basis for novel hardware designs. It is therefore imperative to understand inherent tradeoffs in the design and implementation of transactional memory.

The rest of the introduction is dedicated to describing our contributions.

1.1 Multi-word Synchronization

Multi-word synchronization operations, like *k-read-modify-write* (*kRMW*), read the contents of several data items, compute new values and write them back, all in one atomic operation. A popular special case is *k-compare&swap* (*kCAS*), where the values read from *k* data items are compared against specified values, and if they all match, the items are updated.

Multi-word synchronization facilitates the design and implementation of lock-free concurrent data structures, making it more effective and easier than when using only single-word synchronization operations. For example, removing a node from a doubly-linked list is easy if 3CAS is used to atomically access three nodes—the node to be removed and the two nodes before and after it; a right or left rotation applied on a node in an AVL tree can easily be implemented if 4CAS is used to access the node and its parent and two children.

Modern architectures, however, support in hardware only synchronization primitives like CAS or *load-link/store-conditional* (LL/SC), accessing a single location, or at best, DCAS. Thus, *kRMW* and *kCAS* must be provided in software, at least today.

It is crucial to allow many operations to make progress concurrently and complete without interference in order to utilize the capabilities of contemporary architectures. Clearly, when operations need to simultaneously access the same item, an inherent “hot spot” is created and operations cannot proceed concurrently. However, typical implementations of k RMW create an additional, not obviously necessary, delay, when the progress of an operation is hindered due to operations that do not access the same items. In these implementations, e.g., [20, 52, 62, 91, 98], an operation tries to acquire all the items it needs one by one; if another operation already acquired an item, the operation is blocked and can either *wait* for the item to be released (possibly while *helping* the conflicting operation make progress) or *reset* the conflicting operation and try to acquire the item. As a result, *chains* of operations delaying each other may be created (Chapter 5 presents such examples). It is possible to construct recurring scenarios where an operation is delayed a number of steps proportional to the length of such a chain, causing a lot of work to be invested, while only a few operations complete.

These considerations can be described more precisely through the *conflict graph* of operations that overlap in time. In this graph, vertices represent operations, and an edge connects two vertices if they access the same data item. The *distance* between two operations in a conflict graph is the length of the shortest path between them. For example, simultaneous operations accessing the same data item are at distance one in the conflict graph (see the definition in Chapter 2).

When operations choose their items uniformly at random, it has been shown [49], both analytically and experimentally, that paths in the conflict graph have non-constant length, depending on the total number of operations. This means that even if an operation waits for or helps only (transitively) conflicting operations, it can be delayed by “distant” operations.

These adverse effects of delay chains can be mitigated, greatly improving concurrency, if operations are delayed only due to operations within a fixed distance. Informally, an implementation is *d-local nonblocking* if whenever an operation op takes an infinite number of steps, some operation within distance d from op completes. This implies that the throughput of the algorithm is localized in components of diameter d in the conflict graph, and operations are effectively isolated from operations at distance $> d$.

Being $O(k)$ -local nonblocking, implies that the implementation is *nonblocking* [53] as it guarantees that in every execution, some operation completes after a finite number of steps of some process.

Our first contribution is an $O(k)$ -local nonblocking implementation of k RMW. The implementation stores a constant amount of information (independent of k) in each data item; it does not fix k across operations; and it can be adapted to be dynamic, i.e., get the items one-by-one.

Our main new algorithmic ideas are explained in the context of a *fine-grain* im-

plementation, **BLocalRMW**, in which the delay of an operation may block operations that access *nearby* data items; operations that access data items that are farther than $O(k)$ away in the conflict graph are not affected. This is a variant of the *failure locality* property [24].

A key algorithmic idea is that the effect of delays can be bounded, yielding better concurrency, if an operation decides whether to wait for another operation or reset it by comparing how advanced they are in acquiring their data items. If the conflicting operation is more advanced, the operation waits; otherwise, the operation resets the conflicting operation and seizes the item, so it is not overtaken by another operation.

While a similar approach has been used in many resource allocation algorithms, dating back to the classical *wound-die* and *wound-wait* deadlock prevention schemes [86], it is not at all obvious that it ensures locality. In particular, operations may repeatedly reset each other, without any operation completing within $O(k)$ distance, i.e., $O(k)$ -neighborhood. Therefore, a critical step is analytically bounding the locality properties of this approach; a challenging part of the proof shows that an operation cannot be repeatedly reset without some operation completing in its $O(k)$ -neighborhood. Our proof uses a *potential* method to show progress in the neighborhood of any operation op even if it is repeatedly reset. Toward this end, we maintain a *potential vector* in which each entry indicates the number of items acquired by an operation in the $O(k)$ -neighborhood of op ; we show that the vector increases with each reset of op , and therefore, eventually some operation in the $O(k)$ -neighborhood of op acquires all its data items and completes.

Another important challenge is in handling the symmetric situation, when overlapping operations that have made the same progress, i.e., acquired the same number of items, create a chain in the conflict graph. When breaking symmetry, care is needed to avoid deadlocks and guarantee progress. Breaking symmetry between two operations by relying on the operation identifiers may still create a long delay chain (which may involve all processes), for example, when each process in the chain waiting to acquire the next item is delayed by another process with a higher operation identifier. To avoid these delays, we break ties by having conflicting operations try to *atomically acquire the two operations owning the same number of items*, using DCAS. This efficiently partitions symmetric chains into disjoint constant-length chains, ensuring that operations are delayed only due to close-by conflicts. This is the only scenario in which DCAS is employed: the less common these scenarios are, the less frequently DCAS is used.

Next we present **LocalRMW**, which guarantees progress even when processes stop taking steps, by *helping* a blocking operation that is more advanced instead of waiting for it to complete; we still reset conflicting operations that are less advanced. This algorithm shows how helping mitigates the impact of process failures and proves, in a manner similar to the proof of **BLocalRMW**, that **LocalRMW** is $O(k)$ -local nonblocking.

While other methods avoid long delay chains and improve locality by using colors [5,

8] or randomization [49, 90] to break the symmetry, our implementations demonstrate that DCAS is an effective way to break symmetry and improve locality.

1.2 Concurrent Doubly-Linked List

Generic methodologies applying, for example, multi-word synchronization operations are useful in deriving concurrent data structures. Using these schemes as black boxes facilitates the design as the programmer is unaware of (and is not required to care for) their internal working. The cost of this lack of knowledge is the overhead entailed by the implementation of these schemes, which is greater than required in most cases. Therefore, handcrafted implementations of specific data structures are indispensable.

Our contribution is an approach for handcrafted implementations of a concurrent linked list with improved locality. Using the memory addresses of nodes to determine the order they are acquired can lead to a long delay chain. However, operations on the linked list access its constituent nodes in a predictable, well-organized manner, i.e., two or three consecutive nodes in the list. This fact is exploited to determine the order of nodes acquisition using identifiers that are different from their memory addresses.

A *small set of colors* is built into the nodes; to avoid deadlocks while ensuring short delay chains, the nodes are acquired by order of their colors. The operations guarantee that the modifications applied to the data structure preserve the legality of the nodes' coloring; this is possible since the implementation initializes the data structure and provides operations that are the only means for manipulating it.

Our first algorithm, **CAS-Chromo**, allows removals only at the ends of the linked list and uses only CAS; it can be used as a simple deque or a priority queue. The algorithms we present are conceptually simple: A 3-coloring of the nodes is used by operations to determine the order in which nodes are acquired. After acquiring nodes, the operations apply their changes in isolation. A remove (pop) operation acquires *three* consecutive nodes in the list (at the end), and in a legal coloring, two of these nodes may have the same color. In **CAS-Chromo**, equally colored nodes are acquired by their list order (from left to right) to ensure that there are no deadlocks and that some operation makes progress at any time. Furthermore, removing a node might entail recoloring one of its neighbors while ensuring its neighbor's color is not changed concurrently.

Since the list is 3-colored, we can show that **CAS-Chromo** is *5-local nonblocking*, namely, an operation is delayed only due to operations on nodes within distance 5 of its own nodes on the linked list. When insertions are limited to occur at the ends (i.e., a deque), the analysis can be further refined to show that the algorithm is *3-local nonblocking*; this means that operations at the two ends of a deque containing *more than three nodes* do not delay each other.

Our second algorithm, **DCAS-Chromo**, implements a linked list that allows insertions and removals anywhere in the list and uses DCAS. Removing nodes from the middle

of the linked list is more difficult: a chain of overlapping operations, each trying to acquire two equally colored nodes, can be created. This scenario can occur also in CAS-Chromo, but since there the removals are limited to the ends, the chain contains at most two operations, whereas in DCAS-Chromo the chain can be as long as the number of concurrent operations. Relying on the order of the nodes in the list when acquiring the nodes, as is done in CAS-Chromo, may create long hold-and-wait chains. Instead, we employ DCAS to *atomically acquire two nodes with the same color*. This is another example of utilizing DCAS to break symmetry.

DCAS-Chromo is *4-local nonblocking*, namely, it guarantees that operations on nodes that are separated by more than four nodes never delay each other.

1.3 Disjoint-Access Parallel Transactional Memory

Turning our attention again to generic methodologies for obtaining concurrent data structures, we explore tradeoffs of transactional memory. TM aims to *simplify* the design of parallel systems, while providing *safety*, as well as improve the performance with respect to sequential code by exploiting the *scalability* opportunities offered by multicore systems. However, coarse-grained implementations of TM are often not as scalable as expected, whereas fine-grained implementations entail a non-trivial amount of extra work. We ask whether the promise of transactional memory can be realized, and whether there are implementations that are simple, safe, and scalable.

One property that is considered critical for the scalability of a transactional memory implementation is *disjoint-access parallelism*: transactions on disconnected data do not interfere. Several STMs, e.g., [18, 55], guarantee that transactions access the same memory location only if they are connected in the conflict graph. Disjoint-access parallelism can be considered as the unquantified approach of the local nonblocking property, as *d*-local nonblocking transactions, for any finite *d*, do not delay each other if they are disconnected in the conflict graph.

Another important goal is to optimize *read-only transactions*. It is desirable that the implementation of read-only transactions does not write to the memory, i.e., it is *invisible*, so as to reduce memory contention. Moreover, since read-only transactions do not write to data items at the high-level, it seems plausible that they should eventually be able to obtain a consistent view of the data, provided previous versions are kept (as is done in multi-version implementations [76, 82, 83]). Thus, read-only transactions should terminate successfully, regardless of concurrent transactions; i.e., be *wait-free*.

None of the existing transactional memory implementations is both disjoint-access parallel and has invisible, wait-free read-only transactions. Some are disjoint-access parallel and have invisible but not wait-free read-only transactions [18, 55], while others have invisible, wait-free read-only transactions but are not disjoint-access parallel [82].

The thesis shows that this is an inherent tradeoff—no transactional memory implementation can be disjoint-access parallel and have invisible, wait-free read-only transactions—and one of these desirable properties must always be compromised. In fact, we prove a stronger result, showing that in a disjoint-access parallel transactional memory implementation with wait-free read-only transactions, a transaction reading t data items must write to at least $t - 1$ memory locations. Thus, a read-only transaction must perform one (low-level) write essentially for each item it is reading.

The wait-freedom requirement might seem too restrictive for practical purposes. However, *permissive* implementations [45], which avoid spurious aborts by aborting a transaction only if necessary for ensuring consistency, never abort a read-only transaction, as they can always return values that do not harm consistency. Therefore, having read-only transactions that are not wait-free means the implementation is not permissive. Even if it is sufficient for a read-only transaction to *eventually* commit (after aborting several times), we can prove a similar result where a read-only transaction repeatedly aborts and never commits.

The consistency condition commonly used for transactional memory is *opacity* [47] which is similar to requiring *strict serializability* [78] applied to all transactions (including each aborted transaction, separately), extended to allow operations other than reads and writes. Our proofs only assume strict serializability, and hence hold also under the assumption of opacity, and since we only consider executions with non-aborting transactions it holds also for *virtual world consistency* [60]. In fact, we show how to extend the results to hold also for weaker consistency conditions, *snapshot isolation* [21] and *serializability* [78].

1.4 Transactional Memory with Nontransactional Accesses

A transaction is usually given as a code block that is executed by a single process. In practice, atomic blocks of legacy code often contain operations that are not pure reads and writes. Irrevocable operations, such as I/O calls, allocation and deletion of objects, and lock acquisition, cannot be instrumented to execute within a transaction. In other cases, operations are simply preferred not to be executed within the context of a transaction to improve performance. Hence, applications must access the same memory locations from inside and outside a transaction. This may create scenarios in which transactions are not executed in isolation, and the correctness of the application is no longer guaranteed.

Strong atomicity [66, 71, 92] guarantees consistent ordering of transactions in the presence of non-transactional memory accesses. Supporting strong atomicity efficiently is crucial both for interoperability with legacy code and in order to improve performance, and it should be provided without sacrificing the parallelism offered by modern architectures.

A simple solution is to make each nontransactional operation a (degenerate) transaction, but this means that nontransactional operations incur the overhead associated with a transaction. Although compiler optimizations can reduce this cost in some situations [3, 89], they do not alleviate it completely. Thus, STMs seek to improve performance by supporting *uninstrumented* nontransactional operations [44, 94], which are executed as is, typically as a single access to the shared memory.

An alternative solution is to sacrifice simplicity; some responsibility is shifted to the programmer, complicating the interface between the application and the TM. The programmer is required to adhere to certain rules and guidelines, so as to ensure the integrity of the data, much like is the case in fine-grained implementations.

Many recent STMs [27, 32, 33, 39, 67, 70, 72, 77, 95] provide strong atomicity by supporting *privatization* [92, 94], thereby allowing the programmer to “isolate” some items making them private to a process; the process can thereafter access them nontransactionally, without interference by other processes. It is commonly assumed that privatizing a set of items simply involves disabling all shared references to these items [33, 67, 95], e.g., by nullifying these references. However, it has been claimed that privatization is a major source of overhead for transactional memories [101], and that supporting uninstrumented nontransactional operations can seriously limit their parallelism [23].

We formally prove that indeed, in many important workloads, the hope to combine efficient privatizing transactions with uninstrumented nontransactional reads cannot be realized, unless parallelism is compromised. Specifically, the privatizing transaction must incur an inherent cost, linear in the number of data items that are privatized and later accessed with uninstrumented reads.

Our lower bounds do not apply to overly sequential STMs, which achieve efficient privatization by being coarse-grained and allowing only one transaction to make progress at any time [27, 77], thereby significantly reducing throughput. This proof uses a progress assumption that requires the STM to allow concurrent progress of non-conflicting transactions, and a transaction can abort or block only due to a conflicting pending transaction. It is weaker than the property we assume for read-only transactions in the result for disjoint-access parallel implementations described in Section 1.3.

Under this weak progress assumption, we show that *eager* STMs, in which a transaction may update the memory before it is guaranteed to commit, cannot support privatization.

Our proofs only assume a weak safety property that requires a nontransactional read of a data item that is not preceded by any nontransactional write to return the value written by an earlier committed transaction, or the initial value if no such transaction commits. This property follows from *parameterized opacity* [44], regardless of the memory model imposed on nontransactional reads and writes. We show that STMs allowing more than one transaction to proceed concurrently, without using any privatization mechanism, cannot be opaque.

A key factor in many efficient STMs is not having to track the data sets of other transactions, especially if they are not conflicting. We capture this feature by assuming that the STM is *oblivious*, namely, a transaction does not distinguish between nonconflicting transactions. A simple example is provided by STMs using a global clock [84] or counter [31,82], or a decentralized clock [18], in which a transaction cannot tell whether a process p executes a transaction that writes to item x or a transaction that writes to item y , unless it accesses either x or y ; it can observe that the clock or a counter has increased, but this happens in both cases. A less-immediate example is the behavior of TLRW [32] for so-called *slotted* threads. Several other STMs [27, 33, 39, 72, 95] are also oblivious, and are discussed in detail in Chapter 9.

Our first main result further assumes that reads do not write to the memory (invisible reads) and shows that a transaction privatizing k items must have a data set of size $\Omega(k)$. In an oblivious STM with invisible reads, transactions are unaware of, and hence unaffected by, read-read (trivial) conflicts, which simplifies the proof.

Our second main result removes the assumption of invisible reads, and shows an $\Omega(k)$ lower bound on the number of shared memory accesses performed by a privatizing transaction, where k is the minimum between the number of privatized items and the level of parallelism, i.e., the number of transactions guaranteed to make progress concurrently. The proof is more involved and relies on the assumption that the STM provides a significant level of parallelism. This lower bound explains why the *quiescence* mechanism [33, 72, 94], for example, must compromise parallelism in order to support efficient privatization.

Obliviousness generalizes disjoint-access parallelism, discussed before, and our lower bounds hence hold also for disjoint-access parallel STMs.

1.5 Outline of the Thesis

This dissertation is divided into two main parts. The first part presents methodologies for designing concurrent data structures that ensure that an operation delays only close-by operations. We demonstrate these methodologies with concrete implementations of multi-word synchronization (Chapter 3) and list-based data structures (Chapter 4). We also prove their correctness and analyze their efficiency.

The second part proves inherent tradeoffs for transactional memory. We prove a tradeoff between providing disjoint-access parallelism and the cost of transactions that only read data (Chapter 7), and a tradeoff between the cost of supporting non-transactional accesses and the parallelism offered by the TM (Chapter 8).

Each part starts with a chapter that introduces relevant definitions (Chapters 2 and 6) and ends with a survey of the related work (Chapters 5 and 9). Finally, we conclude this dissertation with a discussion of the results and directions for future work, in Chapter 10.

Part I

Methodologies for Highly-Concurrent Data Structures

Chapter 2

Concurrent Data Types

We consider a standard model for shared memory systems [17] in which a finite set of n *asynchronous processes* p_1, \dots, p_n communicate by applying *primitive operations* (abbreviated *primitives*) to *base objects* l_1, \dots, l_m that form the shared memory.

A *configuration* is a complete description of the system at some point in time. It is represented as a vector $C = (q_1, \dots, q_n, u_1, \dots, u_m)$, where q_i is the local state of process p_i and u_j is the value of base object l_j . There is a unique *initial configuration* in which every process is in its initial state and every base object contains its initial value.

An *event* is a computation *step* by a process consisting of local computation and the application of a primitive to base objects, followed by a change to the state of the process. A process acts as a state machine and changes its state according to the value returned at every step; processes are *asynchronous* in the sense that the time between two consecutive steps of a process is arbitrary.

An event can be the application of a simple read or write primitives: $\text{READ}(l)$ returns the value v of base object l ; $\text{WRITE}(l, v)$ sets the value of base object l to v .

In addition to these ordinary primitives, we allow *read-modify-write* primitives, in particular (non-reading) CAS and DCAS: $\text{CAS}(l, \text{exp}, \text{new})$ writes the value new to base object l if its value is equal to exp , and returns a success or failure indication; DCAS is a generalization of CAS taking as argument two independent base objects, two expected values and two new values. If both base objects hold the corresponding expected values, they are assigned the corresponding new values atomically. A success or failure return value indicates whether the replacement occurred (see Figure 2.1). An *access* to base object l is the application of a primitive to l .

A primitive is *nontrivial* if it may change the value of the object, e.g., a write or CAS; otherwise, it is *trivial*, e.g., a read.

An *execution interval* α is a finite or infinite alternating sequence $C_0, \phi_0, C_1, \phi_1, C_2, \dots$, where C_k is a configuration, ϕ_k is an event and the application of ϕ_k to C_k results in

```

boolean CAS(l, exp, new) {
    // Atomically
    if l = exp then
        l ← new

    return TRUE
return FALSE
}

boolean DCAS(l[2], e[2], n[2]) {
    // Atomically
    if l[1] = e[1] and l[2] = e[2] then
        l[1] ← n[1]
        l[2] ← n[2]
    return TRUE
return FALSE
}

```

Figure 2.1: The *compare&swap* and *double compare&swap* primitives.

C_{k+1} , for every $k = 0, 1, \dots$. An *execution* is an execution interval in which C_0 is the initial configuration.

A *solo execution* of process p is an execution interval in which all steps are taken by process p .

Abstract data types (ADT) consist of a set of *high-level abstract data items* (abbreviated *items*), each initialized to an initial value, and a set of *abstract operations* that provide the only means to manipulate these items. In Part I of the thesis, we often abbreviate and refer to abstract operations of an ADT as *operations*.

A prominent example for abstract data types is provided by list-based data structures: A *double-ended queue* (deque) supports operations that insert and remove nodes at the two ends of the queue; it can be used as a producer-consumer job queue [7]. A *priority queue* is a list where removals are allowed only at the ends, while nodes can be inserted anywhere at the queue; it can be used to queue process identifiers for scheduling purposes. Finally, a generic *doubly-linked list* (hereafter, called simply a *linked list*) consists of nodes, each with a backward and forward references; it allows insertions and removals of nodes anywhere in the linked list.

2.1 Safety Property: Linearizability

The *sequential specification* of the ADT indicates how the items are modified when abstract operations are applied in a serial manner (in isolation). It is given as a sequence of *high-level operations* executed on a set of items, which constitute the operation's *data set*. For example, the data set of a remove operation in a doubly-linked list includes the node to be removed and the two nodes before and after it in the linked list.

An *implementation* of some abstract data type, T , provides a specific data representation for the items and abstract operations of T using base objects, and algorithms that processes must follow to execute the operations of T , specified in terms of primitives applied to base objects.

The *interval of an operation* op is the execution interval that starts at the first event of op and ends at the last event of op , if there is one, taken by a process exe-

$op_1 = 3RMW(v_1, v_2, v_6)$
 $op_2 = 2RMW(v_2, v_5)$
 $op_3 = 2RMW(v_1, v_3)$
 $op_4 = 2RMW(v_3, v_4)$
 $op_5 = 2RMW(v_1, v_4)$

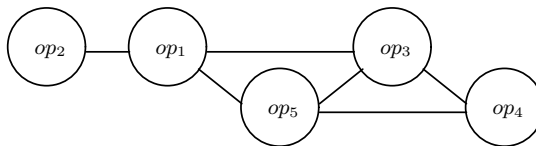


Figure 2.2: The conflict graph of five overlapping operations; data items are shown on left.

cutting the algorithm for op ; if the operation does not complete, its interval is infinite. Two operations *overlap* if their intervals overlap. An operation op is *pending* in some configuration C , if the process executing op has yet taken the last step of op in the execution leading to C .

Two executions are *equivalent* if every process in these executions issues the same operations in the same order and gets the same result for each operation.

An execution is *serial* if no operations overlap; this means that every operation is executed to completion before another operation starts.

An implementation of an ADT is *linearizable* [58] if any execution H can be extended to execution G by discarding some pending operations in H , and completing the others, such that G is equivalent to a serial execution, called its *linearization*, which preserves the order of non-overlapping operations in H ; there might be more than one linearization for an execution.

2.2 Progress and Locality Properties

We use conflicts to capture proximity of operations. Conflicts are defined based on the data set of each operation, such that there is a conflict between two operations if they access the same item. A *conflict graph* formally captures the distance between overlapping operations. Edges in the graph represent conflicts between two concurrent operations, while paths represent the transitive closure of these conflicts. In this chapter, the conflict graph is defined for the case of a static data set that is known in advance. The conflict graph was suggested to model the conflicts of static operations with known data sets. However, some instances of ADTs—and hence the data sets of their operations—are dynamic; they will be addressed in Chapter 4.

In more detail, the *conflict graph* is an undirected graph, where *vertices* represent the abstract operations and *edges* represent conflicts between operations, i.e., an edge connects two operations whose data sets intersect. Figure 2.2 depicts the conflict graph of overlapping k -read-modify-write ($kRMW$) operations that receive k items when they start, and then access and modify them atomically.

The *distance* between two operations, $op \neq op'$, in a conflict graph, is the length (number of edges) of the shortest path between op and op' . The *distance* from an operation to itself is zero. If the operations access a common item, then their distance

is one; if there is no path between the operations, the distance is infinity.

The d -neighborhood of an operation op contains all the operations within distance d from op in the conflict graph of the execution interval of op .

We use the following variant of the failure locality definition [24] suggested by Choy and Singh:

Definition 1 *An implementation has $\langle d_1, d_2 \rangle$ -failure locality if whenever a process takes an infinite number of steps in an operation op , then some operation in the d_1 -neighborhood of op completes, unless a process that issued an operation in the d_2 -neighborhood of op stops taking steps.*

We often abbreviate and say that an algorithm has d failure locality with $d = \max(d_1, d_2)$.

The original definition [24] requires an operation to complete if no operation stops taking steps in its d -neighborhood, while Definition 1 only guarantees that *some* operation in the d -neighborhood completes. This is analogous to the distinction between *starvation-free* and *deadlock-free* mutual exclusion algorithms.

The next definition guarantees progress within the neighborhood of a given diameter from an operation, even when processes executing other operations stop taking steps.

Definition 2 *An implementation is d -local nonblocking if whenever a process takes an infinite number of steps in an operation op , then some operation in the d -neighborhood of op completes.*

Afek et al. [5] present another definition that captures the locality of wait-free implementations. We present the variation of this definition, which is the quantitative variant of the local nonblocking definition:

Definition 3 *An implementation has d -local step complexity if whenever a process takes a finite number of steps in an operation op , which is bounded by a function of d and the number of operations in the d -neighborhood of op , then some operation in the d -neighborhood of op completes.*

Afek et al. [5] also present the local contention definition that captures the locality of implementations in terms of the memory contention they generate:

Definition 4 *An implementation has d -local contention if two processes p_1 and p_2 access the same base object in operations op_1 and op_2 , respectively, only if op_1 and op_2 are within distance d in the conflict graph of the minimal execution interval that contains both the execution interval of op_1 and the execution interval of op_2 .*

Chapter 3

Multi-word Synchronization Operations

The design of concurrent data structures is greatly facilitated by the availability of synchronization operations that atomically modify k arbitrary items, such as k RMW. A k RMW operation *reads* the contents of k data items, computes new values using some *modify* function and *writes* them back to the items. The sequential specifications of the general k RMW operation and the important, special case of a k CAS operation, appear in Figure 3.1.

```
kRMW( $l[k]$ ) {
  // Atomically

  for  $i = 0 \dots k - 1$  do
     $old[i] \leftarrow \text{READ}(l[i])$ 
  modify( $old, new$ )
  for  $i = 0 \dots k - 1$  do
    WRITE( $l[i], new[i]$ )
}

boolean  $kCAS(l[k], exp[k], new[k])$  {
  // Atomically
  res  $\leftarrow$  TRUE
  for  $i = 0 \dots k - 1$  do
    res  $\leftarrow$  res and ( $l[i] = exp[i]$ )
  if res then
    for  $i = 0 \dots k - 1$  do
      WRITE( $l[i], new[i]$ )
    return TRUE
  return FALSE
}
```

Figure 3.1: The k RMW and k CAS operations.

Aiming at increased throughput, we propose a highly-concurrent software implementation of k RMW, with only constant space overhead. Our algorithm ensures that two operations delay each other only if they are within distance $O(k)$ in their conflict graph.

This chapter is organized as follows: Section 3.1 presents BLocalRMW, a *blocking* implementation of k RMW, which has $O(k)$ failure locality, as well as its progress and

⁰The results of this chapter have appeared in [12, 14].

locality proofs. Section 3.2 presents LocalRMW, a $O(k)$ -local nonblocking implementation. This section includes the safety, progress and locality proofs of the algorithm, which extend the proofs of the blocking algorithm. Finally, an implementation for the common case of k CAS is described in Section 3.3.

3.1 BLocalRMW: A Blocking Algorithm with $3k$ Failure Locality

BLocalRMW follows a simple high-level scheme in which an operation tries to acquire all its items, one by one, and applies its changes to these items only while holding all of them.

An important aspect of the algorithm is in handling situations in which an operation op_1 finds that an item it needs is already acquired by another, *blocking* operation op_2 . BLocalRMW uses the number of data items that are already acquired to decide whether op_1 *waits* for op_2 or whether op_1 *resets* op_2 , releasing all the items acquired by op_2 , and seizing the required item. The operation op_1 waits for op_2 only if op_2 is more advanced in acquiring its items. Otherwise, if op_2 has acquired fewer items than op_1 , then op_1 resets op_2 .

An operation resets another operation after acquiring ownership over it. Resetting an operation only involves releasing the items it acquired, and does not involve rolling back its changes, since the operation has not written new values to the items yet.

Another important aspect of BLocalRMW is in handling the *symmetric* case, when op_1 and op_2 have acquired the same number of items, by applying DCAS to atomically acquire ownership of both operations; the operation that acquires ownership, resets the other operation. This breaks apart long hold-and-wait *chains* that would deteriorate the locality as well as hold-and-wait *cycles* that can cause a deadlock.

The high-level scheme appears in Pseudocode 3.1.

The execution of an operation is partitioned into *rounds*, each starting when the operation is reset. At each round, the operation tries to acquire its data items; if the operation is reset, all the items acquired by the operation are released; in the last round, the operation succeeds in acquiring all the items, computes the new values and writes them, and finally, releases all the items.

Rounds are divided into iterations (*executelteration*); at each iteration the operation either tries to make progress (*tryAdvancingOp*), by either acquiring an additional item (*acquireNextItem*) or modifying the data items it has acquired and releasing them, or it handles a conflict (*handleConflict* method). When acquiring an item, the operation increases its counter, otherwise, it discovers that another operation owns the item, and initializes the conflict information, so the conflict can be handled in the next iteration.

Pseudocode 3.1 *k*-Read-Modify-Write: High-level algorithm

```

rmw(Item items[k]) {
  WRITE(dataset, items)
  while READ(modifyDone) = FALSE do
    cnfl ← READ(conflict)
    ctx ← READ(context)
    executeIteration(ctx, cnfl)
}

tryAdvancingOp(Context ctx, Conflict cnfl) {
  if ctx.counter < k then
    acquireNextItem(ctx, cnfl)
  else
    modify()
    WRITE(modifyDone, TRUE)
    releaseDataset(ctx.round)
}

executeIteration(Context ctx, Conflict cnfl) {
  if cnfl.blocking = ⊥ then
    tryAdvancingOp(ctx, cnfl)
  else
    handleConflict(ctx, cnfl)
}

acquireNextItem(Context ctx, Conflict cnfl) {
  if acqOperation(self, ctx, ctx.round) then
    item ← READ(dataset[ctx.counter])
    if acqItem(item, ctx, cnfl) then
      increaseCounter(ctx)
    else
      initializeConflictInfo(ctx, cnfl)
}

```

```

structure Owner {Operation op, int round}
structure Context {int counter, Owner owner, int round}
structure Conflict {int counter, int round, Owner blocking}
class Item {
  Data data
  Owner owner
}

class Operation {
  Item dataset[k]
  Context context
  Conflict conflict
  boolean modifyDone
}

```

Figure 3.2: Data structures and classes for BLocalRMW.

3.1.1 Data Structures

BLocalRMW is derived from the general scheme by substituting the methods for the high-level scheme of Pseudocode 3.1. Its data structures appear in Figure 3.2. Memory locations are grouped into contiguous blocks, called *item objects*. Each item object contains a *data* attribute and an *owner* attribute, including the operation, *op*, and the *round* in which the operation acquired the item; the owner attribute is \perp if no operation has acquired the data item.

For each operation, we maintain an *operation object*¹ containing a *dataset*, referencing the set of items the operation has to access and modify; it is initialized when the operation is first invoked. The *context* is a tuple of a *counter* holding the number of items acquired so far (initially 0), an *owner* and the *round number* of the operation (initially 0). An operation object also contains some local attributes, which are not shared and are only visible to the process executing the operation.² The *modifyDone*

¹These are similar to *transaction descriptors* used in some STM implementations, e.g., [50, 55].

²This will be changed later, when we present LocalRMW.

flag indicates whether or not the modifications of the operation have been applied. The *conflict* stores information about a conflict, if there is any: the owner of the item *blocking* the operation, and the *counter* and *round* values of the blocked operation.

3.1.2 Implementation

The methods of `BLocalRMW` appear in Pseudocode 3.2 and 3.3, described next.

The `handleConflict` method reads the identity of the blocking operation op' (line 2), and its context (line 3). Then it checks whether the round numbers of op or op' changed (line 4), indicating that either one or both of them released the items in its data set and the conflict is resolved.³ If one of the round numbers changes, the method resets the conflict information (line 5) and proceeds to the next iteration (line 6). Otherwise, it compares the counter of op with the counter of op' . If the counter of op is higher than (line 7) or equal to (line 11) the counter of op' , the method tries to reset op' (lines 8, 12) so as to seize the item. For this purpose, op needs to hold the operation objects of both op and op' . When the counter of op is higher than the counter of op' , the method keeps the operation object of op , and tries to acquire the operation object of op' (line 8), using CAS suffices in this case. When the counters are equal, the method releases the operation object of op (line 10) and tries to atomically acquire both operation objects (line 12) by applying DCAS.

The `handleConflict` method releases the operation object of op (line 10) also if the counter of op is lower than the counter of op' , but in this case, it does nothing before the next iteration (line 13). Note that after releasing the operation object, op reacquires the operation object (and resumes the execution of the operation) only when it is no longer blocked by op' (since one of the round numbers changed) or when it is about to reset op' (line 12). This allows other operation to reset op if op is blocking them (see the proof of Lemma 3.5).

The `acqOperation` method acquires ownership on the operation by setting the *owner* attribute to the reserved word *self*, denoting the object of the operation whose code is being executed (line 16). To simplify the pseudocode, as well as the correctness proofs, when acquiring an operation object, CAS is applied on the entire *context* and not just on the *owner* field.

The `reset` method reads the item held by the blocking operation (line 21), and then seizes it by overriding the value written in the item's owner field (line 22). Then, it increases the counter of the (blocked) operation (line 23) and releases the data set of the blocking operation (line 24).

The `releaseDataset` method reads (line 29) and releases (line 30) every item the operation owns (line 28). The context of the operation is reset by setting its counter to 0 its owner to \perp , and increasing the round number (line 31).

³This is similar to the way Shavit and Touitou [91] use the version number to validate the operation.

Pseudocode 3.2 *k*-Read-Modify-Write: methods for BLocalRMW (continued in Pseudocode 3.3)

```

1: handleConflict(Context ctx, Conflict cnfl) {
2:   blkOp  $\leftarrow$  conflict.blocking.op
3:   blkCtx  $\leftarrow$  READ(blkOp.context)
4:   if  $\langle$ ctx.round,blkCtx.round $\rangle \neq \langle$ round,conflict.blocking.round $\rangle$  then
5:     conflict  $\leftarrow$   $\langle \perp, \perp, \perp \rangle$  // reset conflict
6:     return
   // conflict with an operation with a lower counter
7:   if conflict.counter > blkCtx.counter then
8:     if CAS(blkOp.context,  $\langle$ blkCtx.counter, $\perp$ ,blkCtx.round $\rangle$ ,
            $\langle$ blkCtx.counter,self,ctx.round $\rangle$ ,blkCtx.round $\rangle$ )
           then reset(blkOp)
9:     return
10:  WRITE(context,  $\langle$ ctx.counter, $\perp$ ,ctx.round $\rangle$ ) // release this operation object
   // conflict with an operation with an equal counter
11:  if conflict.counter = blkCtx.counter then
12:    if DCAS(context, blkOp.context,
            $\langle$ ctx.counter, $\perp$ ,ctx.round $\rangle$ ,  $\langle$ blkCtx.counter, $\perp$ ,blkCtx.round $\rangle$ ,
            $\langle$ ctx.counter,self,ctx.round $\rangle$ ,ctx.round $\rangle$ ,
            $\langle$ blkCtx.counter,self,ctx.round $\rangle$ ,blkCtx.round $\rangle$ )
           then reset(blkOp)
   // conflict with an operation with a higher counter
13:  if conflict.counter < blkCtx.counter then No-op
14: }

15: boolean acqOperation(Operation op, Context ctx, int rnd) {
16:   CAS(op.context,  $\langle$ ctx.counter, $\perp$ ,ctx.round $\rangle$ ,  $\langle$ ctx.counter,self,rnd $\rangle$ ,ctx.round $\rangle$ )
17:   return (op.ctx.owner  $\neq$   $\perp$  and op.ctx.owner.op = self)
18: }

19: reset(Operation blkOp) { // resetting another blocking operation
20:   ctx  $\leftarrow$  READ(context)
21:   item  $\leftarrow$  READ(dataset[ctx.counter]) // item held by the blocking operation
22:   WRITE(item.owner, self,ctx.round) // seize item
23:   WRITE(context, ctx.counter+1,ctx.owner,ctx.round) // increase counter
24:   blkOp.releaseDataset() // release the data set of the blocking operation
25: }

26: releaseDataset() {
27:   ctx  $\leftarrow$  READ(context)
28:   for  $i = 0 \dots$  ctx.counter-1 do
29:     item  $\leftarrow$  READ(dataset[ $i$ ]) // an item to release
30:     WRITE(item.owner,  $\perp$ ) // release item
   // release operation object, reset counter, increase round number
31:   WRITE(context,  $\langle 0, \perp, \text{ctx.round}+1 \rangle$ )
32: }

```

Pseudocode 3.3 BLocalRMW (continued from Pseudocode 3.2)

```

33: boolean acqItem(Item item, Context ctx, Conflict cnfl) {
34:   CAS(item.owner,  $\perp$ ,  $\langle self, ctx.round \rangle$ ) // acquire the current item
35:   owner  $\leftarrow$  READ(item.owner)
36:   return (owner  $\neq \perp$  and owner.op = self) // item is owned by this operation
37: }

38: increaseCounter(Context ctx) {
39:   WRITE(context,  $\langle ctx.counter+1, ctx.owner, ctx.round \rangle$ )
40: }

41: initializeConflictInfo(Context ctx, Conflict cnfl) {
42:   conflict  $\leftarrow \langle ctx.counter, ctx.round, owner \rangle$ 
43: }

```

3.1.3 An Execution Example

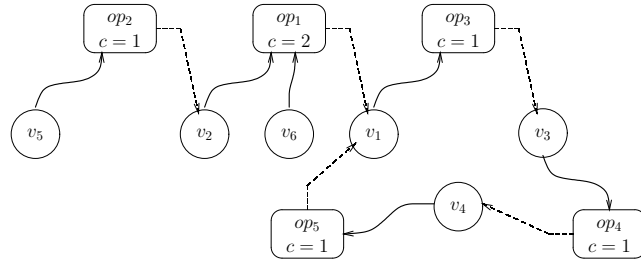
Figure 3.3 shows an example of a chain of conflicts between the operations from Figure 2.2. In Figure 3.3(a) op_2 holds v_5 and needs to acquire v_2 ; op_1 holds v_2 and v_6 and it needs to acquire v_1 ; op_3 , op_4 , and op_5 , hold v_1 , v_3 , and v_4 respectively and need to acquire the successor item in the loop. The counter of op_1 is 2, and the counters of op_2 , op_3 , op_4 , op_5 , are 1. Therefore, the operation op_1 is blocked by an operation op_3 (with a lower counter) and tries to reset op_3 . In Figure 3.3(b) op_1 completes the reset, and acquired all its data items (the counter is 3), and can apply its changes and release the items. Operation op_2 holds v_5 and is blocked by op_1 on v_2 (with a higher counter). Hence, op_2 releases its own operation object, and waits for op_1 to release v_2 .

The conflicts between op_3 , op_4 and op_5 may lead to a cycle of hold-and-wait operations, since they have equal counters. Figure 3.3(c) shows how DCAS breaks the symmetry and avoids a deadlock: the processes release their operation objects; op_3 tries to atomically acquire the operation objects of op_3 and op_4 , op_4 tries to acquire the operation objects of op_4 and op_5 , and op_5 tries to acquire the operation objects of op_5 and op_3 . In Figure 3.3(c), only op_5 succeeds to acquire the operation objects, and it resets op_3 .

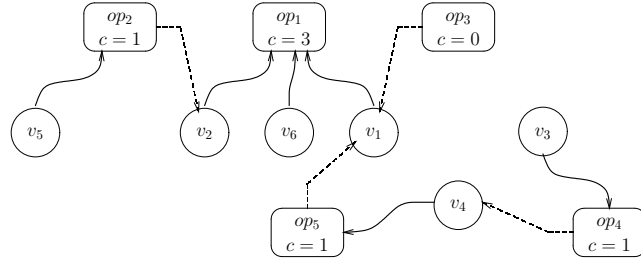
3.1.4 Correctness Proof

The full proof showing that BLocalRMW is linearizable is omitted since it is a simplified version of the safety proof for LocalRMW (see Section 3.2.3). A core issue in this proof is showing that a process owns its own operation object when it has acquired all its items, therefore it is not reset anymore, and its changes are applied in isolation.

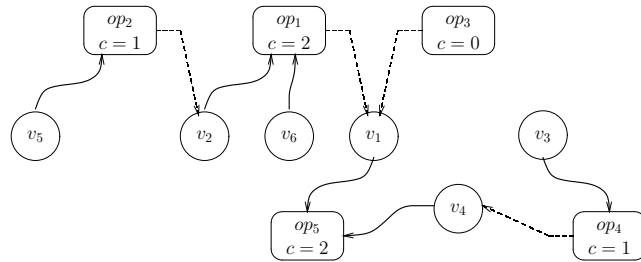
In this section we concentrate on proving the locality of the algorithm by bounding the length of delay chains. One type of delay chain is created when a process



(a) op_1 and op_5 are blocked by op_3 .



(b) A possible scenario after 3.3(a): op_1 resets op_3 .



(c) Another possible scenario after 3.3(a): op_5 resets op_3 .

Figure 3.3: Conflicting operations examples. An operation object points to the next item to be acquired (dashed line); c is the value of its counter. Items are owned by the operation to which they point.

stops taking steps and causes operations of other processes to be blocked; we show, in Lemmas 3.5 and 3.6, that the length of such chains is $O(k)$.

More intricate, and less intuitive, delay chains are created when operations reset other operations. For example, assume an operation op_1 is reset by another operation op_2 , then, a third operation resets op_2 . At some later time, the processes executing op_2 and op_1 can reacquire their operation object, and the same scenario may happen over and over again. It may even seem as if a livelock can happen due to a cycle of resetting operations.

Figure 3.4 illustrates how an operation can be reset many times before some oper-

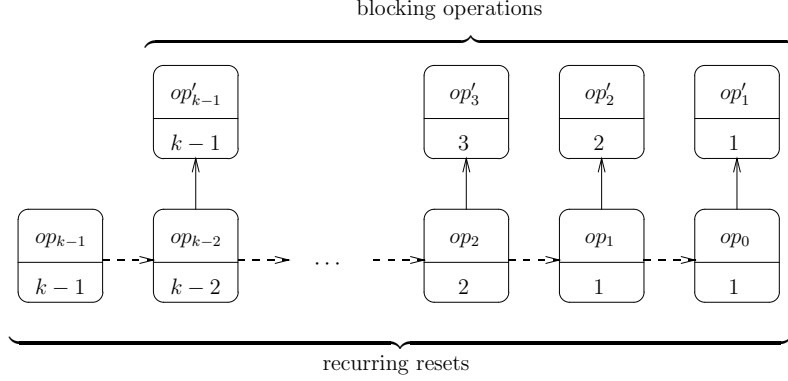


Figure 3.4: A recurring resets scenario. The number below an operation indicates its counter; solid arrows indicate blocking and dashed arrows indicate reset. Initially, for $i \geq 1$, op_i has counter i ; for $i \geq 0$, op_i owns the item op_{i+1} needs next; and op_0 is blocked by op'_1 with equal counter, 1. For $i \geq 1$, after op_i acquires the next item and increases its counter, it is blocked by op'_{i+1} with equal counter $i + 1$.

ation completes in its k -neighborhood. For $i \geq 1$, after op_i acquires an additional item and increases its counter to $i + 1$, it is blocked by op'_{i+1} with equal counter. Initially, op_0 releases its operation object in order to reset op'_1 and seize the item op_0 needs. Instead, op_1 resets op_0 , seizes the item op_0 owns, and increases its own counter to 2. Then, op_1 is blocked by op'_2 with equal counter, 2. In a similar way, op_2 resets op_1 after it releases its operation object, and op_3 resets op_2 , so that their items are released. Later, op_0 and op_1 are able to reacquire their first items. This recurring reset scenario is repeated with longer chains of resets each time.

However, inspecting the example reveals that the longer the chain is, the higher is the counter of its last operation, and after $k/2$ resets of op_0 , op_{k-1} at distance $k - 1$ from op_0 owns all its data items, and it can complete. The following lemmas formalize this intuition. Since an item is seized during a reset, an operation makes progress whenever it resets another operation, as stated in the next lemma. Let $c(op)$ be the counter of op in a given configuration.

Lemma 3.1 *If an operation op_1 resets another operation op_2 , then $c(op_1) = m_1 \geq c(op_2) = m_2$ in the configuration in which the reset is invoked, and $c(op_1) = m_1 + 1$ in the configuration after the reset completes.*

Proof: Before invoking `reset`, op_1 compares the counters of op_1 and op_2 (lines 7 or 11). If $c(op_2) < c(op_1)$ (line 7), then op_1 tries to acquire the operation object of op_2 (line 8). The counter of op_1 does not change while op_1 is holding its operation object. If op_1 acquires the operation object of op_2 , then the counter of op_2 also has not changed since op_1 read it.

If the counters are equal (line 11), then op_1 acquires the operation objects of op_1

and op_2 (line 12), before resetting op_2 (line 12). Similar arguments imply that the counters do not change.

In reset, op_1 increases its counter (line 23), and thus, $c(op_1) = m_1 + 1$ when the reset is completed. ■

Lemmas 3.2–3.4 prove that whenever an operation op is reset, some operation in its neighborhood makes progress, e.g., increasing the counter or completing, and that after op is reset a bounded number of times, some operation in its neighborhood completes. For this purpose, we define a dynamic set of operations, \mathcal{R}_{op} . Whenever an operation in the $(k-1)$ -neighborhood of op completes, \mathcal{R}_{op} is set to $\{op\}$. Whenever an operation $op_j \in \mathcal{R}_{op}$ is reset by an operation op_i , op_j is removed from \mathcal{R}_{op} , unless it is op , and op_i is added to \mathcal{R}_{op} (if it is not already in \mathcal{R}_{op}).

Lemma 3.2 *Every operation op_j , other than op , in \mathcal{R}_{op} is at distance $\leq c(op_j) - 1$ from op .*

Proof: The proof is by induction on the length of the execution interval of op . The base case is when the operation starts; in this case, \mathcal{R}_{op} includes only op and the claim vacuously holds.

For the induction step, assume that the claim holds for some prefix of op 's execution interval, and consider first a step that does not change \mathcal{R}_{op} . If the counter of some operation in \mathcal{R}_{op} increases, then the lemma holds by the inductive assumption. \mathcal{R}_{op} does not change, and by definition, no operation in the $(k-1)$ -neighborhood completes. By the inductive assumption, all operations in \mathcal{R}_{op} are in the $(k-1)$ -neighborhood of op , thus no operation in \mathcal{R}_{op} completes. Since no operation in \mathcal{R}_{op} is reset, the counters of operations in \mathcal{R}_{op} do not decrease.

Consider now a step that changes \mathcal{R}_{op} . If some operation in the $(k-1)$ -neighborhood completes and \mathcal{R}_{op} is set to $\{op\}$, then the claim vacuously holds. Otherwise, an operation op_j resets an operation $op_i \in \mathcal{R}_{op}$; $c(op_i) = m \geq 1$ before the reset, since op_i owns at least one item. By the inductive assumption, op_i is at distance $\leq m - 1$ from op . Therefore, op_j is at distance $\leq m$ from op , and by Lemma 3.1, $c(op_j) > m$ after the reset, and the lemma holds. ■

The *potential vector* of \mathcal{R}_{op} in a configuration C is a vector with n entries holding the counters of operations in \mathcal{R}_{op} . Assume \mathcal{R}_{op} contains r operations, then entries $0 \dots r - 1$ hold the counters of the operations in \mathcal{R}_{op} in decreasing order, and entries $r \dots n - 1$ are all 0's. Note that the entries of the vector are reordered every time the counters are changed, so that all 0's are shuffled to the end of the vector, and other counters appear in decreasing order at the beginning of the vector. We compare vectors in lexicographic order.

Lemma 3.3 *The potential vector of \mathcal{R}_{op} grows with each reset of op , and it does not decrease unless some operation within op 's $(k - 1)$ -neighborhood completes.*

Proof: Assume that no operation within the $(k - 1)$ -neighborhood of op completes. By Lemma 3.2, the operations in \mathcal{R}_{op} are in the $(k - 1)$ -neighborhood of op , and hence, none of them completes.

If the counter of some operation op' in \mathcal{R}_{op} increases, then the potential vector grows.

Consider an operation, op'' , resetting an operation $op' \in \mathcal{R}_{op}$, such that $c(op') = m'$. If op' is not removed from \mathcal{R}_{op} (since it is op), and op'' is added to \mathcal{R}_{op} , then the potential vector of \mathcal{R}_{op} has an additional nonzero entry. Since the vector holds the counters of the operations in \mathcal{R}_{op} in decreasing order, the potential vector grows.

If op' is not removed from \mathcal{R}_{op} and op'' is already in \mathcal{R}_{op} then \mathcal{R}_{op} does not change. Lemma 3.1 implies that $c(op'')$ increases after the reset, and the potential vector grows.

If op' is removed from \mathcal{R}_{op} and op'' replaces op' in \mathcal{R}_{op} , then Lemma 3.1 implies that after the reset $c(op'') > m'$, and the potential vector grows.

Finally, if op' is removed from \mathcal{R}_{op} and op'' is already in \mathcal{R}_{op} , then the entry of op' in the vector is set to 0, and no additional nonzero entry is added to the vector. However, by Lemma 3.1, before the reset $c(op'') \geq c(op')$, and $c(op'')$ increases after the reset. Thus, some entry to the left of op' in the potential vector increases, and the potential vector grows (since entries are reordered to appear in decreasing order). ■

A *progress step* of an operation op is a step in which either op completes or increases its counter. Let n_d be the number of operations in the d -neighborhood of op . By Lemma 3.2, all the operations in \mathcal{R}_{op} are in the $(k - 1)$ -neighborhood of op , thus the size of \mathcal{R}_{op} is at most n_{k-1} . The next lemma uses Lemma 3.3 to show that after a bounded number of progress steps of op , some operation in its $(k - 1)$ -neighborhood completes.

Lemma 3.4 *After at most $n_{k-1}k^2$ progress steps of an operation op , some operation completes in its $(k - 1)$ -neighborhood.*

Proof: If no operation completes in the $(k - 1)$ -neighborhood of op (including op), then after $n_{k-1}k(k - 1)$ progress steps, op increases its counter $n_{k-1}k(k - 1)$ times, which means it is reset at least $n_{k-1}k$ times. By Lemma 3.3, after each reset of op , the potential vector of \mathcal{R}_{op} increases. By the time the vector increases $n_{k-1}k$ times, all the operations in the $(k - 1)$ -neighborhood of op (at most n_{k-1} operations, including op) have their counters equal to k and hereafter these operations do not release their operation objects. On one hand, the operations in this neighborhood cannot increase their counters, on the other hand, no operation resets an operation in this neighborhood, which implies that op completes in its $n_{k-1}k(k - 1) + 1 \leq n_{k-1}k^2$ progress step. ■

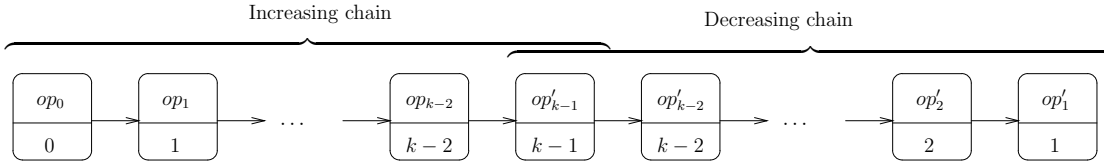


Figure 3.5: The operations op_j create an *increasing chain*; each op_j operation is blocked by the operation op_{j+1} holding the data item t_{i+1} . The operations op'_i create a *decreasing chain*; each op'_i operation is blocked by the operation op'_{i-1} holding its operation object.

This shows that recurring resets do not prevent progress in the neighborhood of an operation. We now discuss delay chains created when processes stop taking steps.

Consider an operation op that stops taking steps while holding a data item. Another operation op' requiring this item cannot complete without resetting op , which it fails to do if op holds its operation object. If $c(op') > c(op)$, op' keeps its operation object and may block a third operation with a higher counter that cannot reset op' , and so on.

Formally, an operation op is m -delayed by another operation op' when op needs to reset op' , $c(op') \leq m$, and some operation other than op has acquired the operation object of op' . In a *decreasing chain* of operations an operation op is m -delayed by the next operation where $m < c(op)$. Since each operation in a decreasing chain has a counter that is strictly lower than the counter of the operation that it blocks, the length of the chain is bounded by k .

The right part of Figure 3.5 presents an example of a decreasing chain. The operation op'_2 needs to acquire an item, and is blocked by another operation op'_1 with a lower counter. The operation op'_1 does not complete either because it stops taking steps or because it is repeatedly being reset by other operations, and the operation object of op'_1 is always reacquired before op'_2 has a chance to reset op'_1 and acquire its next item. For every i , $1 \leq i < k - 1$, the operation op'_{i+1} is i -delayed by the operation op'_i , since some operation holds the operation object of op'_i while it is blocking op'_{i+1} .

We use the notion of a *big step* [26] to prove that the algorithm has $O(k)$ failure locality. A big step is a measure of time for asynchronous systems. This notion is usually applied to all the processes in the system, and we refine it to describe the progress of a subset of the processes. An execution interval α contains a *big step* of some set of operations S if each operation in S takes at least one step in α . Inductively, an execution interval α contains $s > 1$ *big steps* of S if α can be written as $\alpha'\alpha''$, α' contains $s - 1$ big steps of S , and α'' contains a big step of S .

Let l be a constant that is larger than the number of steps a process takes in a single invocation of `executelteration`.

Lemma 3.5 *Consider an operation op that is m -delayed by another operation op' in a configuration C , $1 \leq m \leq k - 1$. Let \mathcal{DC} be a decreasing chain from op' . Then, in an*

execution interval α starting in C that contains $2l$ big steps of the processes $\{op\} \cup \mathcal{DC}$, one of the following happens:

1. some operation in the $m + 2$ -neighborhood of op has a progress step, or
2. some operation op'' holds an operation object of another operation in \mathcal{DC} .

Proof: The proof is by induction on m , with a base case at $m = 1$. The operation op is 1-delayed by op' , namely when op tries to reset op' , $c(op') = 1$ and some operation has acquired the operation object of op' . In this case, $\mathcal{DC} = \{op'\}$.

If op' increases its counter, during α , then Condition 1 holds. If some operation $op'' \neq op'$ holds the operation object of op' , during α , then Condition 2 holds. Thus, assume op' holds its own operation object and it is blocked by another operation op'' , and $c(op'') \geq c(op')$ (op' cannot be blocked by an operation with a lower counter, since $c(op') = 1$).

The execution interval α contains $2l$ big steps of op and op' , during which op' releases its own operation object and some operation reacquires its operation object. If this time, another operation $op'' \neq op'$ acquires the operation object, then Condition 2 holds. Otherwise, op' re-acquires its own operation object. If op' re-acquires its operation object after op'' has completed, then Condition 1 holds. If op' acquires both the operation objects of op' and op'' , then op' resets op'' and increases its counter, during α . Otherwise, op' re-acquires the operation object either after op'' is reset and is no longer blocking op' , or after op' is reset and is no longer blocked by op'' . This implies that during α , after op' re-acquires its own operation object (after releasing it) some operation (possibly, op' itself) in the 1-neighborhood of op' either completes or is reset, increasing the counter of the resetting operation. Thus, some operation in the 2-neighborhood of op (op' , or op'') completes or is reset once during α , and some operation in the 3-neighborhood of op has a progress step and Condition 1 holds.

For the induction step, assume the lemma holds for every operation with counter lower than $m > 1$. If an operation other than op' holds the operation object of op' , then Condition 2 holds, so assume op' holds its own operation object.

If op' is m' -delayed by an operation op'' , $m' < m$, in C , then consider the decreasing chain \mathcal{DC}' from op'' in C . If some operation in the $m + 1$ -neighborhood of op' (and the $m + 2$ -neighborhood of op), has a progress step in α , then Condition 1 holds. Otherwise, α contains $2l$ big steps of $\{op'\} \cup \mathcal{DC}' \subset \{op\} \cup \mathcal{DC}$, and by the inductive assumption, some operation owns another operation object in \mathcal{DC}' , satisfying Condition 2.

Finally, if op' is m' -delayed by an operation op'' , $m' \geq m$, then, as in the base case, some operation in the 3-neighborhood of op has a progress step during α , satisfying Condition 1. ■

Another blocking scenario happens when an operation op is blocked by an operation op' whose counter is higher than $c(op)$. Moreover, op' may be blocked by a third

operation with a higher counter, and so on, creating an *increasing chain*, also depicted in Figure 3.5 (left side). Since the counter of every operation in an increasing chain is strictly larger than the counter of the operation that it blocks, the length of the chain is at most k .

Decreasing and increasing chains, together with recurring resets may create longer delay chains. In an *increasing-decreasing chain* of operations, every operation op is blocked either by the next operation in the chain, op' with $c(op') > c(op)$, or by a decreasing chain; in the later case, op is m -delayed by the head of the decreasing chain, $m \leq c(op)$.

Lemma 3.6 *Consider an operation op with $c(op) = m$ in a configuration C , $0 \leq m \leq k$. Let \mathcal{IDC} be an increasing-decreasing chain that includes op . Then, in an execution interval α starting in C that contains $4l$ big steps of \mathcal{IDC} , one of the following happens:*

1. *Some operation in \mathcal{IDC} acquires all its data items, or*
2. *Some operation in the $(2k - m)$ -neighborhood of op has a progress step, or*
3. *Some operation op'' owns the operation object of another operation $op' \in \mathcal{IDC}$.*

Proof: The proof is by backward induction on m . In the base case, $m = k$, op acquired all its data set, and Condition 1 holds. For the induction step assume $m < k$, and consider what happens when op executes $2l$ steps, in which it completes at least two iterations of the loop in the `rmw` method. We inspect the code.

If there is no conflict, then, since the counter of op is not equal to the size of its data set, op tries to acquire its own operation object (line 16). If another operation holds the operation object of op , then Condition 3 holds.

Otherwise, op holds its operation object (line 16 or line 17), and it tries to acquire another item (line 34). Then, op reads the owner of the current item (line 35). If no operation owns the item (line 36), then op may fail to acquire the item since another operation op' owned it but op' no longer owns the item. Hence, op' either completes or is reset, increasing the counter of the resetting operation in the 2-neighborhood of op and Condition 2 holds. If op owns the current data item (line 36), then the counter of op increases (line 39) and Condition 2 holds.

Otherwise, op is blocked by op' ; it sets the *conflict* information of the operation object (line 42), and starts a new iteration, in which it handles the conflict. In this iteration, op reads the *context* of op' (line 3). If the round number of op' has changed (line 4), then op' either completes or is reset, increasing the counter of the resetting operation in the 2-neighborhood of op and Condition 2 holds.

If $c(op) > c(op')$ (line 7), then op tries to acquire the operation object of op' (line 8) and reset it (line 8).

If $c(op) = c(op')$, op releases its own operation object (line 10) and tries to acquire the operation objects of both op and op' and reset op' (line 12).

If $c(op) \geq c(op')$ and op acquires the operation object of op' then Condition 3 holds. If op fails to acquire the operation objects of op' , then either some operation other than op holds the operation object of op , satisfying Condition 3, or op is m' -delayed by op' , $m' \leq m$, in a configuration C' after $2l$ big steps of \mathcal{IDC} . In the latter case, if some operation in the $(2k - m)$ -neighborhood of op has a progress step in α , then Condition 2 holds. Otherwise, the suffix of α , α' starting at C' includes $2l$ big steps of the decreasing chain from op' (which is contained in \mathcal{IDC}). Hence, Lemma 3.5 implies that either some operation in the $m + 2$ -neighborhood (included in the $(2k - m)$ -neighborhood) of op , has a progress step in α' , satisfying Condition 2, or some operation owns another operation on the decreasing chain from op' , satisfying Condition 3.

If $c(op) < c(op')$ (line 13), the counter of op' is $m' > m$. If some operation in the $(2k - m)$ -neighborhood of op has a progress step in α , then Condition 2 holds. Otherwise, α includes $4l$ big steps of the increasing-decreasing chain from op' (which is contained in \mathcal{IDC}), and the lemma holds by the inductive assumption. \blacksquare

After an operation acquires all its items, it holds its operation object until it completes. Hence, other operations cannot reset its counter, and the operation releases its items within $2l$ steps. Thus, if no operation in an increasing-decreasing chain from op (in its $2k$ -neighborhood) stops taking steps, then some operation op' in this neighborhood makes enough progress steps and by Lemma 3.4, some operation in the k -neighborhood of op' and $3k$ -neighborhood of op completes. This argument is formalized in the proof of the next lemma.

Theorem 3.7 (Failure locality) *BLocalRMW has $\langle 3k, 2k \rangle$ -failure locality.*

Proof: Assume that the counter of an operation op is m at some configuration C_1 in its execution interval, and consider an increasing-decreasing chain \mathcal{IDC}_1 from op in C_1 . Consider also the minimum execution interval α_1 starting in C_1 that contains $4l$ big steps of \mathcal{IDC}_1 followed by either $2l$ steps of an operation op' on \mathcal{IDC}_1 , which acquired all its data items, or $2l$ steps of an operation op'' that owns another operation on \mathcal{IDC}_1 . Since the length of an increasing-decreasing chain is at most $2k - 3$, and no process stops taking steps in the $2k$ -neighborhood of op , an execution interval α' that contains $4l$ big steps of \mathcal{IDC}_1 exists. If some operation op' in \mathcal{IDC}_1 acquired all its data items after α' , then op' (which is in the $(2k - 3)$ -neighborhood of op) completes within $2l$ steps. Alternatively, if some operation op'' owns another operation on \mathcal{IDC}_1 after α' , then op'' (which is in the $(2k - 2)$ -neighborhood of op) increases its counter within $2l$ steps. Otherwise, by Lemma 3.6, some operation in the $2k$ -neighborhood of op has a progress step, in α_1 .

If no operation in the $3k$ -neighborhood of op completes during α_1 , then we consider an increasing-decreasing chain \mathcal{IDC}_2 from op in the configuration after α_1 , and an

execution interval α_2 from this configuration, which contains a progress step of some operation in the $2k$ -neighborhood of op .

In this manner, we define execution intervals $\alpha_3, \alpha_4, \dots$. Since there are at most n_{2k} operations in the $2k$ -neighborhood of op , after at most $n_{2k}n_{k-1}k^2$ execution intervals, one of these operations op' has $n_{k-1}k^2$ progress steps. Lemma 3.4 implies that some operation in the k -neighborhood of op' completes, hence, some operation in the $3k$ -neighborhood of op completes. ■

3.2 LocalRMW: A $3k$ -Local Nonblocking Algorithm

LocalRMW is a variant of BLocalRMW that uses *helping* to ensure progress in the $3k$ -neighborhood of an operation, even when processes stop taking steps. Helping means that when an operation op is blocked by another operation op' with a higher counter, its invoking process p executes op' to ensure it completes and releases the item, instead of waiting for the process p' that invoked op' to do so by itself (which may never happen if p' stops taking steps); we say that p *helps* op' or that op *helps* op' . Helping is *recursive* and if p discovers that op' is blocked by another operation op'' , then p also helps op'' . Note that op still resets op' if its counter is equal or higher than the counter of op' . In addition, op can be blocked while trying to acquire an operation object; in this case as well, p helps the blocking operation.

In the example of Figure 3.3, when op_2 discovers that it is blocked by op_1 with a higher counter, op_2 helps op_1 instead of waiting for it to complete. If op_1 is blocked by an operation that owns the operation object of op_3 , while op_1 tries to reset op_3 , then op_2 helps the blocking operation to proceed until it releases the operation object of op_3 .

Due to helping, an operation may be executed by several *executing processes*, simultaneously. The process that invoked the operation is called its *initiator*.

3.2.1 Data Structures

Since there are several processes executing an operation, some changes are required in the data structures of BLocalRMW and the way they are handled, to ensure that only one executing process performs each step of the operation.

The modified data structures appear in Figure 3.6. The most important change is that *modifyDone* flag and the *conflict* attribute are visible to all processes executing the operation, i.e., kept in the shared memory. In addition, the CAS and DCAS primitives suffer from the *ABA problem* [59]; namely, a process p may read a value A from some memory location l , then other processes change l to B and then back to A, later p applies CAS on l and the comparison succeeds whereas it should have failed. We avoid this problem by associating a monotonically increasing *ABA-prevention counter* with attributes that may hold the same value during an execution. This is the case for the

```
structure Owner {Operation op, int round, int aba}
structure Context {int counter, Owner owner, int round}
structure Conflict {int counter, int round, Owner blocking, int aba}

class Item {
    Data data
    Owner owner
}

class Operation {
    Item dataset[k]
    Context context
    Conflict conflict
    boolean modifyDone
}
```

Figure 3.6: Data structures and classes for LocalRMW.

owner attribute, which is reset when the object is released, and for conflict information, which is reset when the conflict is solved. The attribute and the counter fit into a single memory location and are manipulated atomically; the counter is incremented whenever the attribute is updated. Assuming that the counter has enough bits, the CAS (DCAS) succeeds only if the counter has not changed since the process read the attribute.

3.2.2 Implementation

LocalRMW combines the high-level scheme of Pseudocode 3.1 with the methods of Pseudocode 3.4 and 3.5. The main difference from BLocalRMW is that when an operation op is blocked by another operation op' with higher counter, op helps op' to proceed and release its data items (line 64). Additionally, if an executing process p of op tries to acquire the operation object of op' in order to reset op' and discovers it is owned by an operation op'' , then p helps op'' (line 112).

The `handleConflict` method handles the case where op is blocked by another operation op' , indicated by the `conflict` attribute (line 45). The method reads the conflict information and the context of op' (line 46 and line 47). Then the method verifies that neither op nor op' changed round (line 48), as in BLocalRMW. If one of the round numbers changed, the method invalidates the context (line 49), so the operation object is not released by an executing process of op that is not aware that the conflict is resolved. Then the method resets the conflict information (line 50). If both round numbers did not change, the method compares the counter of op with the counter of op' and handles the conflict as in BLocalRMW; unless the counter of op' is higher than the counter of op , in which case, op helps op' (line 64).

The `acqOperation` method applies CAS to acquire the operation object given as input parameter (line 67), and verifies that this operation owns the given operation object (line 68). The `acqTwoOperations` method is similar, except for using DCAS to acquire the *two* operation objects (line 71), and verifying that this operation owns *both* operations object (line 72).

The `reset` method seizes the item held by the blocking operation (line 76) after

Pseudocode 3.4 LocalRMW (continued in Pseudocode 3.5)

```
44: handleConflict(Context ctx, Conflict cnfl) {
45:   blkOp  $\leftarrow$  cnfl.blocking.op
46:   blkCnfl  $\leftarrow$  READ(blkOp.conflict)
47:   blkCtx  $\leftarrow$  READ(blkOp.context)
48:   if  $\langle$ ctx.round,blkCtx.round $\rangle \neq \langle$ cnfl.round,cnfl.blocking.round $\rangle$  then
49:     CAS(op.context, ctx,
          (ctx.counter, $\langle$ ctx.owner.op,ctx.owner.round,ctx.owner.aba+1 $\rangle$ ,ctx.round))
50:     CAS(conflict, cnfl,  $\langle \perp, \perp, \perp, \text{cnfl.aba}+1 \rangle$ ) // reset conflict
51:     return
    // conflict with an operation with a lower counter
52:   if cnfl.counter > blkCtx.counter then
53:     if acqOperation(blkOp, blkCtx,ctx.round) then
54:       reset(cnfl.counter, cnfl.blocking,  $\langle$ self,ctx.round,ctx.owner.aba $\rangle$ )
55:     return
56:   if  $\langle$ ctx.owner.op,ctx.owner.round $\rangle = \langle$ self,ctx.round $\rangle$  then
57:     if  $\langle$ blkCtx.owner.op,blkCtx.owner.round $\rangle \neq \langle$ self,ctx.round $\rangle$  then
58:       CAS(context, ctx,  $\langle$ ctx.counter, $\langle \perp, \perp, \text{ctx.owner.aba}+1 \rangle$ ,ctx.round $\rangle$ )
59:     return
    // conflict with an operation with an equal counter
60:   if cnfl.counter = blkCtx.counter then
61:     if acqTwoOperations(self, ctx, blkOp, blkCtx, ctx.round) then
62:       reset(cnfl.counter, cnfl.blocking,  $\langle$ self,ctx.round,ctx.owner.aba $\rangle$ )
    // conflict with an operation with a higher counter
63:   if cnfl.counter < blkCtx.counter then
64:     blkOp.executeIteration(blkCnfl, blkCtx)
65: }

66: boolean acqOperation(Operation op, Context ctx, int rnd) {
67:   CAS(op.context,  $\langle$ ctx.counter,  $\langle \perp, \perp, \text{ctx.owner.aba} \rangle$ ,ctx.round $\rangle$ ,
        (ctx.counter, $\langle$ self,rnd,ctx.owner.aba+1 $\rangle$ ,ctx.round))
68:   return op.verifyOwner(self, rnd, ctx.round)
69: }

70: boolean acqTwoOperations(Operation op1, op2, Context cx1, cx2, int rnd) {
71:   DCAS(op1.context, op2.context,
        (cx1.counter, $\langle \perp, \perp, \text{cx1.owner.aba} \rangle$ ,cx1.round),
        (cx2.counter, $\langle \perp, \perp, \text{cx2.owner.aba} \rangle$ ,cx2.round),
        (cx1.counter, $\langle$ self,rnd,cx1.owner.aba+1 $\rangle$ ,cx1.round),
        (cx2.counter, $\langle$ self,rnd,cx2.owner.aba+1 $\rangle$ ,cx2.round))
72:   return op1.verifyOwner(self, rnd, cx1.round) and op2.verifyOwner(self, rnd, cx2.round)
73: }

74: reset(int ctr, Owner blocking, Owner owner) {
75:   item  $\leftarrow$  READ(dataset[ctr])
76:   item.acquire(blocking, owner.op, owner.round, ctr) // seize the item
77:   itmOwner  $\leftarrow$  READ(item.owner)
78:   if  $\langle$ itmOwner.op, itmOwner.round $\rangle = \langle$ owner.op, owner.round $\rangle$  then
79:     CAS(context,  $\langle$ ctr,owner,owner.round $\rangle$ ,  $\langle$ ctr+1,owner,owner.round $\rangle$ )
80:   blocking.op.releaseDataset(blocking.round)
81: }
```

Pseudocode 3.5 LocalRMW (continued from Pseudocode 3.4)

```
82: releaseDataset(int rnd) {
83:   ctx ← READ(context)
84:   for  $i = 0$  to  $dataset.size-1$  do
85:     item ← READ(dataset[i])
86:     itmOwner ← READ(item.owner)
87:     CAS(item.owner, ⟨self,rnd,itmOwner.aba⟩, ⟨⊥,⊥, itmOwner.aba+1⟩)
      // increase round number, reset counter, release operation object
88:     CAS(context, ⟨ctx.counter,ctx.owner,rnd⟩, ⟨0,⟨⊥,⊥,ctx.owner.aba+1⟩,rnd+1⟩)
89: }

90: boolean acqItem(Item item, Context ctx, Conflict cnfl) {
91:   owner ← READ(item.owner)
92:   if owner = ⊥ then // item has no owner
93:     if CAS(conflict, ⟨⊥,⊥,⊥, cnfl.aba⟩, ⟨⊥,⊥,⊥, cnfl.aba+1⟩) then
94:       item.acquire(owner, self, ctx.round, ctx.counter) // acquire the item
95:       owner ← READ(item.owner)
96:       return (owner != ⊥ and owner.op = self and owner.round = ctx.round)
97: }

98: increaseCounter(Context ctx) {
99:   CAS(context, ctx, ⟨ctx.counter+1,ctx.owner,ctx.round⟩)
100: }

101: initializeConflictInfo(Context ctx, Conflict cnfl) {
102:   CAS(conflict, ⟨⊥,⊥,⊥, cnfl.aba⟩, ⟨ctx.counter,ctx.round,owner,cnfl.aba+1⟩)
103: }

104: boolean verifyOwner(Operation op, int opRnd, int rnd) {
105:   ctx ← READ(context)
106:   ownerOp ← ctx.owner.op
107:   if ⟨ownerOp,ctx.owner.round⟩ = ⟨op,opRnd⟩ and ctx.round = rnd then
108:     return TRUE // indicate success
109:   if ownerOp != ⊥ and ctx.round = rnd then
110:     ownerCnfl ← READ(ownerOp.conflict)
111:     ownerCtx ← READ(ownerOp.context)
112:     ownerOp.executeIteration(ownerCnfl,ownerCtx) // help the owner
113:   return FALSE // indicate failure
114: }

// Item's method
115: acquire(Owner ownr, Operation op, int rnd, int ctr) {
116:   ctx ← READ(op.context)
117:   if ⟨ctx.counter,ctx.owner.op,ctx.owner.round,ctx.round⟩ = ⟨ctr,op,rnd,rnd⟩ then
118:     CAS(owner, ownr, ⟨op,rnd,ownr.aba+1⟩) // acquire the item
119: }
```

reading it (line 75). If the item is owned by the operation in round number as specified in the input parameter (line 78), then the method increases the counter of the operation (line 79) and releases the data set of the blocking operation (line 80).

As in `BLocalRMW`, the `releaseDataset` method reads (line 86) and releases (line 87) every item that is owned by the operation, and then resets the context of the operation (line 88). However, in order to avoid modifications when the operation changes its round, the round number given as input parameter is used when releasing the items and updating the context of the operation.

The `acqItem` method checks if the current item is released (line 92) and tries to acquire it (line 94). Before doing so, it invalidates the conflict information, by “touching” the ABA value (line 93), so it is not set by another executing process that encountered contention on the same item. Finally, if the item is owned by the operation, the method returns success (line 96).

3.2.3 Safety Proof

We prove that `LocalRMW` is linearizable by showing that the executing processes are synchronized. We identify, for each operation, a *linearization point* in its interval, so that the operation appears to occur atomically at this point. The linearization point of an operation is when it sets the *modifyDone* flag to `TRUE`.⁴ A concrete implementation of the *kRMW* operation defines the `modify` method, and must ensure that the data items are changed only if the *modifyDone* flag of *op* is not `TRUE`, i.e., before the linearization point. Section 3.3 presents an example of `modify` method for the common and important case of *kCAS*.

We first provide some definitions required for the proofs. An operation *op* is in a $\langle c, l, r \rangle$ -context if the *context* of *op* is $\langle c, l, r \rangle$. An operation is in the *r*-th round if it is in a $\langle c, l, r \rangle$ -context for some *c* and *l*. An item object *t* or an operation object *o* are $\langle op, r \rangle$ -owned if the *owner* of *t* or the *context.owner* of *o* respectively, are equal to $\langle op, r, aba \rangle$, for some *aba*. A process *p* $\langle op, r \rangle$ -acquires an item *t* (operation object *o*) if *p* applies a CAS to *t* (*o*) such that prior to applying the CAS *t* (*o*) is not $\langle op, r \rangle$ -owned, and after which *t* (*o*) is $\langle op, r \rangle$ -owned. Similarly, we define the process that $\langle op, r \rangle$ -releases an item (operation object).

We prove that an operation applies its changes exactly once after acquiring all its data items; this claim is used in the proof of Lemma 3.9.

Lemma 3.8 *For every i , $0 \leq i < k$, l and r , when op shifts from the $\langle i, l, r \rangle$ -context to the $\langle i + 1, l, r \rangle$ -context, op is $\langle op, r \rangle$ -owned.*

Proof: An executing process *p* increases the counter of *op* either in line 99, or in line 79 after resetting another operation. In the first case, *p* increases the counter

⁴This is similar to how linearization points are defined in [8].

(line 99) only if op is $\langle op, r \rangle$ -owned. In the second case, one of the input parameters for the `reset` method is a pair, which contains the operation object of op and its round number r , as p previously read them; then, p shifts op to the $\langle i + 1, l, r \rangle$ -context, only if it is $\langle op, r \rangle$ -owned.

In both cases, the CAS is successful only if op is $\langle op, r \rangle$ -owned and the ABA value read is not changed, implying the lemma. \blacksquare

Lemma 3.9 *The following properties hold for every $r \geq 0$, i , $0 \leq i < k$, and every process p :*

1. *If op is in a $\langle i, l, r \rangle$ -context and p $\langle op, r \rangle$ -releases an item then op is $\langle op', r' \rangle$ -owned and $op' \neq op$.*
2. *If op is in a $\langle i, \langle op, r, a \rangle, r \rangle$ -context, then the j -th item of op is $\langle op, r \rangle$ -owned, for every j , $0 \leq j < i$.*
3. *If p $\langle op, r \rangle$ -acquires an item then op is in the r -th round and is $\langle op, r \rangle$ -owned.*

Proof: The proof is by induction on the execution order. The base case is the empty execution where all the properties are vacuously satisfied. Next we prove the induction step for each property; we only review steps that are relevant to the properties. We say that an owner $\langle cop, cr, a \rangle$ is *blocking op in its r -th round on the i -th item of op* if the *conflict* of op is equal to $\langle i, r, \langle cop, cr, a \rangle, aba \rangle$, for some aba value.

Property 1: An executing process $\langle op, r \rangle$ -acquires an item only if it is in the data set of op . By Property 3, an executing process $\langle op, r \rangle$ -acquires the item only if op is in the r -th round. Since $i < k$, p does not apply the `releaseDataset` method in the `execute` method of op . Thus, p $\langle op, r \rangle$ -releases an item t in the `reset` method of another operation op' . Prior to $\langle op, r \rangle$ -releasing t (lines 54 or 62), p acquires op (lines 53 or 61) and verifies that op is in the r -th round and is $\langle op', r' \rangle$ -owned (line 107).

If another executing process, p' , $\langle op', r' \rangle$ -releases op then it applied line 88, and op is no longer in the r -th round. Prior to this, an item, t , in op 's data set is either not $\langle op, r \rangle$ -owned or p' $\langle op, r \rangle$ -releases t (line 87). Whether op is $\langle op', r' \rangle$ -owned or is not in the r -th round, Property 3 implies that no executing process $\langle op, r \rangle$ -acquires any item. Thus, p fails to $\langle op, r \rangle$ -release t (in line 76 or 80).

Property 2: An executing process p increases the counter of op and shifts to the $\langle i, \langle op, r, a \rangle, r \rangle$ -context either in line 99 or in line 79 after resetting another operation. Since the CAS is successful, op applies the CAS in a $\langle i - 1, \langle op, r, a \rangle, r \rangle$ -context. By the inductive assumption for every j , $0 \leq j < i - 1$, the j -th item of op is $\langle op, r \rangle$ -owned. Next we prove that the $(i - 1)$ -th item is also $\langle op, r \rangle$ -owned.

If p increases the counter in line 99, then it reads the $(i - 1)$ -th item of op and the item's owner (line 91) and verifies that the $(i - 1)$ -th item is $\langle op, r \rangle$ -owned (line 96). If p increases the counter in line 79, it reads the $(i - 1)$ -th item of op (line 75) and the item's owner (line 77) and verifies that the $(i - 1)$ -th item is $\langle op, r \rangle$ -owned (line 78). In both cases, since the ABA value in the context of op has not changed after p reads it,

no operation op' other than op $\langle op', r' \rangle$ -owned op . By property 1, no executing process $\langle op, r \rangle$ -releases the $(i - 1)$ -th item.

Hence, when op increases the counter and shifts to a $\langle i, \langle op, r, a \rangle, r \rangle$ -context, the j -th item of op is $\langle op, r \rangle$ -owned for every j , $0 \leq j < i$. Since $i < k$, property 1 implies that while op is in the $\langle i, \langle op, r, a \rangle, r \rangle$ -context and is $\langle op, r \rangle$ -owned, no executing process $\langle op, r \rangle$ -releases any item.

It is left to handle the scenario in which an executing process $\langle op, r \rangle$ -releases op and another (consider the first) executing process p' $\langle op, r' \rangle$ -acquires op afterwards. Assume p' shifts op to a $\langle i', \langle op, r', a' \rangle, r' \rangle$ -context. If while op is not $\langle op, r \rangle$ -owned some executing process $\langle op, r \rangle$ -releases an item then, Property 1 implies that op was $\langle op'', r'' \rangle$ -owned ($op'' \neq op$), and some executing process $\langle op'', r'' \rangle$ -releases op (in line 88), before p' $\langle op, r' \rangle$ -acquires op . Thus, $i' = 0$, and the claim vacuously holds. Otherwise, by Property 1, while op is not $\langle op, r \rangle$ -owned, op was not reset, and no executing process $\langle op, r \rangle$ -releases any item. Hence, $r' = r$, $i' = i$, and the property holds.

Property 3: Assume, towards a contradiction, that the owner or round number of an operation nop change after an executing process p verifies that nop is in a $\langle ctr, \langle nop, nr, la \rangle, nr \rangle$ -context (line 117) and before p $\langle nop, nr \rangle$ -acquires an item t in the acquire method (line 118).

The first case is when an executing process, p' , $\langle nop, nr \rangle$ -releases nop for the first time in line 88. If p' applies line 88 while executing the reset method of another operation op' . Before the reset (line 54 or 62), p' verified that nop is $\langle op', r' \rangle$ -owned (line 53 or 61). Thus, this is not the first time an executing process $\langle nop, nr \rangle$ -releases nop after p read the context of nop , a contradiction. Otherwise, p' applied line 88 in the execute method of nop after reading $c(nop) = k$ while p reads $c(nop) = ctr < k$. By Property 2, when the counter is set to $ctr + 1$, t is $\langle nop, nr \rangle$ -owned, and the CAS applied by p so as to acquire t fails since at least the ABA value changes, a contradiction.

The second case is when an executing process p' changes the context by applying line 58. If p invokes the acquire method in line 94, then p reads that no owner is blocking nop while p' reads that some $\langle cop, cr, ca \rangle$ owner is blocking nop . Consider the executing process p'' that set *conflict* of nop to this value. Since p acquires t , no other operation acquires t after p reads it and before p $\langle nop, nr \rangle$ -acquires t . Hence, the interleaving of the steps is as follows: p'' reads the *conflict* of nop , including its ABA value. Then, p'' reads the owner on t (line 91) and discovers that t is $\langle cop, cr \rangle$ -owned. Afterwards, p reads that no operation owns t (line 92) and invalidates *conflict* of nop (by “touching” the ABA value). Hence, p'' cannot set *conflict* (line 102), since at least the ABA value changes when p invalidates *conflict*, a contradiction.

Otherwise, p invokes the acquire method in line 76. In this case both p and p' read that nop is blocked. p reads that the owner $\langle cop, cr, a \rangle$ is blocking nop on its i -th item in the nr -th round and acquires the operation object of cop in configuration C so as to reset cop . Assume p' reads the same value in *conflict* of nop . That is, p' reads that

the owner $\langle cop, cr, a \rangle$ is blocking nop on its i -th item in the nr -th round. Claim 3.10 implies that p' cannot apply line 58.

Claim 3.10 *During the execution interval α that starts in C and ends when the reset is completed, p' does not $\langle op, r \rangle$ -release op in line 58.*

Proof: If p $\langle op, r \rangle$ -acquires cop and op atomically in line 61 then either p' discovers that both cop and op are $\langle op, r \rangle$ -owned or it fails to $\langle op, r \rangle$ -release op since at least the ABA value has changed.

Otherwise, p $\langle op, r \rangle$ -acquires cop in line 53. p reads the context of cop (line 47) and discovers that $i > c(cop)$ (line 52). By Lemma 3.8, if the counter of cop increases then cop is $\langle cop, cr' \rangle$ -owned for some cr' , after cop was $\langle op, r \rangle$ -released in line 88 at the end of α . So, during α , $i > c(cop)$ and p' does not get to apply line 58 successfully. ■

Assume p' reads a different value in *conflict* of nop . That is, p' reads that an operation defined by $\langle cop', cr', a' \rangle$ is blocking nop on its i' -th item in the nr' -th round. If $\langle cop', cr', a' \rangle$ was blocking nop prior to $\langle cop, cr, a \rangle$, then some executing process invalidates the context of nop (line 49) before resetting *conflict* of nop (line 50) and setting it to $\langle cop, cr, a \rangle$. Hence, p' cannot change the context of nop , a contradiction. Otherwise, $\langle cop, cr, a \rangle$ was blocking nop prior to $\langle cop', cr', a' \rangle$.

Claim 3.11 *During the execution interval α that starts in C and ends when the i -th item of op is $\langle op, r \rangle$ -owned, the round numbers of op and cop do not change.*

Proof: Consider the first event e changing the round number of cop by an executing process p'' in line 88. If p'' increases the round number in the *execute* method of cop then the counter of cop increased to the size of cop 's data set. By Lemma 3.8, when the counter increases, cop is $\langle cop, cr' \rangle$ -owned for some cr' . Otherwise, p'' increases the round number of cop in the *reset* method of another operation op' , after verifying that cop is $\langle op', r' \rangle$ -owned. Both cases imply that cop was $\langle op, r \rangle$ -released first in line 88, which implies an increase in the round number of cop , contradicting the assumption that e is the first event changing the round number.

Consider the first event e' changing the round number of op applied by an executing process p'' in line 88. If p'' increases the round number of op after executing the *modify* method then the counter of op increased to k . By property 2, when the counter is set to $i + 1$ the i -th item is already $\langle op, r \rangle$ -owned, and the claim holds. Otherwise, p'' increases the round number while executing the *reset* method of another operation op' after verifying that op is $\langle op', r' \rangle$ -owned. If some executing process $\langle op, r \rangle$ -releases op in line 88 prior to e' , then this implies an increase in the round number of op , contradicting the assumption that e' is the first event changing the round number. Otherwise, some executing process $\langle op, r \rangle$ -releases op in line 58 during α , contradicting Claim 3.10. ■

By Claim 3.11, neither *nop* nor *cop* change their round numbers. Hence, no executing process can set *conflict* of *nop* (line 50) to $\langle cop', cr', a' \rangle$ before $p \langle nop, nr \rangle$ -acquires t . ■

The value of a data item is changed by *op* only when *op* owns all its data items. Thus, operations apply their changes in isolation, and can be considered to take effect atomically at the linearization point.

Theorem 3.12 (Linearizability) *LocalRMW is linearizable.*

Proof: Lemma 3.8 implies that when an operation *op* shifts to the $\langle k, l, r \rangle$ -context, it is $\langle op, r \rangle$ -owned, i.e., $l = \langle op, r, a \rangle$, and Lemma 3.9(2) implies that *op* owns all its data items.

An execution process p' of *op* reads the $\langle ic, il, ir \rangle$ -context of *op* either before or after the shift, i.e., either $ic < k$ or $ic = k$.

If $ic < k$, any attempt to change the context by p' after the shift and before the *modifyDone* flag of *op* is set to TRUE fails: p' fails to apply the CAS in lines 99, 49, 58 using *ic* since the counter has changed. *ic* is also used in line 79 while executing the reset method (lines 54, 62). Changing the context in lines 67 and 71, while trying to acquire the operation, (lines 53, 61), fails since the operation is $\langle op, r \rangle$ -owned and the owner is not equal to \perp . Lemma 3.9(1) implies that p' does not $\langle op, r \rangle$ -release any item of *op*.

If $ic = k$, then p' invokes the *modify* method and sets the *modifyDone* flag to TRUE. Only then p' $\langle op, r \rangle$ -releases the items of *op* (line 87 in the *releaseDataset* method and changes the context of *op* (line 88).

Since *modify* was invoked after *op* acquired all its data items, the implementation ensures that the data items are changed only if the *modifyDone* flag of *op* is not TRUE, hence, guaranteeing that while applying the changes all the items in the data set of *op* are $\langle op, r \rangle$ -owned. ■

3.2.4 Progress and Locality Proofs

The locality and progress proofs of LocalRMW are similar to those of BLocalRMW. Lemmas 3.1-3.6 can be adapted to hold also for LocalRMW. Below, we prove that if a process p takes many steps in executing *op*, it will eventually get to help any process that might be blocking it. In this way, helping “simulates” a scenario in which the initiators of nearby operations do not stop taking steps, alleviating the effects of blocking. Specifically, the next lemma proves that after a finite number of steps by p , there is progress in the $2k$ -neighborhood of *op*, and the following lemma uses it to show that eventually, some operation in the $3k$ -neighborhood of *op* completes.

The proof relies on a parameterized notion of an increasing-decreasing chain: in an $\langle l, m \rangle$ -*increasing-decreasing* chain, the length of the increasing chain is l and the counter of the first operation in the decreasing chain is m .

Lemma 3.13 *Given an operation op in a configuration C , consider an $\langle l, m \rangle$ -increasing-decreasing chain \mathcal{IDC} from op . There is an execution interval α from C that includes $O(l + m + k)$ steps of an executing process of op , such that either α is a big step of the \mathcal{IDC} or α contains a progress step of operation in the $2k$ -neighborhood of op .*

Proof: We prove the lemma by double induction on l and m .

First consider the case that $l = 1$ and $m \geq 0$. The base case is a $\langle 1, 0 \rangle$ -increasing-decreasing chain, which includes only op , and the lemma clearly holds in any execution with one step of an executing process of op .

Next assume the claim holds for a $\langle 1, m' \rangle$ -increasing-decreasing chain with any m' , $0 \leq m' < m$. Consider an operation op with $c(op) \geq m$ that is m -delayed by another operation op' that is the first operation in the decreasing chain, with $c(op') = m$. When op discovers that it is blocked by op' with lower or equal counter (line 52 or line 60), op tries to acquire the operation object of op' (line 53 or line 61). Then op verifies that it owns the operation object of op' (line 68 or line 72). If it succeeds to acquire the operation object of op' (line 107), op resets op' (line 54 or line 62); op increases its counter and completes the reset within $O(k)$ steps, and the claim is satisfied.

Otherwise, op fails to acquire both the operation object of op and op' . Assume op discovers that some operation op'' acquired one of the operation objects (line 109). If op'' is resetting op or op' , op helps op'' to complete the reset and increase its counter within $O(k)$ steps, and the claim holds since op'' is in the 2-neighborhood of op .

If op discovers that op' holds the operation object of op' then op helps op' . We note that if no operation in the $2k$ -neighborhood has a progress step, op' has a $\langle 1, m' \rangle$ -increasing-decreasing chain, where $m' < m$. By the inductive assumption, after $O(l + m' + k)$, which is $O(l + m + k)$ steps of the executing process one of the properties holds.

If the round number of op increases, then op was reset increasing the counter of the resetting operation and the claim is satisfied. If the counter of op' increases the claim is also satisfied. Otherwise, op fails to acquire the operation objects of op and op' , yet no operation holds them when the executing process checks the owners (line 105). If this occurs more than once, while none of the above scenarios apply, then it implies that op' releases and re-acquires its operation object. Hence, either some operation op'' blocking op' completed or was reset (increasing the counter of the resetting operation) and is no longer blocking op' , or op' was reset (increasing the counter of the resetting operation) and is no longer blocked by op'' . Hence some operation in the 3-neighborhood of op has a progress step, satisfying the claim.

To prove the claim for any $l > 1$, assume it holds for any $\langle l - 1, m \rangle$ -increasing-decreasing chain with $m \geq 0$. Consider an operation op that is blocked by an operation op' with $c(op') > c(op)$, which is blocked by an $\langle l - 1, m \rangle$ -increasing-decreasing chain. When op discovers that op is blocked by op' with higher counter (line 63), it helps op'

(line 64) and becomes an executing process of op' . By the inductive assumption, after $O(l - 1 + m + k)$ steps of the executing process one of the properties holds. ■

Recall that n_d is the number of operations in the d -neighborhood of op .

Lemma 3.14 *After $n_{2k}n_{k-1}k^2$ steps of an executing process of operation op , some operation in the $3k$ -neighborhood of op completes.*

Proof: Consider an operation op in a configuration C , with an $\langle l, m \rangle$ -increasing-decreasing chain from op , in C . Note that op by itself is a $\langle 1, m \rangle$ -increasing-decreasing chain, where m is the counter of op . Lemma 3.13 implies that there is an execution interval α_1 starting at C , in which either an executing process of op takes $O(k)$ steps and some operation in the $2k$ -neighborhood of op has a progress step in α_1 , or α_1 contains $O(1)$ big steps of the increasing-decreasing chain.

Consider such $n_{2k}n_{k-1}k^2$ consecutive execution intervals $\alpha_1, \alpha_2, \dots$. Lemma 3.13 and Lemma 3.6 (adapted for LocalRMW), imply $n_{2k}n_{k-1}k^2$ progress steps in the $2k$ -neighborhood of op . Since the number of operations in the $2k$ -neighborhood of op is at most n_{2k} , some operation in this neighborhood has $n_{k-1}k^2$ progress steps and by Lemma 3.4 (adapted for LocalRMW), some operation in the $3k$ -neighborhood of op completes. ■

Recall that an implementation has *d -local step complexity* if the step complexity of an operation is bounded by a function of the number of operations in its d -neighborhood. This notion can be considered as the quantitative analogue of the local nonblocking definition, and indeed Lemma 3.14 implies that LocalRMW has $3k$ -local step complexity. Moreover, since the number of processes is finite, it follows that an operation has an infinite number of steps without completing only if infinitely many operations complete in its $3k$ -neighborhood.

Corollary 3.15 (Local nonblocking) *LocalRMW is $3k$ -local nonblocking.*

Recall that an implementation has *d -local contention* if two operations accessing the same memory location simultaneously are at distance $\leq d$.

Lemma 3.16 *LocalRMW has $(4k - 6)$ -local contention.*

Proof: We show that the initiator of an operation op_1, p_1 , accesses the same memory location as the initiator of an operation op_2, p_2 , only if op_2 is in the $(4k - 6)$ -neighborhood of op_1 .

An initiator accesses an item t or an operation object o only while helping an operation op_1 such that o is the operation object of op_1 or t is in the data set of op_1 . It can be proved by induction that an operation op_1 only helps another operation op_2

if op_2 is on an increasing-decreasing chain from op_1 , or if op_2 holds an operation on an increasing-decreasing chain from op_1 .

Thus, p_1 contends with p_2 only if they are helping an operation op_3 (which may be either op_1 or op_2). The operation op_3 is on an increasing-decreasing chain from both op_1 and op_2 , or owns an operation object on such a chain, thus, it is within the $(2k - 3)$ -neighborhood of both op_1 and op_2 , and the distance between op_1 and op_2 is at most $4k - 6$. ■

3.3 Example: *kCAS*

The *kCAS* operation extends and refines the implementation of the *kRMW* operation. In addition to the attributes of an *Operation* instance, each *kCAS* operation instance contains two additional arrays of size k : *expValues* and *newValues*. The *result* flag indicates whether the operation succeeded to update the new values of the items, like in the unary *CAS* primitive. Being a refinement of the *kRMW* operation, when invoking a *kCAS* operation, the process executes the *rmw* method, during which the *result* flag is set, and completes the operation by returning the value written in *result*.

Pseudocode 3.6 presents the *modify* method for a *kCAS* operation. A process executing the *modify* method first verifies that all the values of the items in *op*'s data set are equal to their expected values (lines 123-127), then the *result* flag is set by the result of the consistency test (line 128). If the consistency test is successful (line 129) and the modifications of the operation have not been applied yet (line 134), then the values of the items in *op*'s data set are set to their new values, respectively (lines 130-136). ABA-prevention counters are associated with the data the operation is modifying, to ensure that each operation modifies its data set only once while owning all items in the data set, and before the *modifyDone* flag of the operation is set to `TRUE`.

We now assume that the *modify* method of a *kCAS* operation *op* is invoked only after all the data items of *op* are $\langle op, r \rangle$ -owned. Under this assumption, we prove that the data items of *op* are modified only if the *modifyDone* flag of *op* is not set to `TRUE` yet. This satisfies the precondition required by Section 3.2.3, and therefore satisfies the specification of a *kCAS* operation.

Lemma 3.17 *Consider a *kCAS* operation op , and assume the *modify* method of op is invoked after op is in a $\langle k, l, r \rangle$ -context. The value of the i -th data item of op changes (line 136) at most once, prior to setting *modifyDone* to `TRUE`.*

Proof: The pseudocode implies that *op* is successful (*result* is equal to `TRUE`), if and only if the consistency test (lines 123-127) applied by some executing process was successful. Furthermore, since *result* is changed only once (from `NULL` to `TRUE` or `FALSE`), if one process reads the operation is successful (line 129) then all the other executing processes also read the operation is successful.

Pseudocode 3.6 The modify method for k CAS

```

class kCAS(extends Operation) {
  Data expValues[k]
  Data newValues[k]
  Result result // initialized to NULL, set to TRUE or FALSE
}

120: kCAS::modify() {
121:   Data oldValues[k]
122:   res ← TRUE
123:   for i = 0 to k - 1 do
124:     item ← READ(dataset[i])
125:     oldValues[i] ← READ(item.data)
126:     exp ← READ(expValues[i])
127:     res ← res and (oldValues[i].val = exp)
128:   CAS(result, NULL, res)
129:   if READ(result) = FALSE then return
130:   for i = 0 to k - 1 do
131:     item ← READ(dataset[i])
132:     exp ← READ(expValues[i])
133:     new ← READ(newValues[i])
134:     if READ(modifyDone) = TRUE then return
135:     if exp != new then
        // write modified values
136:       CAS(item.data, ⟨exp, oldValues[i].aba⟩, ⟨new, oldValues[i].aba+1⟩)
137: }
```

Prior to changing the value of the i -th data item of op (line 136), an executing process, p reads the data of the item, including the ABA value (line 125), it verifies that the operation is successful (line 129), and it verifies that the operation has not yet set *modifyDone* to TRUE (line 134).

If another process changes the i -th item after p reads it, then p fails to change it again since at least the ABA value has changed.

The fact that op is in a $\langle k, l, r \rangle$ -context implies that prior to checking the *modifyDone* flag, no other operation has changed the item. All executing processes of op read that op is successful and try to apply its changes, including the process setting *modifyDone* to TRUE. If another executing process of op changes the i -th item before p reads it, then p fails to change it again since the value of the data was changed by the executing process (line 135). ■

Chapter 4

Doubly-Linked Lists

In static abstract data types, such as the k RMW operation discussed in the previous chapter, the data set of an operation does not change during its execution. However, most data structures—and hence the data sets of their operations—are dynamic. For example, the data set of an operation removing node t in a linked list includes the neighbors of t . However, since the linked list is dynamically changing, another operation can remove one of the neighbors or insert a new neighbor next to t , thus changing the data set during the execution of the operation.

The implementation of the data structure, as well as its correctness proof, must accommodate these dynamic changes to ensure the operation are applied correctly. Furthermore, the data set needs to be defined dynamically, with respect to the changes in the state of the data structure, and the dynamic changes must be reflected also in the conflict graph and distance related measures.

Recall that the definition of a conflict graph in Chapter 2 relies on static data set. This chapter addresses the dynamic case; we first give formal definitions of the state of a dynamic data structure and then define the data set of its operations.

Consider an instance T of a data structure. The *set of states* of T after an execution α contains every state of T that is reachable after a linearization of α .

Definition 5 *If \mathcal{ST} is the set of states of an instance of a data structure at a configuration C , then the data set of an operation at C is the union, over every state s in \mathcal{ST} , of the set of items accessed by the operation when executed from s . The data set of an operation is the union of the data sets in all the configurations during its execution.*

In the *conflict graph of a configuration C* the vertices represent the operations whose execution intervals contains C and an edge connects two operations whose data sets intersect. The vertices of a conflict graph of a dynamic data structure may change during the execution. Therefore, the *conflict graph* of an execution interval α is the

⁰The results of this chapter have appeared in [11].

union of the conflict graphs of all configurations C in α ; that is, the vertices (edges, respectively) in the graph are the union of the vertices (edges, respectively) of all these conflict graphs.

This implies the dynamic version of the distance related measures such as local nonblocking, local contention, and local step complexity.

This chapter considers a specific dynamic data structure, *doubly-linked list*; the items are the *nodes* composing the linked list. Each node has links to its left and right neighboring nodes. Two special *anchor* nodes serve as the leftmost and rightmost nodes in the list, denoted LA and RA , respectively; they cannot be removed from it, and hold no left link or no right link, respectively. A node is *valid* if it is either an anchor, or both its left link and right link pointers are not null.

We concentrate on the following operations: *PushLeft*, *PopLeft* operations (and their equivalents *PushRight*, *PopRight*), which inserts a node or removes the node next to the left (right) anchor respectively; *InsertRight* operation (and its equivalent *InsertLeft*) and *Remove* operation applied to some valid node called *source* in the linked list, which inserts a node to the left of, or removes, the source node. The sequential specification of these operations appear in Figure 4.1. Figure 4.2 shows an example of several overlapping operations and their data sets in a specific configuration: op_1 is a *PushLeft* operation, op_2 is a *PopLeft* operation, op_5 is a *PushRight* operation, op_3 inserts a new node to the right of m_2 , and op_4 removes m_4 .

This chapter is organized as follows: **CAS-Chromo**, a CAS-based implementation of a priority queue allowing insertions anywhere and removals only from the ends appears in Section 4.1. Section 4.2 outlines the modifications needed to obtain **DCAS-Chromo**, a DCAS-based implementation of a linked list supporting removals anywhere. The correctness proof of both algorithms is given in Section 4.3, while their locality properties are proved in Section 4.4.

4.1 CAS-Chromo: Priority Queue and Deque

CAS-Chromo is a doubly-linked list implementation that allows insertions anywhere (*PushLeft*, *PushRight*, *InsertRight*, and *InsertLeft*) and removals only at the ends (*PopLeft*, and *PopRight*); see Figure 4.1. Our description focuses on the *PushLeft* and *PopLeft* operations.

We presents an approach for improving the locality of concurrent linked-list implementations, by exploiting the fact that operations on the linked-list always access two or three consecutive nodes in the list. The key novelty of our approach is in having nodes that are legally colored¹ with three ordered colors, $c_1 < c_2 < c_3$. (In Figure 4.2, for example, the nodes are legally colored with three colors, yellow < blue < red.) The

¹Adjacent nodes in the list have different colors.

```

PushLeft(nd) {
  nd.right ← LeftAnchor.right
  nd.left ← LeftAnchor
  LeftAnchor.right.left ← nd
  LeftAnchor.right ← nd
  return SUCCESS
}

```

(a) PushLeft(*nd*): Insert *nd* to the right of the left anchor and return SUCCESS.

```

InsertRight(nd) {
  if source is not valid or
  source is right anchor then
    return INVALID
  nd.right ← source.right
  nd.left ← source
  source.right.left ← nd
  source.right ← nd
  return SUCCESS
}

```

(c) InsertRight(*nd*): If *source* is a valid node other than the right anchor, then insert *nd* to the right of *source* and return SUCCESS; otherwise, return INVALID and the linked list is unchanged.

```

PopLeft() {
  if empty list then return EMPTY
  subject ← LeftAnchor.right
  LeftAnchor.right ← subject.right
  LeftAnchor.right.left ← LeftAnchor
  subject.right ← ⊥
  subject.left ← ⊥
  return SUCCESS
}

```

(b) PopLeft(): If the linked list is not empty, then remove the *subject* next to the left anchor and return SUCCESS; otherwise, return EMPTY and the linked list is unchanged.

```

Remove() {
  if source is not valid or
  source is anchor then
    return INVALID
  source.left.right ← source.right
  source.right.left ← source.left
  source.right ← ⊥
  source.left ← ⊥
  return SUCCESS
}

```

(d) Remove(): If *source* is a valid node other than an anchor, then remove *source* from the linked list and return SUCCESS; otherwise, return INVALID and the linked list is unchanged.

Figure 4.1: Sequential specification of the linked list operations; it follows the description of the operations in [6].

operations preserve the legality of the nodes' coloring, and there is no need to color the accessed nodes from scratch, each time an operation is invoked.

As in the previous chapter, CAS-Chromo follows the common scheme [20, 91, 98] of having an operation first acquire its data set (ACQUIRE phase), and then apply its

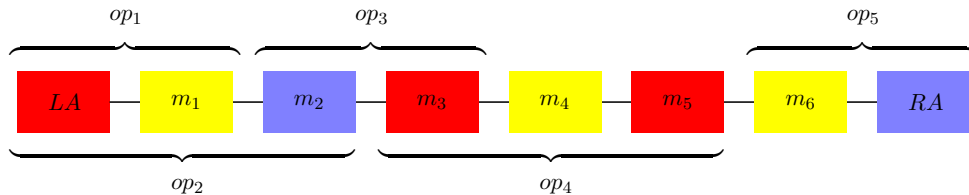


Figure 4.2: Overlapping operations of a doubly-linked list, *LA* and *RA* are left and right anchors respectively.

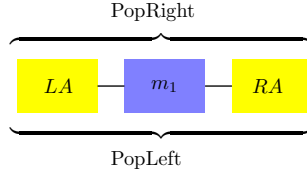


Figure 4.3: An example for two operations competing on a single node, where the anchors have the same color.

changes atomically on these nodes (APPLY phase); finally, the operation releases the items (RELEASE phase). During the ACQUIRE phase, a *hold-and-wait* scenario happens when an operation holds one or more nodes and waits for another operation to release another node. Since an operation acquires its data set in an increasing order of colors, and the colors of neighboring nodes are different, hold-and-wait chains have (strictly) *increasing* colors; the chains are short, since the number of colors is small.

Push and insert operations access exactly two adjacent nodes in the list, which must have different colors. Pop operations access three consecutive nodes, two of which may have the same color. To avoid hold-and-wait cycles, pop operations acquire monochromatic nodes according to their order in the list, from left to right. For example, consider a doubly-linked list containing a single node as depicted in Figure 4.3, where the two anchors have the same color. When the *PopLeft* and *PopRight* operations compete on removing the single node in the list, both of them try to acquire the left anchor first, and at least one of them (that succeeds in acquiring the left anchor) succeeds in popping the last item. Moreover, pop operations occur only at the ends of the list, and hence, each of them adds at most one edge to a hold-and-wait chain (one edge at each end of the chain) in addition to at most three edges connecting operations that have conflicts on nodes with increasing colors. This is used to show that the length of a hold-and-wait chain is at most 5 (see Theorem 4.11).

Maintaining a Legal Coloring: Since operations apply their changes in exclusion, they can ensure that the coloring is legal at all times, by careful color changes during the APPLY phase.

The operations use a temporary color $c_0 < c_1$, which is white in the figures, to simplify the task of maintaining the coloring legal: In the *PushLeft* operation, the color of the new node is c_0 . After the new node is in the list, and while the data set is not released yet, the new node is assigned with a color different than its neighbors. In the *PopLeft* operation, after the data set is acquired, the color of the right neighbor of the node to be removed is changed to c_0 . After the node is removed, the color of the right neighbor is changed to be different from its neighbors’.

Helping: The simple algorithm described so far may block if a process stops taking steps while holding a node. An operation op_1 is *blocked* if one of its nodes is already

owned by another *blocking* operation op_2 . As in the previous chapter, helping is used to guarantee that some operation makes progress at any time while preserving the good locality properties of the implementation. Recall that helping is recursive: in this way, *hold-and-help* chains replace hold-and-wait chains, and it is possible to prove that their length is bounded by the number of colors (see Lemma 4.6). For example, assume that operation op_3 in Figure 4.2 owns m_2 and then tries to acquire m_3 , with color red. If op_4 already owns m_3 , then op_3 helps op_4 . However, red is the largest color, implying that op_4 already owns all the nodes in its data set. Thus, when helping op_4 , op_3 only needs to apply the changes of op_4 , and not recursively help additional operations.

Dynamic Aspects: Another aspect of the algorithm is in handling the dynamic nature of the data structure, that is, after an operation reads the data set and verifies that the nodes are valid, another operation may modify the nodes and even the list structure, changing the data set of op . Our algorithm addresses this problem in a manner similar to [50]. A *data set memento*, holding a view of the data set when the operation is started, traces inconsistencies in the data set due to changes applied by concurrent operations. If, while acquiring the data set, an operation discovers that a node in its data set memento is invalid or inconsistent with its memento, it restarts. That is, the operation skips the APPLY phase and goes directly to the RELEASE phase where it releases all the nodes it already acquired and re-invokes the operation. Otherwise, the operation completes its ACQUIRE phase, and the nodes it has acquired are consistent with the data set memento, so the operation can continue with the APPLY phase as in a static scheme.

As in the k RMW operation in Chapter 3, a linked list operation may have several *rounds*, caused by inconsistencies between the items and its data set memento. The scheme is re-applied at each round until the operation *completes*. Figure 4.4 shows the state transition diagram of one round of an operation.

4.1.1 Data Structures

Figure 4.5 lists the main types and data structures used in the algorithm.

Operations are objects, whose structure and behavior are defined in the *Operation* class. An operation object is initialized with all the data required for its execution, specifically the source node on which the operation is applied, and the subject node to be inserted to the list (in insert or push operations).

Nodes are also objects. In addition to its *data* attribute, a node contains two pointers *left* and *right*, a *color* and an *owner*. A node memento is composed of a reference to the node itself, *node*, and a copy of the node's meta data except for the owner, so the node is consistent with its memento even if it is acquired and released arbitrarily, as long as the other attributed do not change.

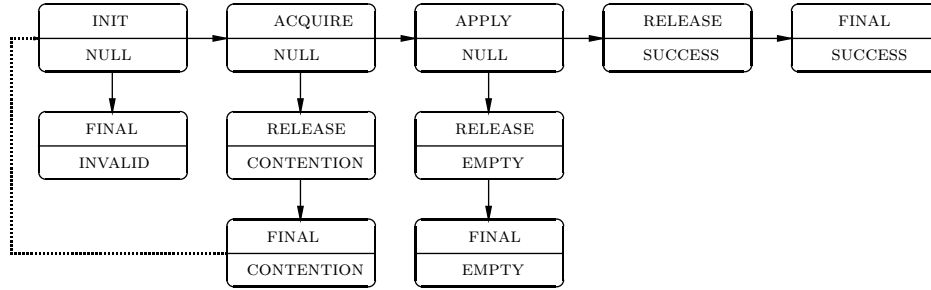


Figure 4.4: The state transitions of an operation: the lower part of the state is the *result* of the current round; the dashed line indicates a new round. The best-case scenario, encountering no contention, appear at the top. If the list is empty during a pop operation, then the operation completes without changing the linked list. If the source node is removed during the INIT phase of an operation, then the operation need not apply its changes. If there is contention during the ACQUIRE phase, the operation releases its nodes and it is re-invoked.

Due to helping, several processes may execute the same operation op (by calling the `execute` method of op simultaneously); these are the *executing processes* of op . One of the executing processes—the one that first called the `execute` method of op —is the *initiator* of op .

Since an operation may have several rounds, its execution goes through alternating phases of acquiring and releasing nodes. The *state* of an operation is a tuple $\langle seq, phase, result \rangle$: seq is the *round number*, an integer, initially 0, that is incremented when the initiator re-invokes the operation; $phase$ indicates the scheme phase within the round, set to INIT at the beginning of a round; $result$ holds the result of the current round execution, set to NULL at the beginning of a round.

To avoid the ABA problem we apply the same solution as in LocalRMW, and associate each attributes that may hold the same value during an execution with a monotonically increasing ABA-prevention counter.

4.1.2 Implementation

Pseudocodes 4.1, 4.2, 4.3 and 4.4 present the code for CAS-Chromo.

The initiator starts the operation with the `execute` method and as long as it suffers from contention and is unable to complete (line 15), it repeatedly tries to clear the attributes (line 3) and re-invoke the operation. It generates the new data set memento (line 11), and then “helps” itself to follow the scheme (line 13): acquire nodes in its data set (line 21), apply its changes (line 24), and release the data set (line 26).

In the `acquireColor` method, an operation op repeatedly tries to acquire all nodes with a given color c in its current data set memento; after verifying the nodes are valid and consistent with their mementos (line 38), op tries to acquire the nodes (line 41).

```

define LEFT = 0, MIDDLE = 1, RIGHT = 2
define Phase = {INIT, ACQUIRE, APPLY, RELEASE, FINAL}
define Result = {NULL, SUCCESS, CONTENTION, INVALID, EMPTY}
structure State {int seq, Phase phase, Result result}
structure NodePtr {Node node, int aba}
structure Color {Color color, int aba}
structure Owner {Operation op, int seq, int aba}

structure Node {
    Data    data,
    NodePtr left,
    NodePtr right,
    Color   color,
    Owner   owner
}
structure NodeMemento {
    Node    node,
    NodePtr left,
    NodePtr right,
    Color   color
}

class Operation {
    State    state // initially (0,INIT,NULL)
    Node     source // in push/pop operations the source is an anchor
    Node     subject // either the new node or the removed node
    NodeMemento[3] datasetMemento
    Color[3] colorSet
}

```

Figure 4.5: Data structures for CAS-Chromo and DCAS-Chromo.

If op discovers that none of these nodes is owned by another operation, when failing to acquire them, it simply retries. Otherwise, op finds that a node in its data set is owned by another, blocking operation op' (line 46), and helps op' to complete (line 47).

During the execution of an operation, a process applies validity checks to ensure the consistency of the execution, as well as other properties of the implementation, such as locality. Next we describe these tests, give the motivation for applying them, and explain what steps are taken in case of failure, i.e., when the result of the test is negative.

Before helping another operation op' during the `acquireColor` method, the executing process of op verifies (again) that the nodes are consistent with their mementos (line 44). This is crucial for maintaining the locality properties of the algorithm. Lemma 4.6 proves that the length of helping chains is bounded by the number of colors, by showing that a process helps along a chain with monotonically increasing colors. The consistency check ensures that the color of the node did not decrease; without this check, a process may help along an unbounded chain with colors alternately increasing and decreasing.

The state attribute is used to synchronize the executing processes of an operation. It is possible that a delayed (slow) process, executing a previous round, tries to acquire nodes that are associated with this previous round or releases nodes that were re-acquired in the current round. Therefore, an executing process verifies, before acquiring

Pseudocode 4.1 CAS-Chromo: code for main methods

```
1: Result execute() {
2:   do
3:     clear memento and color set
4:     toInitState()
5:     if source is invalid then
6:       owner  $\leftarrow$  source.owner
7:       owner.op.help(owner.seq)
8:       toFinalInvalidState()
9:     if state.result = INVALID then
10:      return
11:     else // new round
12:       cloneDataset()
13:       toAcqState()
14:       help(state.seq) // help myself
15:       toFinalState()
16:     while state.result = CONTENTION
17:   return state.result
18: }

19: help(int seq) {
20:   if state  $\neq$   $\langle$ seq,ACQUIRE,NULL $\rangle$  then
21:     return
22:   for each color  $c$  by increasing order do
23:     acquireColor( $c$ ,seq)
24:   toApplyState(seq)
25:   if state.phase = APPLY then
26:     applyChanges()
27:   toReleaseState()
28:   releaseDataset(seq)
29: }

30: releaseDataset(int seq) {
31:   for each Node  $nd$  in  $seq$ -th memento do
32:     owner  $\leftarrow$  nd.owner
33:     if owner =  $\langle$ self,seq,t $\rangle$  and state.phase = RELEASE then
34:       CAS(nd.owner, owner,  $\langle$  $\perp$ , $\perp$ ,t+1 $\rangle$ )
35: }

36: acquireColor(Color  $c$ , int seq) {
37:    $\{nd_i\} \leftarrow$   $c$ -colored nodes in  $seq$ -th memento
38:   while true do
39:      $\{owner_i\} \leftarrow$  get all owners from  $\{nd_i\}$ 
40:     checkNodes( $\{nd_i\}$ ,seq)
41:     if state  $\neq$   $\langle$ seq,ACQUIRE,NULL $\rangle$  then return
42:     for each  $nd_i$  from left to right do // owner all equally colored nodes
43:       if  $owner_i = \langle \perp, \perp, t \rangle$  then CAS( $nd_i$ .owner,  $owner_i$ ,  $\langle$ self,seq,t+1 $\rangle$ )
44:       // check if acquired all  $c$ -colored nodes or blocked
45:     owner  $\leftarrow$  get owner from  $\{nd_i\}$ 
46:     if owner.op = self then return // succeeded
47:     checkNodes( $\{nd_i\}$ ,seq)
48:     if state  $\neq$   $\langle$ seq,ACQUIRE,NULL $\rangle$  then return
49:     if owner.op  $\neq$   $\perp$  then
50:       owner.op.help(owner.seq) // blocked by owner.op - help it
51: }

52: checkNodes(Set(Node)  $\{nd_i\}$ , int seq) {
53:   if invalid node or inconsistent memento then
54:     for each  $i$  do // "touch" the ABA counters of the owner
55:        $l_i \leftarrow nd_i$ .owner
56:       CAS( $nd_i$ .owner,  $l_i$ ,  $\langle$  $l_i$ .op,  $l_i$ .seq,  $l_i$ .aba+1 $\rangle$ )
57:   toReleaseContentionState(seq)
58: }
```

Pseudocode 4.2 CAS-Chromo: code for the clone methods

```

56: PushLeft::cloneDataset() {
    // subject is pushed, source is the left anchor
57:   cloneNode(source, LEFT)
58:   cloneNode(subject, MIDDLE)
59:   right ← datasetMemento[LEFT].right.node
60:   cloneNode(right, RIGHT)
61: }

62: PopLeft::cloneDataset() {
    // subject is popped, source is the left anchor
63:   cloneNode(source, LEFT)
64:   subject ← datasetMemento[LEFT].right.node
65:   cloneNode(subject, MIDDLE)
66:   righth ← datasetMemento[MIDDLE].right.node
67:   cloneNode(righth, RIGHT)
68: }

69: cloneNode(Node nd, int i) {
70:   if nd = ⊥ then return
    // Assume atomic assignment
71:   datasetMemento[i] ← ⟨nd,nd.left,nd.right,nd.data,nd.color⟩
72:   add nd's memento color to the color set
73: }

```

a node, that the round number of its execution matches the one in the state of the operation (line 39). Furthermore, if an executing process detects inconsistency between the node and the operation's memento (line 38 or line 44) then it violates the owners of these nodes by “touching” them² (line 53) before advancing the operation to the RELEASE phase. This prevents other delayed processes from acquiring the nodes when the operation is not in the ACQUIRE phase (see Lemma 4.2(5)).

To prevent a process from releasing nodes acquired in a later round, the operation adds the round number to any node it acquires (line 41); before a process releases a node, it verifies in `releaseDataset` (line 31) that the round numbers of the owner and its own parameter are equal (see Lemma 4.2(4)).

Different operations extending the *Operation* class, refine the protocols for cloning and manipulating the data set, according to their specifications. Pseudocode 4.2 shows how the data set memento (the source node and both or one of its neighbors) is created. The `applyChanges` method changes the nodes according to the specification of the operation and maintains a legal coloring. Pseudocode 4.3 describes the implementation of these methods for the `PushLeft` and `PopLeft` operations. Finally, the methods of Pseudocode 4.4, apply the state transitions of the operations, as depicted in the state transition diagram in Figure 4.4.

²This is similar to the way an operation is invalidated before releasing its nodes in [8].

Pseudocode 4.3 CAS-Chromo: code for the apply-changes methods

```

74: PushLeft::applyChanges() {
    // MIDDLE is the new node
    // LEFT is the left anchor
75:   updateRight(MIDDLE,RIGHT)
76:   updateLeft(MIDDLE,LEFT)
    // the new node is now valid
77:   updateColor(MIDDLE)
78:   updateLeft(RIGHT,MIDDLE)
79:   updateRight(LEFT,MIDDLE)
80: }

92: updateRight(int i, int j) {
93:   nmi ← i-th node memento
94:   nmj ← j-th node memento
95:   nd ← nmi.node
96:   newr ← nmj.node
97:   rt ← nmi.right
98:   CAS(nd.right, rt, ⟨newr,rt.aba+1⟩)
99: }

108: updateColor(int i) {
109:   nm ← i-th node memento
110:   nd ← nm.node
111:   lftc ← nd.left.node.color
112:   rtc ← nd.right.node.color
113:   newc ← color not in {lftc,rtc}
114:   clr ← nm.color
115:   CAS(nd.color, clr, ⟨newc,clr.aba+1⟩)
116: }

81: PopLeft::applyChanges() {
    // MIDDLE is popped
    // LEFT is the left anchor
82:   if empty list then
83:     toReleaseEmptyState()
84:   if state.result = EMPTY then return
85:   setTempColor(RIGHT)
86:   updateRight(LEFT,RIGHT)
87:   updateLeft(RIGHT,LEFT)
88:   updateRight(MIDDLE,⊥)
    // the popped node is now invalid
89:   updateLeft(MIDDLE,⊥)
90:   updateColor(RIGHT)
91: }

100: updateLeft(int i, int j) {
101:   nmi ← i-th node memento
102:   nmj ← j-th node memento
103:   nd ← nmi.node
104:   newl ← nmj.node
105:   lft ← nmi.left
106:   CAS(nd.left, lft, ⟨newl,lft.aba+1⟩)
107: }

117: setTempColor(int i) {
118:   nm ← i-th node memento
119:   nd ← nm.node
120:   clr ← nm.color
121:   CAS(nd.color, clr, ⟨c0,clr.aba+1⟩)
122: }

```

4.2 DCAS-Chromo: A Doubly-Linked List Algorithm

We now describe the extensions needed to obtain DCAS-Chromo, which allows removals from the middle of the list. This may create long helping chains, as demonstrated in Figure 4.6, which shows a long linked list of nodes with alternating colors: red, yellow, red, yellow, Consider a set of concurrent operations, each of which is trying to remove a different yellow-colored node, by acquiring the node and its two red-colored neighbors. If the two red-colored nodes are acquired one at a time, in the same order, e.g., first the left neighbor, it is possible that an operation holds its left red node, and needs to help all operations to its right.

One might suggest to extend the notion of a legal coloring and require that any

Pseudocode 4.4 CAS-Chromo: code for state transitions

```

123: toInitState() {
124:   s ← state
125:   CAS(state, s, ⟨s.seq+1,INIT,NULL⟩)
126: }

127: toAcqState() {
128:   s ← state
129:   if s.phase != INIT then return
130:   CAS(state, s, ⟨s.seq,ACQUIRE,NULL⟩)
131: }

132: toApplyState(int seq) {
133:   s ← state
134:   if s != ⟨seq,ACQUIRE,NULL⟩ then
135:     return
136:   CAS(state, s, ⟨seq,APPLY,SUCCESS⟩)
137: }

138: toReleaseState() {
139:   s ← state
140:   if s.phase != APPLY then return
141:   // linearization point - LP2
142:   CAS(state, s, ⟨s.seq,RELEASE,s.result⟩)
143: }

143: toFinalState() {
144:   s ← state
145:   if s.phase != RELEASE then return
146:   CAS(state, s, ⟨s.seq,FINAL,s.result⟩)
147: }

148: toFinalInvalidState() {
149:   s ← state
150:   if s.phase != INIT then return
151:   // linearization point - LP1
152:   CAS(state, s, ⟨s.seq,FINAL,INVALID⟩)
153: }

153: toReleaseContentionState(int seq) {
154:   s ← state
155:   if s != ⟨seq,ACQUIRE,NULL⟩ then return
156:   CAS(state, s, ⟨seq,RELEASE,CONTENTION⟩)
157: }

158: toReleaseEmptyState() {
159:   s ← state
160:   if s != ⟨seq,APPLY,SUCCESS⟩ then return
161:   CAS(state, s, ⟨seq,RELEASE,EMPTY⟩)
162: }

```

triple of neighboring nodes is assigned different colors. This certainly allows to follow the color-based scheme, but how can we preserve this extended coloring property? In particular, when a node is removed, it is necessary to acquire *four* nodes in order to legally re-color the remaining three nodes; this requires to further extend the coloring property to any four consecutive nodes, which in turn requires to acquire *five* nodes and so on; this unlimited expansion of the coloring property seems inevitable.

Our way to break out of this vicious circle is to acquire equally-colored nodes *atomically*. An operation accesses at most three consecutive nodes, which are legally colored, so at most two of them have the same color. We use DCAS to atomically acquire these nodes, e.g., the two red-colored nodes in the scenario of Figure 4.6, and to break the symmetry.

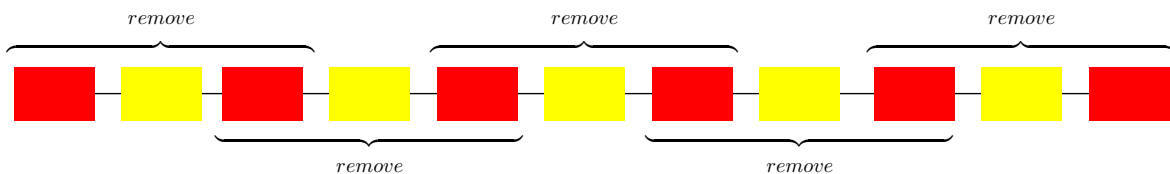


Figure 4.6: An example of a symmetric scenario; each operation removes a different yellow-colored node, and accesses the node and its two red-colored neighbors.

Pseudocode 4.5 Code changes for DCAS-Chromo

```
163: Remove::cloneDataset() {
164:   cloneNode(source, MIDDLE)
165:   left ← datasetMemento[MIDDLE].left.node
166:   cloneNode(left, LEFT)
167:   right ← datasetMemento[MIDDLE].right.node
168:   cloneNode(right, RIGHT)
169: }

170: acquireColor(Color c, int seq) {
171:   {ndi} ← get all nodes with color c from the seq-th memento
172:   while true do
173:     {owneri} ← get all owners from {ndi}
174:     checkNodes({ndi},seq)
175:     if state != ⟨seq,ACQUIRE,NULL⟩ then return
    // atomically owner two equally colored nodes
176:     if for each i, owneri = ⟨⊥,⊥,ti⟩ then
177:       DCAS(nd1.owner,           nd2.owner,
            owner1,             owner2,
            ⟨self,seq,owner1.aba+1⟩, ⟨self,seq,owner2.aba+1⟩)
178:     owner ← get owner from {ndi} // check if succeeded or blocked
179:     if owner.op = self then return // acquired all c-colored nodes
180:     checkNodes({ndi},seq)
181:     if state != ⟨seq,ACQUIRE,NULL⟩ then return
182:     if owner.op != ⊥ then // blocked by owner.op operation
183:       owner.op.help(owner.seq) // help blocking operation
184: }

185: Remove::applyChanges() {
    // the MIDDLE node in the memnto is removed
186:   setTempColor(RIGHT)
187:   updateRight(LEFT,RIGHT)
188:   updateLeft(RIGHT,LEFT)
189:   updateRight(MIDDLE,⊥) // the removed node is now invalid
190:   updateLeft(MIDDLE,⊥)
191:   updateColor(RIGHT)
192: }
```

Pseudocode 4.5 presents the code for the additional operation *Remove*, for removing a non-anchor valid node (the source node) from the list. The `applyChanges` method is very similar to the `pop` operations (removing nodes from the ends) except that it does not handle the case where the list is empty. The most important modification, relative to *CAS-Chromo*, is in the `acquireColor` method, which now uses `DCAS` when acquiring two nodes with the same color. That is, instead of acquiring the nodes one by one (Pseudocode 4.1, lines 40-41) `DCAS` acquires both nodes atomically (Pseudocode 4.5, lines 176-177).

4.3 Safety Proof

In this section, we prove that CAS-Chromo and DCAS-Chromo are linearizable (Theorem 4.3). The proof does not assume that DCAS is used, and thus it holds for both algorithms. Using DCAS is critical only for proving that the coloring is legal and showing locality properties of the algorithm, and is considered only in Section 4.4.1.

The linearizability of both algorithms hinges on showing that executing processes preserve the correct phase transitions of the operation—acquiring, changing and releasing nodes—and take steps in accordance with the operations’ phase. Most importantly, nodes in the data set are changed only while all of them are owned. As mentioned before, this is somewhat more complicated than in previous work [5, 8, 20, 91, 98], since the data set is dynamic.

It can be inferred directly from the methods of Pseudocode 4.4, which are the only way to make state transitions, that an operation follows the state transition diagram in Figure 4.4. Moreover, the round number is increased before every round. Hence, no operation makes a transition to the same state tuple more than once.

The following terminology is used in the proofs. The ACQUIRE phase of the r -th round is called the r -th ACQUIRE phase, and similarly for the other phases. The code implies that an operation is in APPLY phase at most once (since all transitions from it are to a final state), in which case the operation completes and will not be re-invoked again; this is called the *last round*. Only the initiator generates the data set memento, once per round (line 11); the data set memento written in the r -th round is called the r -th data set memento; the data set memento of the last round is called the *last data set memento*.

Several executing processes can make the transitions to the APPLY and RELEASE phases concurrently, but other transitions are only made by the initiator. A transition to the APPLY phase only occurs once, in the last round; the method implementing a state transition to the RELEASE phase in case of contention takes as argument the round number, to ensure the transition occurs only if it is executed for the correct round.

The next two lemmas state the main invariants of the algorithm.

Lemma 4.1 *An operation op successfully applies changes only when it is in APPLY phase, and only to nodes in op ’s last data set memento.*

Proof: An executing process of op that calls `applyChanges` first verifies that the operation is in the APPLY phase (line 23), implying that this is the last round of the operation, and it holds the last data set memento. Since an operation changes only nodes in its data set memento, executing processes apply the same changes on the same nodes (from the last data set memento).

Let p_j be the process that advances op from APPLY phase to RELEASE phase (line 25). Before changing the state, p_j executes `applyChanges` (line 24), thus CAS is applied at

least once to each of the attributes based on values in the node mementos (see the code in Pseudocode 4.3), prior to the state transition. Once a CAS is applied successfully the attribute is inconsistent with its memento since at least the ABA-prevention counter has changed. If after the transition another process p_i applies CAS to some attribute while applying op 's changes, p_i 's CAS fails. ■

The main properties we use are stated in items 1-4 of Lemma 4.2. In order to prove them, it is useful to carry in the induction two additional items (numbered 5 and 6).

Lemma 4.2 *The following claims all hold for every operation op :*

1. *If op acquires a node t in the r -th round, then the r -th memento of t in op 's data set is valid and t , excluding t 's owner, has not changed after it was cloned by op in the r -th round.*
2. *op advances from the r -th ACQUIRE phase to the r -th APPLY phase only if all the nodes in its data set are consistent with the r -th data set memento, and are owned by op .*
3. *op only applies changes to nodes that are owned by it.*
4. *op releases nodes only when it is in RELEASE phase, and during its r -th RELEASE phase it only releases nodes it has acquired in the r -th round.*
5. *op acquires nodes only when it is in ACQUIRE phase, and during its r -th ACQUIRE phase it only acquires nodes that are in its r -th data set memento.*
6. *When an executing process of op returns from $acquireColor(c,r)$, either all nodes with color c in the r -th data set memento are owned by op , or op is not in the r -th ACQUIRE phase.*

Proof: The claims are proved simultaneously by induction on the execution length. In the base case, the empty execution, all claims vacuously hold. In the induction step, we consider each claim:

1. If the r -th memento of a node t in op 's data set is invalid no executing process tries to acquire t (line 38, and line 53).

Now, assume t has changed after op cloned it in the r -th round, and that p_i , an executing process of op calls $acquireColor(c,r)$, where c is the color of t . If the change occurs before p_i verifies t is consistent with its memento (line 38), then p_i does not try to acquire t . Otherwise the change occurs after verifying the consistency, and in particular after p_i reads t 's owner (line 37). By Lemma 4.1, the change is applied by an operation op' in APPLY phase. By (3), t is acquired by op' when applying the change. If op' acquired t before p_i reads t 's owner, then by (4) it is not released until after op' applied the changes, i.e., until after p_i reads the owner, then p_i does not try to acquire t . Otherwise, op' acquired t after p_i reads t 's owner, in this case, p_i fails acquiring t , since the owner's ABA-prevention counter has changed.

2. A transition of op from the r -th ACQUIRE phase to the r -th APPLY phase by an executing process occurs only after the executing process calls `acquireColor(c,r)` with all colors from the r -th color set while op is in the r -th ACQUIRE phase. By (6), when returning from each such round, all relevant nodes are owned by op , and by (4), they were not released since then. As the set of colors includes all nodes in the r -th data set memento, they are all owned by op while the transition occurs. By (1), no node in the r -th data set memento is changed before op advances to the APPLY phase. So, when the transition occurs all nodes in the r -th data set memento are owned by op , and they are consistent with their mementos.
3. By Lemma 4.1, op only applies changes while it is in APPLY phase and only to nodes that are in the last data set memento. By (2), when the transition to APPLY phase occurs all nodes in the r -th (last) data set memento are owned by op , and by (4), it does not release the nodes while it is in the APPLY phase, implying that op changes a node only while owning it.
4. The transition of op from the r -th RELEASE phase to the r -th FINAL phase (line 14) by the initiator of op , p , occurs after p calls the `releaseDataset` method (line 26), to release all nodes in the r -th data set memento (line 29). All processes executing the r -th round try to release the same nodes from the r -th data set memento, since by (5), op only acquires nodes from the r -th data set memento in the r -th round. When a process p_i releases op 's data set while executing the r -th round of op , it reads the owner of a node t (line 30), and tries to release t (line 32) after verifying that op is in the RELEASE phase and t is acquired by op in its r -th round (line 31). It can be easily verified from the code that after p completes the `releaseDataset` method all the nodes that were owned by op are released. Thus, if p_i tries to release the nodes after the transition occurs, the CAS fails since the ABA-prevention counter of the owner has changed.
5. Consider an executing process p_i executing the r -th round of op . First p_i reads t from op 's r -th data set memento (line 35); then p_i reads t 's owner (line 37); verifies that t 's memento is valid (line 38); and that op is in the r -th ACQUIRE phase (line 39). Let p_j be the executing process that advances op from the r -th ACQUIRE phase either to the r -th APPLY phase or to the r -th RELEASE phase. Assume the transition occurs after p_i verifies the phase, and specifically after it reads t 's owner, but before p_i tries to acquire t (line 41). If p_j advances op to the r -th APPLY phase, by (2) all nodes in the r -th data set memento, including t , are owned by op when p_j makes the transition. Thus either p_i discovers that t is acquired by op or it fails acquiring t , since at least the ABA-prevention counter has changed.

Otherwise, p_j advances to the r -th RELEASE phase since it discovers that some

node t' in the r -th data set memento is inconsistent with its memento (line 38 or line 44). There are three cases depending on the order between the colors of the nodes:

- (i) t' and t have the same color. Before the transition occurs, p_j violates the owners of both nodes by “touching” their ABA-prevention counter (line 53). By (1), since t changed while op is in the r -th ACQUIRE phase, p_i cannot successfully acquire it in this round.
 - (ii) t' has lower color than t . Thus, p_i executes `acquireColor(c', r)`, where c' is the color of t' , before trying to acquire t . By (6), t' was acquired by op and by (1), it has not changed while op is in the r -th ACQUIRE phase, which contradicts the assumption that p_j discovers it is inconsistent with its memento.
 - (iii) t' has higher color than t . Thus, p_j executes `acquireColor(c, r)`, where c is the color of t , before discovering the change in t' . By (6), t was acquired by op . Thus either p_i discovers that t is owned by op or it fails acquiring t , since at least the ABA-prevention counter has changed.
6. An executing process of op that executes `acquireColor(c, r)` first verifies that op is in the r -th ACQUIRE phase (line 19). It returns from the method in one of three cases: Two cases are when it recognizes a change in the state (line 39 or line 45), and it is evident by the state diagram that when returning from the method, op is no longer in the r -th ACQUIRE phase. The third case is after verifying all the nodes with color c in the r -th data set memento are owned by op (line 43). Now, if when returning from the method some of the nodes are not owned by op , then, by (4), they are released by the operation that owned them, while it is in RELEASE phase, so this case also satisfies the condition. ■

Linearizability follows directly from these properties. The next theorem applies for both CAS-Chromo and DCAS-Chromo, since they follow the same scheme.

Theorem 4.3 (Linearizability) *CAS-Chromo and DCAS-Chromo are linearizable.*

Proof: Similarly to the previous chapter, we prove the theorem by identifying, for every operation, a *linearization point* inside its interval, so that the operation appears to occur atomically at this point. The linearization point of an operation op_i is either at the transition to state `(last, FINAL, INVALID)` (line 8), or at the transition to state `(last, RELEASE, SUCCESS)` (line 25); that is, when the CAS of the transition is applied successfully (line 151, marked LP1, or line 141, marked LP2, respectively). Only one of these occurs in an execution of an operation, and the linearization point is well defined.

In the first case, op_i discovers that the *source* node is invalid, since another operation op_j removes it. Before its transition to the FINAL phase, op_i helps op_j (line 7). By

Lemma 4.1, the transition to the APPLY phase of op_j already occurred and op_i helps it to complete in case it has not completed yet. Thus, op_i need not apply its changes, and it is linearized after op_j .

In the second case, Lemma 4.2 (1) and (2) imply that when the transition to the APPLY phase occurs in configuration C , all the nodes in the data set memento of op_i are valid, have not changed since they were cloned (except for their owners) and they are owned by op_i . By Lemma 4.2(4), these nodes remain unreleased while op_i is in the APPLY phase, which means, by Lemma 4.2(3), that no other operation changes these nodes and they are modified only by op_i during the execution of the APPLY phase. Finally, the `applyChanges` methods clearly preserve the specification of the corresponding doubly-linked list operations. ■

4.4 Progress and Locality Proofs

In this section, the legality of the coloring is used to show that the algorithms are local nonblocking. Formally, a node is *legally colored* if its color is different from the colors of its neighbors; the left anchor is legally colored if its color is different from its right neighbor, and analogously for the right anchor.

4.4.1 DCAS-Chromo is Local Nonblocking

It is simple to see that all operations access three consecutive nodes in the linked list, and that each operation only changes the color of a single node: insert and push operations change the color of the new node, and a pop and remove operations change the color of the right node in their data set. No operation changes the color of the left node in its data set. Since an operation only changes a node while owning it, this ensures that the colors of two adjacent nodes is not changed at the same time, even if concurrent operations access them.

Recall that a node is valid if it is an anchor or both its left and right links are not null. The coloring property of the algorithm is stated in the following lemma.

Lemma 4.4 *All valid nodes are legally colored.*

Proof: By Theorem 4.3, we can assume that the changes are applied in isolation from other operations. Therefore, the proof of this lemma is merely a step-by-step sequential analysis of the `applyChanges` methods of the various operations.

The proof is by induction on the execution order. In the base case, the linked list is empty: the left anchor is colored c_1 and the right anchor is colored c_3 , and hence, they are legally colored.

Induction step: a node can become illegally colored only when some operation applies its changes to the node or one of its neighbors. By Lemma 4.2(3), an operation

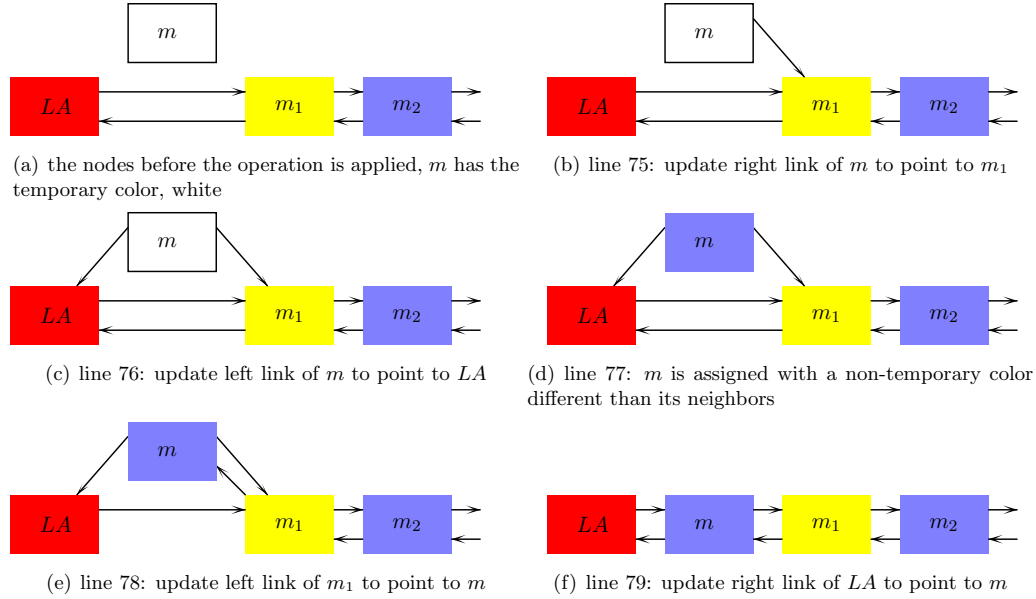


Figure 4.7: An example of an execution of the `applyChanges` method of `PushLeft` operation (see Pseudocode 4.3). Given the nodes LA, m_1, m_2 from Figure 4.2, op_1 pushes a new node, m , into the left side of the list.

changes a node only if it holds it. This implies that no node is inserted or removed immediately to the left or to the right of an operation data set while the operation applies its changes. Moreover, by the above observation a pop and remove operations only change the color of the right node in the data set and an insert or push operations only change the color of the new, i.e., middle, node in the data set. Thus, we can derive the next claim:

Claim 4.5 *An operation changes a node's color only if it holds the node and its left neighbor.*

We analyze every step in the APPLY phase of an operation, and we show that after each such step the node that was changed is still legally colored. Consider first the `PushLeft` operation presented in Figure 4.7. The data set of the operation, op_1 , is the new node (m), the left anchor (LA), and its right neighbor (m_1). While op_1 is applying its changes, other operations neither remove nor insert nodes to the right of the right neighbor node, and also do not change the colors of the nodes in the data set. Claim 4.5 imply that other operations do not change the color of an additional right neighbor (m_2). For `PushRight` or insert operations, the induction assumption also implies that a left neighbor is legally colored even if its color is changed by another operation. It remains to show, by inspecting the code, that the changes applied by the operation keep the nodes legally colored.

- Line 75, update right link of the new node: the new node is not yet valid (Figure 4.7(b));
- Line 76, update left link of the new node: the new node is valid and it is legally colored (Figure 4.7(c));
- Line 77, the new node is assigned with a non-temporary color different than its neighbors and the new node is legally colored (Figure 4.7(d));
- Line 78, update left link of the right neighbor (m_1): the right neighbor has color different than the colors of the new node and the right neighbor (m_2), and thus it is legally colored (Figure 4.7(e)).
- Line 79, update right link of the left anchor (or the source node in the case of an *insertRight* operation): the left anchor has color different than the color of the new node (in the case of an *insertRight* operation the source node also has color different than the color of the left neighbor), and thus it is legally colored (Figure 4.7(f));

We next analyze the *PopLeft* operation (see Figure 4.8). The data set of the operation, op_2 , is the left anchor (LA) and its right neighbors (m_1 and m_2). While op_2 is applying its changes, other operations neither remove nor insert nodes to the right of the right neighbor, and also do not change the colors of the nodes in the data set. Claim 4.5 implies that other operations do not change the color of an additional right neighbor (m_3). For a *PopRight* operation, the induction assumption implies that the left neighbor is legally colored even if its color is changed by another operation. We show again that the operation's changes keep the nodes legally colored:

- Line 85, the right neighbor (m_2) is assigned with the temporary color: the right neighbor is legally colored, since the subject node (m_1) and the right neighbor (m_3) have colors different than the temporary color (Figure 4.8(b));
- Line 86, update right link of the left anchor (m_2): the left anchor has a non-temporary color, and thus it is legally colored (Figure 4.8(c));
- Line 87, update left link of the right neighbor: the right neighbor is legally colored, since the left anchor and the adjacent node to the right have colors different than the temporary color (Figure 4.8(d));
- Line 88, set right link of the subject node to null: the source node is now invalid (Figure 4.8(e));
- Line 89, set left link of the subject node to null (Figure 4.8(e));
- Line 90, the right neighbor is assigned with a non-temporary color different than its neighbors, and thus it is legally colored (Figure 4.8(f)).

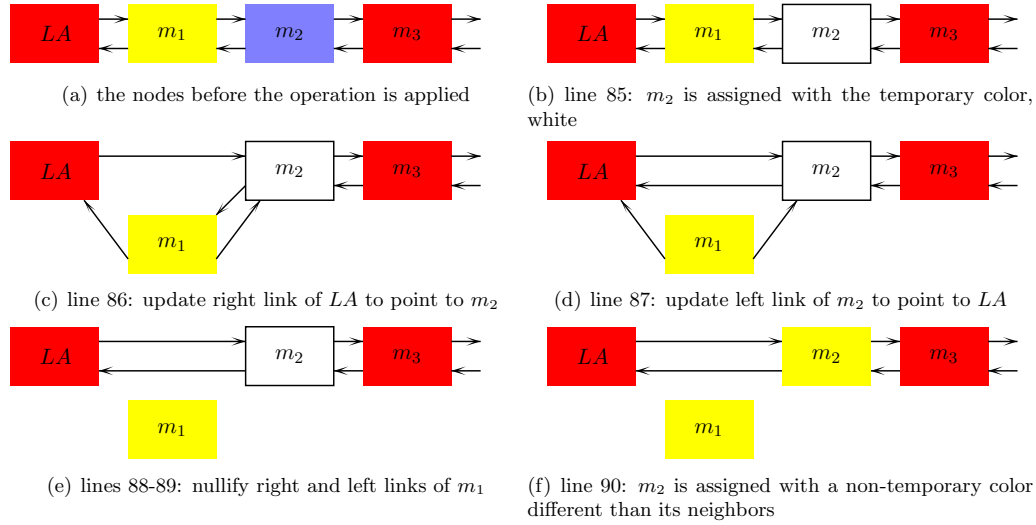


Figure 4.8: An example of an execution of the `applyChanges` method of `PopLeft` operation (see Pseudocode 4.3). Given the nodes LA, m_1, m_2, m_3 from Figure 4.2, op_2 pops the node m_1 .

It remains to show that the `remove` operation keeps the coloring legal (see Figure 4.9). The `remove` operation manipulates its data set in a manner similar to the `pop` operation. We analyze the `Remove` operation, op_4 , presented earlier in Figure 4.2. Figure 4.9(a) presents the nodes m_2, m_3, m_4, m_5, m_6 from Figure 4.2, op_4 removes the source node m_4 . The data set of the operation is the source node and its neighbors (m_3 and m_5). During the `ACQUIRE` phase, op_4 first acquires m_4 , and then atomically acquires m_3 and m_5 , which are equally colored. While op_4 is applying its changes, other operations neither remove nor insert nodes to the left of the left neighbor and to the right of the right neighbor, and also do not change the colors of the nodes in the data set. Claim 4.5 implies that other operations do not change the color of an additional right neighbor (m_6). Moreover, the left neighbor (m_2) is legally colored even if its color is changed by another operation, while op_4 is applying its changes. We omit the detailed description of this operation as it follows the lines of the description of the `PopLeft` operation.

We show that the operation's changes keep the coloring legal:

Line 186, the right neighbor (m_5) is assigned with the temporary color: the right neighbor is legally colored, since the source node (m_4) and the right neighbor (m_6) have colors different than the temporary color (Figure 4.9(b));

Line 187, update right link of the left neighbor (m_3): the left neighbor has a non-temporary color, different than the color of the left neighbor (m_2), and the temporary color of the right neighbor, and thus it is legally colored (Figure 4.9(c));

Line 188, update left link of the right neighbor: the right neighbor is legally colored,

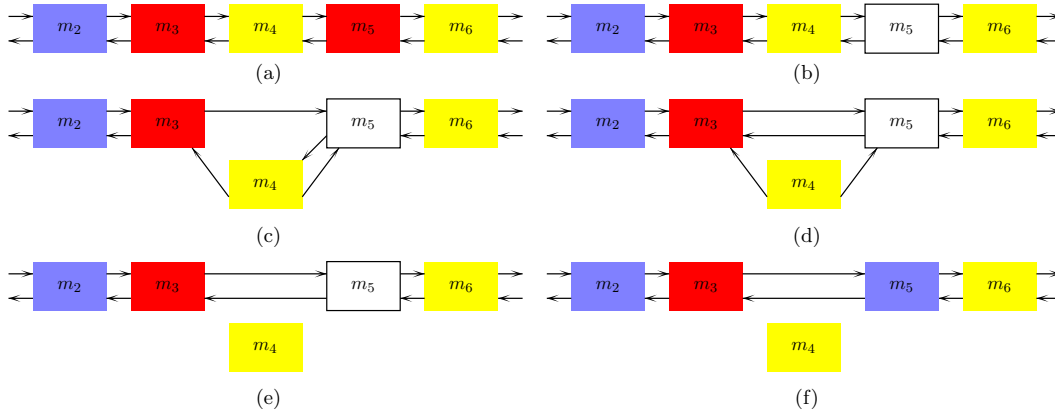


Figure 4.9: An example of an execution of a *Remove* operation— op_4 from Figure 4.2.

since the left neighbor and the right neighbor have colors different than the temporary color (Figure 4.9(d));

Line 189, set right and left links of the source node to null: the source node is now invalid (Figure 4.9(e));

Line 190, set right and left links of the source node to null (Figure 4.9(e));

Line 191, the right neighbor is assigned with a non-temporary color different than its neighbors: the right neighbor is legally colored (Figure 4.9(f)). ■

An operation acquires at most three consecutive nodes, and by the lemma, at most two of them have the same color, hence, DCAS suffices to acquire equally colored nodes in the operations' data set.

The proof that the algorithm is local nonblocking starts by showing that a process only helps operations within constant distance of the operation it is executing. In the proof, we consider the number of `help` methods the process started executing but have not yet completed. Recall the nodes are colored with 4 colors, $c_0 < c_1 < c_2 < c_3$.

Lemma 4.6 *Consider process p that called $h > 0$ help methods and completed $h' < h$ of them. If the last call is the $help(r_i)$ method of an operation op_i , then the r_i -th round of op_i acquires a node with color greater than or equal to $c_{h-h'-1}$.*

Proof: The proof is by induction on $k = h - h' - 1$; the base case is when $k = 0$. A node with color c_0 is either a new node that is acquired by the operation during its initialization, or the right node in a remove operation that is colored c_0 during the APPLY phase after all the data set is acquired by the operation.

In the induction step, $k > 0$, and since $h - h' \geq 2$, p called `help` method for at least one operation other than op_i and did not complete. Assume the penultimate `help` method called and not completed by p is for operation op_j . When the `help` method of

op_j was called, op_j already acquired color $\geq c_{k-1}$, by the induction assumption, and it tries to acquire node t with color $c > c_{k-1}$. Process p reads t 's owner (line 42), and discovers op_j failed to acquire t (line 43) and that t is consistent with its memento (lines 44-45), i.e., its color did not change. Then, p discovers that op_j is blocked by operation op_i in its r_i -th round (line 46) and calls the last `help` method (line 47). Since equally colored nodes are acquired atomically, op_i acquired color $c \geq c_k$, and the lemma follows. ■

Note that $h - h' - 1$ bounds from above the distance to operations that a process helps. Thus, Lemma 4.6 implies:

Corollary 4.7 *If op_i helps op_j , at distance d , then op_j already acquired a node with color greater or equal to c_d . In particular, op_j is in the 3-neighborhood of op_i , and if $d = 3$ then op_j completed the ACQUIRE phase.*

The local contention property immediately follows

Theorem 4.8 (Local contention) *DCAS-Chromo has 7-local contention.*

Proof: Two processes p_i and p_j access the same memory location if they help execute operations, op_k and op_l respectively, within distance one. By Corollary 4.7, op_i is in the 3-neighborhood of op_k and op_j is in the 3-neighborhood of op_l . Thus, the distance between op_i and op_j is at most 7. ■

We proceed to prove the local nonblocking property of the algorithm using Corollary 4.7. Once an operation is in its APPLY phase, it is straightforward that it completes after one of its executing processes takes a constant number of steps. It remains to prove that if the operation is blocked outside the APPLY phase, then some “nearby” operation (in a sense made precise by Lemma 4.9) completes. We do so by considering executions of the loop of `acquireColor(c, r)` method (lines 36-47), called a *c-acquiring iteration*. Lemma 4.9 below shows that in every acquiring iteration of an executing process, whether successful or not, some “nearby” operation makes progress.

Fix an arbitrary operation op_b initiated by process p_b ; progress in the neighborhood of op_b is tracked by three counters:

- The *completed operations counter*, denoted co , initially 0, is increased whenever an operation in the 4-neighborhood of op_b completes.
- The *color counter*, denoted cl , holds the color of the last acquiring iteration that the initiator of op_b executed.
- The *changes counter*, denoted ch , initially 0, is increased whenever an operation in the 5-neighborhood of op_b changes an item in its data set.

The values of the counters in a configuration C are denoted $co(C)$, $cl(C)$, $ch(C)$. Note that co and ch are nondecreasing, while cl is not necessarily monotone.

Assume that process p_b takes an infinite number of steps, executing infinitely many acquiring iterations, without completing op_b . Let the configurations at the start of these acquiring iterations be denoted $C_0, C_1, \dots, C_t, \dots$, in the order they occur. The next lemma argues that each configuration C_t is a milestone in the progress of the operations in the neighborhood of op_b :

Lemma 4.9 *For every $t > 0$, the value of at least one of the counters co , cl , or ch at configuration C_t is strictly larger than the value of the corresponding counter in configuration C_{t-1} .*

Proof: Assume that process p_b starts a c -acquiring iteration at C_{t-1} , during the $acquireColor(c, r)$ method of op , the j -th operation of p_i , and a c' -acquiring iteration at C_t , during a $acquireColor(c', r')$ method of op' , the j' -th operation of $p_{i'}$. By Corollary 4.7, op and op' are in the 3-neighborhood of op_b .

Consider first the case that $i \neq i'$, i.e., the operations are issued by different processes.

If op completes before the second iteration, then $co(C_{t-1}) < co(C_t)$. Otherwise, the first acquiring iteration of op failed. If the failure is due to contention, then some operation op_l at distance one from op applied a change to its data set; since op_l is in the 4-neighborhood of op_b , $ch(C_{t-1}) < ch(C_t)$. Otherwise, the first acquiring iteration of op failed since op' blocked it. That is, op fails to acquire a node t with color c since it is already owned by op' , and then p_b helps op' and executes the second c' -acquiring iteration. Since op' atomically acquires all the nodes with color c , $c < c'$ and hence, $cl(C_{t-1}) < cl(C_t)$.

Now, consider the case that $i = i'$, i.e., both operations are issued by the same process, and therefore, $j \leq j'$. If $j < j'$, p_i completed its j -th operation and $co(C_{t-1}) < co(C_t)$.

Otherwise, $j = j'$, i.e., both acquiring iterations are of the same operation. The round number of the acquiring iterations is monotonically increasing, thus, $r \leq r'$. If $r < r'$, then op is re-invoked before the second acquiring iteration, due to contention in the first iteration. Thus, in the first iteration some operation op_l at distance one from op applied a change to its data set that failed op . The operation op_l is in the 4-neighborhood of op_b and $ch(C_{t-1}) < ch(C_t)$.

Otherwise, $r = r'$, i.e., both acquiring iterations are of the same round. The colors p_b is acquiring in the same round of the same operation are nondecreasing, thus, $c \leq c'$. If $c < c'$ then $cl(C_{t-1}) < cl(C_t)$.

Finally, we are left with the case that $i = i'$, $j = j'$, $r = r'$, and $c = c'$, that is, two consecutive c -acquiring iterations in the same round of the same operation. The process executing the acquiring iterations, p_b , fails to acquire some node t with color

c in the first iteration. Then, without helping any other operation (since the node is already released), p_b retries the acquiring iteration. The process p_b fails to acquire t in the first iteration since another operation op_l , in the 1-neighborhood of op , owns t . The operation op_l is in the 4-neighborhood of op_b . If op_l releases t after it completes, then $co(C_{t-1}) < co(C_t)$. Otherwise, the node is released since op_l discovers that a node it is trying to acquire, t' , is inconsistent with its memento. Thus, another operation op_k in the 1-neighborhood of op_l changed t' after op_l generated the memento of t' . Since op_k is in the 5-neighborhood of op_b , $ch(C_{t-1}) < ch(C_t)$. ■

Theorem 4.10 *DCAS-Chromo is a 4-local nonblocking implementation of a doubly-linked list.*

Proof: Consider the initiator p_b of an operation op_b . By Lemma 4.9, at least one counter increases with each acquiring iteration of p_b . Since the color counter is at most 3, after at most three consecutive acquiring iterations, some counter other than the color counter must increase. If the completed operations counter increases, then some pending operation in the 4-neighborhood of op_b completes. Otherwise, the changes counter increases. Once it is in the APPLY phase, an operation completes within a constant number of changes. Thus, after p_b executes a number of acquiring iterations that is linear in the number of operations in the 5-neighborhood of op_b , some pending operation in the 4-neighborhood of op_b completes. Since there is a finite number of processes, it follows that after p_b takes a finite number of steps, some operation in the 4-neighborhood of op_b completes. ■

4.4.2 CAS-Chromo is Local Nonblocking

CAS-Chromo does not support removals from the middle of the linked list, and hence only *pop* operations acquire three nodes, which may include two nodes with the same color. Since operations acquire equally colored nodes by their order in the list from left to right, whenever a process calls a new *help* method it helps a new operation. Thus, the number of *pop* operations a process started helping and did not complete is at most two, and helping cycles are avoided.

We revise Lemma 4.6 and Corollary 4.7 in order to prove the locality properties of CAS-Chromo.

Lemma 4.6' *Consider process p that called $h > 0$ help methods and completed $h' < h$ of them, such that from the $h - h'$ uncompleted help methods, ℓ are of pop operations. If the last call is the $help(r_i)$ method of an operation op_i , then the r_i -th round of op_i acquires a node with color greater than or equal to $c_{h-h'-1-\ell}$.*

Proof: The proof is by induction on $k = h - h' - 1$. The base case, $k = 0$, follows by arguments similar to those applied in the base case of the proof of Lemma 4.6.

In the induction step, $k > 0$ and since in this case, $h - h' \geq 2$, p called the help method of least one operation other than op_i did not complete. Assume the penultimate help method called and not completed by p is the help method of operation op_j .

We first assume that op_i is a pop operation. By the induction assumption, when the help method of op_j is called, op_j owns a color greater or equal to $c_{k-1-(\ell-1)} = c_{k-\ell}$. Process p helps op_j to acquire a node t with color $c > c_{k-\ell}$. We review p 's steps while helping op_j to show that op_i acquired color c : p reads t 's owner (line 42), and discovers op_j failed to acquire t (otherwise it returns in line 43) and that t is consistent with its memento (line 44), i.e., its color is still c . Then p discovers that op_j is blocked by operation op_i in its r_i -th round (line 46) and calls the last help method (line 47). The operation op_i acquired t with color c in its r_i -th round, and the lemma holds.

If op_i is not a pop operation, the induction assumption implies that when the help method of op_j is called, op_j already owns color greater or equal to $c_{k-1-\ell}$. Process p helps op_j to acquire some node t with color $c \geq c_{k-\ell}$ and takes the same steps as in the previous case, to find that op_j is blocked by op_i in its r_i -th round, then it calls the last help method. The operation op_i , which is not a pop operation, has only one node t in its data set with color $c \geq c_{k-\ell}$, which it acquired in its r_i -th round, and the lemma holds. ■

Corollary 4.7' *If op_i helps op_j , at distance d , then op_j already acquired a node with color greater or equal to c_{d-2} . In particular, op_j is in the 5-neighborhood of op_i , and if $d = 5$ then op_j completed the ACQUIRE phase.*

The revised lemma and corollary imply that CAS-Chromo has 11-local contention, in a manner similar to the proof for DCAS-Chromo. We next argue that the worst-case scenario, in terms of delay and helping chains, is the one presented in Figure 4.10. This implies a better bound, of 5-local contention. Assume op_0 is a *PushLeft* operation, op_1 is a *PopLeft* operation, op_2 and op_3 are *InsertRight* operations, op_4 is a *PopRight* operation and op_5 is a *PushRight* operation. Consider an execution in which op_1 acquires the left anchor, op_2 acquires m_2 , op_3 acquires m_3 , op_4 acquires m_4 , and op_5 acquires m_5 and the right anchor. The operation op_5 is in the 5-neighborhood of op_0 ; these operation contend while accessing the right anchor: op_0 tries to acquire the left anchor and thus it helps op_1 ; op_1 tries to acquire m_2 and thus it helps op_2 ; op_2 tries to acquire m_3 and thus it helps op_3 ; op_3 tries to acquire m_4 and thus it helps op_4 ; op_4 tries to acquire the right anchor and thus it helps op_5 . This implies:

Theorem 4.11 *CAS-Chromo is an implementation of a doubly-linked list, allowing removals only at the ends, that has 5-local contention and is 5-local nonblocking.*

An implementation of the deque data structure provides operations only at the ends. Therefore, the worst case scenario has a shorter linked list consisting only the

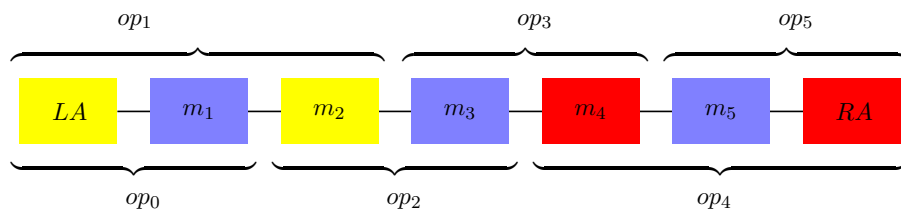


Figure 4.10: CAS-Chromo worst case scenario for overlapping deque operations.

nodes m_1 , m_2 and m_5 (and the anchors), and the execution only of operations op_0 , op_1 , op_4 , and op_5 in Figure 4.10, where op_1 and op_4 access a common node, m_2 . The analysis can be further tightened to show:

Theorem 4.12 *CAS-Chromo is an implementation of a deque that has 3-local contention and is 3-local nonblocking.*

Chapter 5

Related Work

Many implementations for concurrent data structures were proposed; in part, Table 5.1 lists implementations of linked lists. Most implementations use unary primitives, but some apply, in addition, primitives that may change two memory locations in one step, e.g., DCAS. In [35], the authors argue that primitives more powerful than DCAS, e.g., 3CAS, are needed in order to obtain simple and efficient implementations of data structures guaranteeing that some operation makes progress at any time. Our results, for multi-word synchronization, as well as for doubly-linked lists, leverage DCAS and indicate that it suffices for these purposes, and it may have a significant role in facilitating concurrent programming.

If DCAS is not provided by the architecture, using an $O(\log^* n)$ -local nonblocking implementation of DCAS from CAS [8] in our multi-word algorithm yields a CAS-based k RMW implementation that is $O(k + \log^* n)$ -local nonblocking. Likewise, integrating this implementation into our DCAS-Chromo yields a doubly-linked list implementation from CAS that is $O(\log^* n)$ -local nonblocking.

Next, we overview previous implementations of linked lists, and compare them to our algorithms

Harris [51] used CAS to implement a singly-linked list, with insertions and removals anywhere; however, in this algorithm, a process can access a node previously removed from the linked list, possibly yielding an unbounded chain of uncollected garbage nodes. Michael [73] fixed these memory management issues. Elsewhere [74], Michael proposed an implementation of a deque; in this algorithm, a single word (called *anchor*) holds the head and tail pointers, causing all operations to interfere with each other, and making the implementation inherently sequential. Sundell and Tsigas [97] avoid the use of a single anchor, allowing operations on the two ends to proceed concurrently. They extend the algorithm to allow insertions and removals in the middle of the list [96]; in the latter algorithm, a long path of overlapping removals may cause interference among distant operations; moreover, during intermediate states, there can be a consecutive sequence of inconsistent backward links, causing part of the list to behave as singly-

Algorithm	Insertions	Removals	Uses DCAS	Interference	Comments
Harris [51]	anywhere	anywhere	no	any pair	singly-linked list
Greenwald [43]	anywhere	anywhere	yes	any pair	
Michael [73]	anywhere	anywhere	no	any pair	singly-linked list
Sundell and Tsigas [96]	anywhere	anywhere	no	any pair	
DCAS-Chromo	anywhere	anywhere	yes	distance ≤ 7	
Greenwald [42]	ends	ends	yes	opposite ends	
Agesen et al. [6]	ends	ends	yes	distance = 1	
Michael [74]	ends	ends	no	opposite ends	
Herlihy et al. [54]	ends	ends	no	distance = 1	obstruction free; array-based
Sundell and Tsigas [96]	ends	ends	no	distance ≤ 2	
CAS-Chromo	ends	ends	no	distance ≤ 3	
CAS-Chromo	anywhere	ends	no	distance ≤ 5	

Table 5.1: Comparison of linked list algorithms; *interference* indicates which operations may delay other operations.

linked. In an *obstruction-free* [54] implementation operations terminate successfully only when eventually executing solo for long enough. An obstruction-free deque, was proposed by Herlihy et al. [54]; besides blocking when there is even a little contention, this array-based implementation bounds the deque’s size.

Greenwald [42, 43] suggests to use DCAS to simplify the design of many data structures. His implementations of deques, singly-linked and doubly-linked lists synchronize via a single designated memory location, resulting in a strictly sequential execution. Agesen et al. [6] present the first DCAS-based, dynamically-sized deque implementation supporting concurrent access to both ends of the deque, and has 1-local step complexity; this algorithm does not allow operations in the middle of the linked list. SNARK [28] is an attempt for further improvement that uses only a single DCAS primitive per operation in the best case, instead of two. Unfortunately, SNARK is incorrect and the corrected version allows removed nodes to be accessed from within the deque, thus preventing the garbage collector from reclaiming long chains of unused nodes [35]. In our algorithms, an operation completes within $O(1)$ steps in an execution suffix if it is running solo, i.e., it has constant *obstruction-free step complexity* [38]. Our algorithms do not leave accessible chains of stale “garbage” nodes.

Alongside these handcrafted implementations, generic techniques simulating multi-word synchronization can be used to systematically derive implementations of data structures, such as linked lists, from arbitrary lock-based algorithms (see Table 5.2).

Algorithm	Locality	Progress	Space per item	Uses DCAS	Dynamic
Turek et al. [98]	$O(n)$	local nonblocking	$O(1)$	no	no
Barnes [20]	$O(n)$	local nonblocking	$O(1)$	no	no
Shavit and Touitou [91]	$O(n)$	blocking	$O(1)$	no	no
Afek et al. [5]	$O(k + \log^* n)$	local nonblocking	$O(k)$	no	no
Harris et al. [52]	$O(n)$	local nonblocking	$O(1)$	no	yes
Herlihy et al. [55]	$O(n)$	obstruction free	$O(1)$	no	yes
BLocalRMW (this paper)	$O(k)$	blocking	$O(1)$	yes	can be
BLocalRMW using [8]	$O(k + \log^* n)$	blocking	$O(1)$	no	can be
LocalRMW (this paper)	$O(k)$	local nonblocking	$O(1)$	yes	can be
LocalRMW using [8]	$O(k + \log^* n)$	local nonblocking	$O(1)$	no	can be

Table 5.2: Comparison of multi-word synchronization algorithms, showing locality and progress properties, as well as the space complexity per item. The table also indicates whether the algorithm uses DCAS, and whether the algorithm is or can be made to be dynamic.

The first implementations of multi-word synchronization that use helping are the “locking without blocking” schemes [20,98]: an operation starts by acquiring the data items in its data set (ACQUIRE phase); then, the changes are applied on these data items (APPLY phase); finally, the operation releases the items (RELEASE phase).

In these schemes operations recursively help other operations, without releasing the items they have acquired; these algorithms are $O(n)$ -local nonblocking. During the ACQUIRE phase, an operation may hold one or more items while waiting for another operation to release another item. The latter operation might also be waiting for a third operation to release an item, leading to a hold-and-wait chain of operations.

As was described, helping is *recursive*, possibly causing long *helping chains*. For example, consider the list in Figure 5.1, and assume the nodes are acquired by the

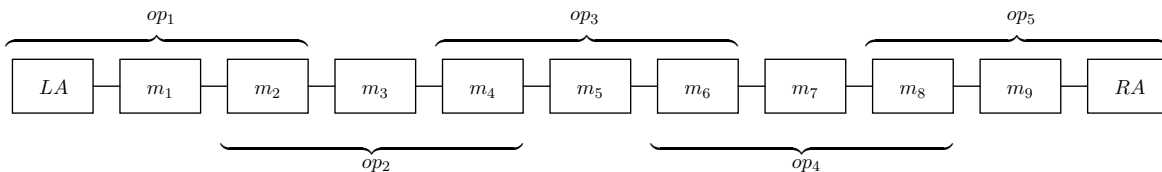


Figure 5.1: Using generic schemes to execute the operations depicted in Figure 4.6 yields long helping chains.

operations from left to right. Consider an execution α in which op_1, op_2, op_3, op_4 and op_5 concurrently acquire their two left-most nodes successfully, and then op_1 tries to acquire m_2 while the other operations are delayed. Since m_2 is owned by op_2 , op_1 helps op_2 ; since m_4 is owned by op_3 , op_1 helps op_3 ; since m_6 is owned by op_4 , op_1 helps op_4 ; and since m_8 is owned by op_5 , op_1 helps op_5 . Thus op_1 is delayed by a chain of conflicting operations. In general, op_1 can be delayed by any operation within finite distance from it, implying that the implementation is not local.

Static software transactional memory [91] also implements multi-word synchronization. Operations acquire items by the order of their memory addresses, and help only operations at distance 1. Nevertheless, it is $O(n)$ -local nonblocking, as demonstrated by the following example. Consider again an execution that starts with op_1, op_2, op_3 and op_4 acquiring their low-address nodes successfully, then op_1 fails to acquire m_2 , op_2 fails to acquire m_4 , and op_3 fails to acquire m_5 ; each operation then helps its (immediate) neighbor. Prior to helping, op_1, op_2 and op_3 release their nodes, thus op_1 and op_2 discover their help is unnecessary. Assume that op_4 completes, and again op_1, op_2 and op_3 try to acquire their data sets. It is possible that op_1, op_2 and op_3 acquire their low-address nodes, and op_1 tries, in vain, to help op_2 , which releases its nodes due to op_3 , etc. As the length of the path of overlapping operations increases, the number of times op_1 futilely helps op_2 increases as well.

The *color-based* scheme for binary operations [8] bounds the length of helping chains by coloring the data items with ordered colors. An operation starts by coloring the nodes it is going to access with a constant number of colors, so that neighboring nodes have different colors, and then acquires data items in an increasing order of colors. In this scheme, op helps op' only if op' already owns a higher color. It can be shown that the length of helping chains is bounded by the number of colors, and an operation helps only operations at constant distance.

Afek et al. [5] present a CAS-based implementation of k RMW that is $O(k + \log^* n)$ -local nonblocking,¹ matching the locality properties of the CAS-based version of LocalRMW. Their implementation works recursively in k , going through the items according to their memory addresses, and coloring the items before proceeding to acquire them; at the base of the recursion (for $k = 2$), it employs the DCAS implementation of Attiya and Dagan [8]. Due to its recursive structure, the algorithm is quite complicated, making it hard to derive detailed pseudocode and correctness proof, which are not provided in their paper.

More importantly, the recursive structure of the implementation of [5] requires to store $O(k)$ information in each data item, and to hard-wire k , uniformly for all operations. In contrast, LocalRMW stores a fixed amount of information per data item, regardless of k ; in fact, it can be modified so that each operation accesses a

¹They state $O(\log^* n)$ -local complexities, treating k as a constant.

different number of data items. Moreover, the implementation of [5] acquires items in increasing order and performs preparatory calculation (coloring) on them, implying that it must receive all items when it starts; i.e., it is inherently *static*. In contrast, our implementation does not depend on the memory addresses of the items and can be made to work when data items are given one-by-one.

Other implementations of dynamic multi-word synchronization operations, such as [52], use recursive helping, and are $O(n)$ -local nonblocking. DSTM [55] provides multi-word synchronization, which is dynamic and does not use helping: a blocked transaction releases its items and retries. However, DSTM has $O(n)$ failure locality in the scenario given for [91], modified so that instead of completing, the transaction at the end of the chain stops taking steps. In this scenario, transactions that stop taking steps can cause a transaction at distance $O(n)$ to retry over and over again. Moreover, DSTM provides only the weaker progress property of *obstruction-freedom*.

Reviewing Table 5.2 shows that while generic schemes can be used to derive list-based data structures, the resulting implementations when using most of these techniques [20,52,55,91,98] incur $O(n)$ locality, meaning that any pair of operations within distance n may interfere with each other. Using the colored-based schemes [5,8] yields implementations that are $O(\log^* n)$ -local nonblocking. These schemes must color the nodes at the beginning of each operation, leading to complicated implementations. By keeping the coloring legal, our linked list algorithms render this initial coloring obsolete, thereby avoiding its cost. In particular, our progress proofs imply that CAS-Chromo is 3-local nonblocking, and DCAS-Chromo is 4-local nonblocking.

Several STMs use a designated *contention manager* for deciding how to handle conflicts. Like our BLocalRMW and LocalRMW, some contention managers, e.g., Size-Matters [87], Karma and Polka [88], arbitrate between conflicting transactions based on the number of acquired items or bytes accessed. These contention managers, however, do not address symmetry breaking in the case of equal progress, and scenarios similar to the one given for [91] can create long *delay chains*. Also, they neither state nor prove analytical bounds on their progress and locality.

Schneider and Wattenhofer [90] evaluate a contention manager by its *makespan*, i.e., the total execution time of all operations. Their analysis implies that in our example of overlapping operations, a chain of length n can yield $O(n)$ makespan. They use randomization to break symmetry and improve the locality of contention management, but their algorithm is still with high probability $O(\log n)$ away from the optimum, thus not having constant locality.

Part II

Inherent Limitations for Transactional Memory

Chapter 6

The Transactional Memory Approach

In this chapter, we present the basic notions that are required for our impossibilities and lower bounds results; it is not a comprehensive presentation of the transactional memory model.

Transactional memory (TM) encapsulates high-level abstract data items, as in abstract data types. A *transaction* in a TM is a sequence of high-level operations executed by a single process on a set of data items. It is guaranteed that if a transaction commits, then all its operations appear to be executed atomically, as one indivisible operation. The collection of data items accessed by a transaction is the transaction's *data set*; in particular, the items written by the transaction are its *write set*, and the items read by the transaction are its *read set*.

A complete interface of a TM includes high-level read and write operations, as well as two special high-level operations, *try-commit* and *abort*. Specifically, a *read* operation specifies the item to read, and returns the value read by the operation or an abort indication; a *write* operation specifies the item and value to be written and might return an abort indication; a *try-commit* operation returns an indication whether the transaction committed or aborted; and an *abort* operation returns an indication that the transaction aborted. If while reading, writing or trying to commit, the operation returns an abort indication, e.g., due to a conflict with another transaction then the transaction is *forcibly aborted*.

As opposed to abstract operations of an abstract data type, there is no predefined semantics for a transaction; the semantics of a transaction comes from atomically aggregating its sequence of high-level operations. Figure 6.1 depicts the concurrent execution of two transactions. Every transaction begins with a sequence of read and write operations; either a try-commit operation or an abort operation can be invoked once during the execution of a transaction, as its last operation. When the last operation of a transaction is a try-commit operation and it returns a commit indication

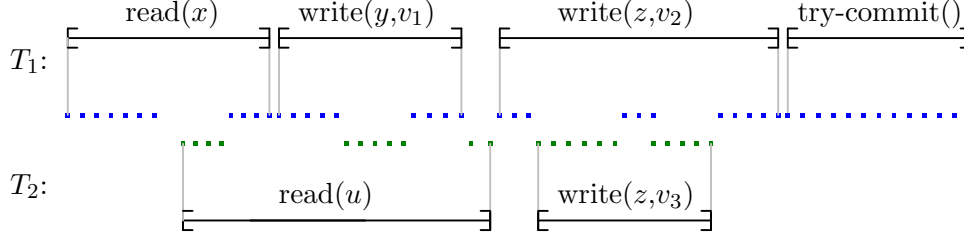


Figure 6.1: An example of transactions execution. Two level of abstraction: square brackets denote invocations and responses of high-level operations; dotted lines denote low-level steps on base objects by T_1 and T_2 .

the transaction is *committed*; when the last operation returns an abort indication the transaction is *aborted*; otherwise, the transaction is *pending*. Consider for example, the two transactions depicted in Figure 6.1. If the last operation of one of the transactions returns an abort indication then the transaction is (forcibly) aborted. Otherwise, T_1 is committed, and T_2 is pending.

In this part, we use the term operations to refer to high-level operations of transaction (unlike Part I, where operations referred to the abstract operations of an ADT), we still reserve the term primitives for the low-level steps of the implementation.

6.1 Software Implementation of TM

A *software implementation* of transactional memory (*STM*) provides data representation for transactions and data items using base objects, and algorithms, specified as primitives for the high-level read, write, try-commit (abbreviated, commit), and abort operations.

An execution of a transaction encompasses two levels of abstraction: The high-level has a sequence of high-level operations accessing data items. At the low-level, these operations are translated into executions in which a sequence of events apply primitives to base objects. Figure 6.1 presents these two levels of abstractions: The upper dotted lines are the low-level steps of the operations of T_1 ; the lower dotted lines are the low-level steps of the operations of T_2 ; the execution is the interleaving of these steps.

Since our main results for transactional memory are impossibility results, we concentrate on the low-level executions and do not elaborate further on the manner a transaction issues its operations; this only makes our impossibility results stronger.

Two low-level executions α_1 and α_2 , containing steps on base objects, are *indistinguishable* to a process p , denoted $\alpha_1 \stackrel{p}{\sim} \alpha_2$, if p goes through the same sequence of state changes in α_1 and in α_2 ; in particular, this implies that it executes the same primitives on base objects, and receives the same return value from those primitives, in both executions.

6.2 Safety Properties

In a *serial execution*, the transactions are executed to completion in isolation one after the other (see Section 2.1). An STM is *serializable* [78] if any, finite or infinite, execution has a *committed projection execution* attained by discarding all pending and aborted transactions such that the projection execution is equivalent to some serial execution of the committed transactions; we assume that this *serialization* order preserves the *per-process order*, i.e., transactions of the same process maintain their order.

Since the definition applies also to infinite execution, it implies a liveness condition. If transactions by the same process read a data item over and over again they eventually return a “fresh” value: the last value written to the item either before the first operation reading the item or after the last value read by the process was written, whichever comes later. Traditional definitions of serializability (e.g., [78, 100]), however, apply only to finite executions, and hence, admit non-live implementations, where read operations may miss values written to data items.

An STM is *strictly serializable* if the serialization order preserves the order of non-overlapping transactions [78]; this notion is called *order-preserving serializability* in [100], and is the analogue of *linearizability* [58] for transactions. Note that in this case, the discussion of non-live implementations is irrelevant, since transactions reading an item must eventually, return the value written to the item by another transaction.

The consistency condition commonly used for transactional memory is *opacity* [47]; very roughly stated, opacity requires all transactions to appear to execute sequentially in an order that agrees with the order of non-overlapping transactions, and all transactions (including aborted ones) are internally consistent, i.e., are strictly serializable. Therefore, we only assume strict serializability for our lower bounds and do not present the formal definition of opacity; this only makes our impossibility results stronger.

6.3 Progress Properties

The strongest progress property is *wait-freedom* that guarantees that in every execution, a transaction completes after a finite number of steps of the process executing it. The common progress condition, however, used for non-locking TM implementations is the weaker *obstruction-freedom*, discussed in Chapter 5. It ensures that a transaction commits when its executing process is eventually executing solo for long enough.

Lock-based STMs may guarantee *progressiveness* [48]: An STM is *weakly progressive* if a transaction that does not encounter nontrivial conflicts¹ cannot be forcibly aborted; it is *strongly progressive* if, in addition, when a set of transactions have nontrivial conflicts on a single item then not all of them can be forcibly aborted.

¹A *conflict* occurs when two operations access the same data item; the conflict is *nontrivial* if one of the operations is a write.

A transaction *blocks* if it takes an infinite number of steps without committing or aborting. The following progress condition requires a transaction to commit if it has no nontrivial conflict with any pending transaction; that is, a transaction can abort or block only due to a nontrivial conflict with such a transaction. A transaction T is *logically committed* in a configuration C if T does not abort in any infinite extension from C ; it might be blocked after C , but if it is not then eventually, it commits.

Definition 6 (*l*-progressive STM) *An STM is l -progressive, $l \geq 0$, if a transaction T aborts or blocks in a solo execution (of all the transaction or a suffix of it) after an execution α that contains l or less incomplete transactions, only due to a nontrivial conflict with an incomplete logically committed transaction.*

A transaction that must commit according to this definition becomes logically committed at some point, at the latest, right before it commits. It means that, in the absence of conflicts, the STM must ensure parallelism. This property (for $l \geq 0$) is satisfied by weakly progressive STMs, and by obstruction-free STMs, as l -progressiveness implies a transaction must not abort or block if it runs solo after an execution without nontrivial conflicts.

Minimal progressiveness [64] guarantees a transaction is blocked or forcibly aborted only if it is concurrent to another transaction. More formally, a configuration C is *quiescent* if no transaction is pending in C , i.e., it is not inside the interval of any transaction. In a minimally progressive STM, a transaction terminates successfully if it runs alone from a quiescent configuration. This property (for $l \geq 0$) is satisfied by l -progressive TM implementations, as every l -progressive TM is also minimally progressive. In the other direction, every minimally progressive TM is 0-progressive.

An implementation is *permissive* with respect to a safety property [45] if it never aborts a transaction unless necessary for ensuring the safety property.

6.4 Disjoint-Access Parallelism

Disjoint-access parallelism [62] captures the intuition that transactions accessing disjoint parts of the data should not interfere with each other [57].

We consider the dynamic version of a data set and a conflict graph as defined in Chapter 4, where transactions replace the operations. For a configuration C , in which \mathcal{ST} is the set of states of the items, the data set of a transaction T in C is the union, over every state s in \mathcal{ST} , of the set of items accessed by T when executed from s . The vertices in the conflict graph of an execution interval α represent all the transactions whose execution intervals overlap with α ; edges represent conflicts between transactions, i.e., an edge connects two transactions if their data sets in α are not disjoint.

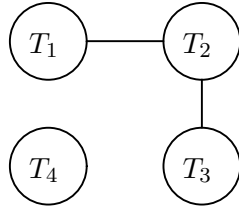


Figure 6.2: An example of a simple conflict graph: T_1 accesses items x , y and z ; T_2 accesses items u and y ; T_3 accesses items u and w ; T_4 does not access the items x , y , z , u and w .

Consider for example, in addition to T_1 and T_2 , depicted in Figure 6.1, two transactions: T_3 that reads item u and writes to item w , and T_4 that does not access the items x , y , z , u and w . Figure 6.2 shows the conflict graph of an execution of these 4 transactions.

Two transactions T_1 and T_2 are *disjoint access* if there is no path between them in the conflict graph of the minimal execution interval containing the intervals of T_1 and T_2 . In the example above, T_1 and T_4 are disjoint access, while T_1 and T_3 are not disjoint access as a path is connecting them in the conflict graph.

Two events *contend* on a base object o if they both access o , and at least one of them applies a non-trivial primitive to o . Two processes *concurrently contend* on a base object o if they have pending events at the same configuration that contend on o .

Definition 7 *An STM is weakly disjoint-access parallel if two processes p_1 and p_2 , executing transactions T_1 and T_2 , concurrently contend on the same base object, only if T_1 and T_2 are not disjoint access.*

This definition captures the first condition of the disjoint-access parallelism property of Israeli and Rappoport [62], in accordance with most of the literature (cf. [57]). Our requirement is weaker than theirs, as we allow two processes to apply a trivial primitive on the same base object when executing two transactions even if they are disjoint access. Moreover, our definition only prohibits concurrent contending accesses, allowing transactions to contend on a base object o at different points of the execution.

The original definition [62] also restricts the impact of concurrent transactions on the *step complexity* of a transaction; our results do not rely on this additional condition, again, as in the consistency condition, making them stronger.

Next we present a stronger notion of disjoint-access parallelism. It is stronger in the sense that it does not allow two disjoint access transactions to both apply even trivial primitives to the same base object. Two processes *concurrently access* a base object o if both have a pending access to o at some configuration.

Definition 8 *An STM is (strongly) disjoint-access parallel if two processes p_1 and p_2 , executing transactions T_1 and T_2 , concurrently access the same base object, only if T_1 and T_2 are not disjoint access.*

We illustrate the difference between these properties using the transactions in our example: if the STM is strongly disjoint-access parallel, then T_4 can not read any base object to which T_1 , T_2 or T_3 have a pending access, whereas, if the STM is weakly disjoint-access parallel, T_1 and T_4 can have pending reads to the same base object. In both cases, since T_1 and T_4 are disjoint access, they can not both have pending contending events to the same base object at any configuration.

Definition 7 can be interpreted as an unquantified variant of local contention (Chapter 2), as transactions in implementations that have d -local contention, for any finite d , do not access the same base objects if they are disjoint access; therefore, they do not contend on the same base objects, all the more so, do not concurrently contend on it.

Guerraoui and Kapalka [46] present a *strict* notion of disjoint-access parallelism. Two transactions are *disjoint* if they are not conflicting, i.e., their data sets are disjoint.

Definition 9 *An STM is strictly disjoint-access parallel if two processes p_1 and p_2 , executing transactions T_1 and T_2 , concurrently contend on the same base object, only if T_1 and T_2 are not disjoint.*

This notion is much stronger than the one originally proposed by Israeli and Rappoport [62], where two transactions with disjoint data sets are allowed to access the same base objects, provided they are connected via other transactions. All other transactions have to progress in parallel, even if they are concurrent. For example, transactions T_1 and T_3 , depicted in Figure 6.2, are not disjoint access; they are not allowed to contend on a base object in a strict disjoint-access parallel implementation as they are disjoint. Strict disjoint-access parallelism, like Definition 7, allows concurrent reads to the same base objects even by transactions that are not connected in the conflict graph.

6.5 Invisibility of Reads

An STM has *invisible reads*, if an execution of any transaction is indistinguishable from the execution of a transaction writing the same values to the same items, while omitting all read operations. More formally, consider an execution α that includes a transaction T of process p with write set W and read set R , and consider a transaction T' of process p writing the same values to W in the same order as in T , but with an empty read set. If the STM has invisible reads, then there is an execution α' that includes T' instead of T , such that α' is indistinguishable to all other processes from α .

Read-only transactions access the memory only through read operations, i.e., they have empty write sets. Implementations of read-only transactions that do not write to the memory *at all* and only apply trivial primitives to base objects are called *invisible*.

Chapter 7

Limitations on Disjoint-Access Parallel STMs

Software transactional memory is a popular approach for alleviating the difficulty of programming concurrent applications. Two fundamental properties of STM are *disjoint-access parallelism* and the *invisibility* of read operations, defined in the previous chapter. *Disjoint access parallelism* ensures that operations on disconnected data do not interfere, and thus it is critical for STM scalability. The *invisibility* of read operations means that their implementation does not write to the shared memory, thereby reducing memory contention.

We prove an inherent tradeoff for implementations of transactional memories: they cannot be both disjoint-access parallel and have read-only transactions that are invisible and always terminate successfully. In fact, a lower bound, linear in the number of items the transaction is reading, is proved for disjoint-access parallel STM implementation.

This chapter is organized as follows: Section 7.1 presents an impossibility result showing that in a disjoint-access parallel STM with invisible read-only transactions, some read-only transaction may never terminate successfully; this result is proved using only three processes. Section 7.2 strengthens this result and shows that a read-only transaction on t items (in a disjoint-access parallel STM with read-only transactions that always terminate successfully) must apply write primitives to $t - 1$ base objects; this result requires $t + 1$ processes. These proofs only assume *strict serializability*, and hence hold also under the assumption of opacity. Section 7.3 extends the results to hold even with the weaker conditions of *snapshot isolation* and *serializability*.

⁰The results of this chapter have appeared in [16].

7.1 Impossibility of Invisible Read-Only Transactions

We prove that in a disjoint-access parallel STM implementation with invisible read-only transactions, some read-only transaction will not terminate successfully in a finite number of steps; this is formally stated in Theorem 7.4.

Specifically, we construct an infinite execution of a read-only transaction. This execution consists of a single read-only transaction with one complete update transaction between any pair of consecutive steps by the read-only transaction; an *update* is a transaction with a singleton write set and an empty read set. We first define a special (finite) execution of this form, called *flippable*, and show that such a read-only transaction cannot terminate successfully. Then we show how a flippable execution can be repeatedly extended to construct successively longer flippable executions.

An execution is called flippable since there are two similar executions in which we flip the position of two update transactions and one of the executions is indistinguishable from the original execution. One type of flipped execution is called a *forward* flip since an update transaction is moved earlier in the execution, while other is called a *backward* flip since an update transaction is deferred in the execution. Formally:

Definition 10 *A flippable execution of length k with t updaters is a finite execution $E_k = U_0 s_1 U_1 \dots s_k U_k$ executed by processes p_0, \dots, p_{t-1} executing update transactions and process q executing a read-only transaction, which reads and returns the value of t data items $i_0 \dots i_{t-1}$. The execution E_k satisfies all the following conditions:*

1. for $j = 1, \dots, k$, s_j is a single step by q ,
2. for $j = 0, \dots, k$, U_j is a solo execution of a complete update transaction, in which process $p_h \in \{p_0, \dots, p_{t-1}\}$, writes $j + 1$ to the data item i_h
3. consecutive updates are executed by different processes, and
4. for any l , $0 < l \leq k$, the execution

$$E_k = U_0 s_1 U_1 \dots s_{l-1} U_{l-1} s_l U_l \dots s_k U_k$$

is indistinguishable to all processes from one of the following executions:

$$\overleftarrow{F}_l = U_0 s_1 U_1 \dots s_{l-1} U_l U_{l-1} s_l \dots s_k U_k$$

in which the update transaction U_l is executed before $U_{l-1} s_l$ instead of after $U_{l-1} s_l$ (forward flip) or

$$\overrightarrow{F}_l = U_0 s_1 U_1 \dots s_{l-1} s_l U_l U_{l-1} \dots s_k U_k$$

in which the update transaction U_{l-1} is executed after $s_l U_l$ instead of before $s_l U_l$ (backward flip).

$$\begin{array}{rcccccccc}
 q & : & s_1 & & \dots & s_{l-1} & & s_l & & \dots & s_k \\
 p_0 & : & U_0 & & & & & U_{l-1} & & & U_k \\
 p_1 & : & & & U_1 & \dots & & & & U_l & \dots
 \end{array}$$

(a) E_k .

$$\begin{array}{rcccccccc}
 q & : & s_1 & & \dots & s_{l-1} & & s_l & & \dots & s_k \\
 p_0 & : & U_0 & & & & & U_{l-1} & & & U_k \\
 p_1 & : & & & U_1 & \dots & \boxed{U_l} & & & & \dots
 \end{array}$$

(b) Forward flip: U_l is performed before $U_{l-1}s_l$.

$$\begin{array}{rcccccccc}
 q & : & s_1 & & \dots & s_{l-1} & s_l & & \dots & s_k \\
 p_0 & : & U_0 & & & & & \boxed{U_{l-1}} & & & U_k \\
 p_1 & : & & & U_1 & \dots & & U_l & & & \dots
 \end{array}$$

(c) Backward flip: U_{l-1} is performed after $s_l U_l$.

Figure 7.1: A flippable execution of length k with two updaters: Figure 7.1(a) shows a flippable execution E_k ; Figure 7.1(b) shows the *forward* flip execution of E_k , where the update transaction U_l by process p_1 is executed before the update transaction U_{l-1} by process p_0 and before the step s_l of the read-only transaction; Figure 7.1(c) shows the *backward* flip execution of E_k , where the update U_{l-1} by process p_0 is deferred after the update transaction U_l by process p_1 and after the step s_l of the read-only transaction.

Figures 7.1(b) and 7.1(c) present the forward and the backward flips of the execution in Figure 7.1(a).

This definition, and the structure of our proof, is similar to the lower bound of Attiya, Ellen and Fatourou [9] on the step complexity of update operations in implementations of atomic snapshot objects. The main difference is that our definition of a flippable execution has *two* types of flipped executions. Additionally, the proof of the lower bound in Section 7.2 uses a flippable execution with t processes executing update transactions instead of just two.

The next lemma proves that if the implementation has a flippable execution then the read-only transaction in this execution does not terminate; it is proved by arguments similar to those applied in [9], extended to handle the possibility of two kinds of flips (forward and backward).

Lemma 7.1 *The read-only transaction in a flippable execution does not terminate successfully.*

Proof: Let $E_k = U_0 s_1 U_1 \dots s_k U_k$ be a flippable execution. Assume, towards a contradiction, that q successfully terminates its read-only transaction in E_k , with a result (v_0, \dots, v_{t-1}) . The proof first fixes the serialization of the update transactions, and

then shows that it is not possible to serial the read-only transaction among the update transactions, using the forward and backward flip executions, which are indistinguishable to q from E_k .

Since the update transactions in the execution E_k do not overlap, they must be serialized in the order U_0, \dots, U_k . Since all steps of the read-only transaction by q are after U_0 and before U_k , it has a unique serialization point between U_{l-1} and U_l , for some l , $1 \leq l \leq k$. Let i_h be the item written by U_{l-1} , and recall that U_{l-1} writes l to i_h ; hence $v_h = l$.

The execution E_k is indistinguishable to process q from F_l , which is either the forward flip

$$\overleftarrow{F}_l = U_0 s_1 U_1 \dots s_{l-1} U_l U_{l-1} s_l s_{l+1} \dots U_k$$

in which update U_l is executed before $U_{l-1} s_l$ instead of after $U_{l-1} s_l$; or the backward flip

$$\overrightarrow{F}_l = U_0 s_1 U_1 \dots s_{l-1} s_l U_l U_{l-1} s_{l+1} \dots U_k$$

in which update U_{l-1} is executed after $s_l U_l$ instead of before $s_l U_l$. Hence, the read-only transaction executed by q in F_l returns the same vector, (v_0, \dots, v_{t-1}) , as in E_k .

Since the update transactions do not overlap in F_l , they are serialized in the order $U_0, \dots, U_l, U_{l-1}, \dots, U_k$, that is, the same as for E_k , except that U_{l-1} and U_l are flipped. Since two consecutive update transactions are to different items, the values of $\{i_0, \dots, i_{t-1}\}$ are the same after both update transactions have been executed, no matter which has been executed first. Hence, at all points in the serialization of F_l , except between U_l and U_{l-1} , the value of all items $\{i_0, \dots, i_{t-1}\}$ is the same as its value in the corresponding points in the serialization of E_k . Thus, the read-only transaction of q can only be serialized after U_l and before U_{l-1} in F_l . However, since U_{l-1} is the first write of l to i_h , the value of i_h is not l before U_{l-1} , and hence, the read-only transaction executed by q cannot be serialized between U_l and U_{l-1} . This contradicts the assumption that the read-only transaction terminates successfully. ■

It remains to prove that a flippable execution exists. Lemma 7.3 (below) shows how to inductively construct a flippable execution, when read-only transactions are invisible. The crux of this lemma is quite different from [9], as it relies on weakly disjoint-access parallelism. A critical step in the proof is provided by Lemma 7.2, showing that in a weakly disjoint-access parallel STM, two consecutive updates by different processes on different items cannot contend on the same base objects. Note that two consecutive update transactions do not contradict weak disjoint-access parallelism since the steps of their executing processes are not interleaved, therefore they do not *concurrently* contend. The proof of the next lemma shows that two such consecutive updates can be perturbed to *concurrently* contend on the same base object.

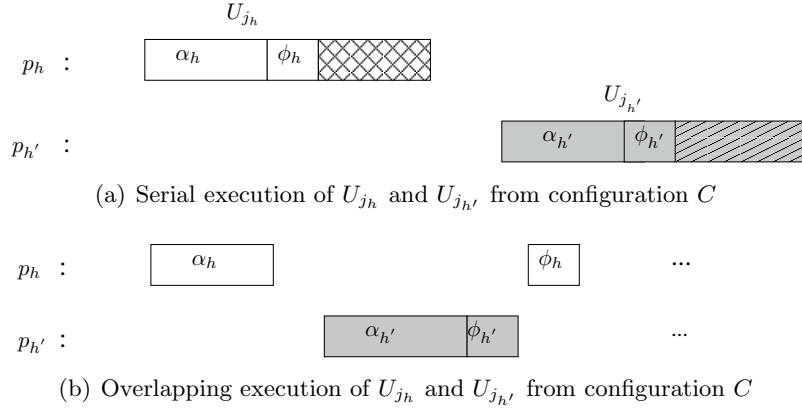


Figure 7.2: Illustration for the proof of Lemma 7.2.

Lemma 7.2 *Given a weakly disjoint-access parallel STM implementation and a quiescent configuration C , consider the consecutive execution of two update transactions $U_{j_h}U_{j_{h'}}$, executed by a process p_h on an item i_h and by process $p_{h'}$ on an item $i_{h'}$, $h \neq h'$, respectively, from C . Then p_h and $p_{h'}$ do not contend on the same base object when executing U_{j_h} and $U_{j_{h'}}$.*

Proof: Assume, towards a contradiction, that p_h and $p_{h'}$ contend on a base object when executing $U_{j_h}U_{j_{h'}}$ from a quiescent configuration C . If in U_{j_h} , p_h applies a non-trivial primitive to a base object on which they contend, let ϕ_h be the last event in U_{j_h} in which p_h applies such a primitive, say, to base object o . Let $\phi_{h'}$ be the first event in $U_{j_{h'}}$ that accesses o .

Otherwise, p_h only applies trivial primitives in U_{j_h} to base objects on which it contends with $p_{h'}$ in $U_{j_{h'}}$; let $\phi_{h'}$ be the first event in $U_{j_{h'}}$ in which $p_{h'}$ applies a non-trivial primitive to some base object, say, o , on which they contend. Let ϕ_h be the last event of p_h in U_{j_h} that accesses o .

In both cases, denote by $\alpha_h\phi_h$ the prefix of the execution of U_h from C and by $\alpha_{h'}\phi_{h'}$ the prefix of the execution of $U_{h'}$ after U_h (see Figure 7.2(a)).

We now create an overlapping execution of the update transactions U_{j_h} and $U_{j_{h'}}$, by processes p_h and $p_{h'}$, from C . We argue that p_h and $p_{h'}$ perform the same steps up to the events ϕ_h and $\phi_{h'}$, and as illustrated in Figure 7.2(b), p_h and $p_{h'}$ concurrently contend on base object o .

In more detail, consider the execution $\alpha_h\alpha_{h'}$ from C , in which p_h executes U_{j_h} until it is about to perform ϕ_h , and then $p_{h'}$ executes $U_{j_{h'}}$ until it is about to perform $\phi_{h'}$. Clearly, p_h is about to perform ϕ_h also after $\alpha_h\alpha_{h'}$. By construction, the execution interval $\alpha_h\alpha_{h'}$ from C is indistinguishable to $p_{h'}$ from the execution interval $U_{j_h}\alpha_{h'}$ from C . Hence, $p_{h'}$ is about to perform the event $\phi_{h'}$ also after $\alpha_h\alpha_{h'}$, that is, $p_{h'}$ and

p_h concurrently contend on o . However, the conflict graph of the execution interval $\alpha_h \alpha_{h'} \phi_{h'} \phi_h$ does not contain a path between the data sets of U_{j_h} and $U_{j_{h'}}$, contradicting the assumption that the implementation is weakly disjoint-access parallel. ■

Since two consecutive updates do not contend on the same base object, we can construct an execution where either the previous update is deferred or the next update is moved forward in the execution without affecting the single step of the read-only transaction in between them. This allows us to inductively construct a flippable execution, in the proof of the next lemma.

Lemma 7.3 *For every $k \geq 0$, every weakly disjoint-access parallel implementation of an STM with invisible read-only transactions, has a flippable execution $E_k = U_0 s_1 U_1 s_2 \dots U_k$ with two updaters p_0 and p_1 , which is indistinguishable to p_0 and p_1 from the execution $E'_k = U_0 U_1 \dots U_k$ in which only p_0 and p_1 take steps.*

Proof: The proof is by induction on the length, k , of the flippable execution E_k executed by a process q and two updaters p_0 and p_1 on two items $\{i_0, i_1\}$. In the base case, $k = 0$, the lemma holds with a solo execution of U_0 , an update transaction by p_0 that writes 1 to i_0 . U_0 successfully terminates since it runs solo from a quiescent configuration.

For the induction step, consider a flippable execution of length $k \geq 1$, $E_k = U_0 s_1 U_1 s_2 \dots U_k$, which is indistinguishable to p_0 and p_1 from the execution $E'_k = U_0 U_1 \dots U_k$. We show how to construct a flippable execution of length $k + 1$, which is indistinguishable from an execution in which only p_0 and p_1 take steps.

By Lemma 7.1, the read-only transaction does not terminate successfully in E_k . Let s_{k+1} be the next step by q . Assume U_k is executed by $p_{h'}$ and let $h = 1 - h'$; note that $h \neq h'$. Let $E_{k+1} = E_k s_{k+1} U_{k+1}$, where process p_h writes $k + 2$ to i_h in the update transaction U_{k+1} . Note that U_{k+1} terminates successfully: The configuration at the end of $E_{k+1} = E_k s_{k+1}$ is indistinguishable from the configuration at the end of E'_k , which is quiescent; since the execution of U_{k+1} from the configuration at the end of E'_k must terminate successfully, by our progress condition, U_{k+1} must also terminate successfully when executing from the configuration at the end of $E_{k+1} = E_k s_{k+1}$.

Since the read-only transaction by q is invisible, $E_{k+1} U_{k+1}$ is indistinguishable to p_0 and p_1 from the execution $E'_k U_{k+1}$.

It remains to prove that E_{k+1} is a flippable execution, i.e., that for every l , $0 < l \leq k + 1$, the execution E_{k+1} is indistinguishable to all processes from either \overleftarrow{F}_l or \overrightarrow{F}_l . For every l , $0 < l \leq k$, by the inductive assumption, the execution

$$E_k = U_0 s_1 U_1 \dots s_{l-1} U_{l-1} s_l U_l \dots s_k U_k$$

is indistinguishable to all processes from the flipped execution F_l which is either

$$\overleftarrow{F}_l = U_0 s_1 U_1 \dots s_{l-1} U_l U_{l-1} s_l \dots U_k$$

or

$$\vec{F}_l = U_0 s_1 U_1 \dots s_{l-1} s_l U_l U_{l-1} \dots s_k U_k.$$

In particular, the configurations at the end of the two executions E_k and F_l are the same. Hence, $E_{k+1} = E_k s_{k+1} U_{k+1}$ and $F_l s_{k+1} U_{k+1}$ are indistinguishable to all processes.

To prove the condition for $l = k + 1$, let C'_{k-1} be the configuration at the end of E'_{k-1} ; C'_{k-1} is quiescent, and Lemma 7.2 implies that $p_{h'}$ and p_h do not contend on the same base object when executing U_k followed by U_{k+1} from C'_{k-1} , namely, in the suffix of E'_{k+1} . By the indistinguishability of E'_{k+1} and E_{k+1} , $p_{h'}$ and p_h do not contend on the same base object while executing U_k and U_{k+1} also in the execution E_{k+1} . Moreover, if q accesses a base object o in s_{k+1} , then either at least one of the two processes p_h or $p_{h'}$ does not access o in U_{k+1} or U_k , respectively, or they both apply a trivial primitive to o . In the former case, if p_h does not access o in U_{k+1} then

$$\overleftarrow{F}_{k+1} = U_0 s_1 U_1 \dots s_k U_{k+1} U_k s_{k+1}$$

is indistinguishable to all processes from E_{k+1} , while if $p_{h'}$ does not access o in U_k , then

$$\vec{F}_{k+1} = U_0 s_1 U_1 \dots s_k s_{k+1} U_{k+1} U_k$$

is indistinguishable to all processes from E_{k+1} . If both p_h and $p_{h'}$ apply a trivial primitive to o , then both flipped executions, \overleftarrow{F}_{k+1} and \vec{F}_{k+1} , are indistinguishable to all processes from E_{k+1} . ■

The impossibility result follows from Lemmas 7.1 and 7.3.

Theorem 7.4 *There is no weakly disjoint-access parallel implementation with invisible read-only transactions of a strictly serializable STM, in which read-only transactions always terminate successfully.*

Proof: Consider a weakly disjoint-access parallel implementation of a strictly serializable STM with invisible read-only transactions. Lemma 7.3 implies that it has a flippable execution E_k , and by Lemmas 7.1, the read-only transaction in E_k does not terminate successfully. ■

The impossibility result stated in Theorem 7.4 holds also for opaque STMs [47], since opacity implies strict serializability.

Our proof shows that the read-only transaction cannot terminate successfully, but it is possible to terminate it unsuccessfully, by *aborting* it. However, when the read-only transaction is retried, it is possible to continue the construction and force it to abort again. Therefore, the proof shows that in disjoint-access parallel implementation some invisible read-only transactions never commit, not even eventually after aborting several times.

Permissive implementations abort a transaction only if necessary for ensuring consistency. For any consistency condition, and any execution, a read-only transaction can always return a consistent values, and the transaction is never required to be aborted to preserve consistency. Therefore, our proof shows that a disjoint-access parallel implementation with invisible read-only transactions that always terminate—however, not always successfully—is not permissive.

7.2 Lower Bound for Read-Only Transactions

The technique of the previous section can be extended to prove that a read-only transaction of t items in a disjoint-access parallel STM implementation, which successfully terminates in a finite number of steps, must apply non-trivial primitives to $t - 1$ base objects; this assumes that there are at least $t + 1$ processes.

The proof of Lemma 7.1—showing that the read-only transaction in a flippable execution cannot terminate successfully—does not rely on the fact that the read-only transaction is invisible, and the lemma continues to hold. On the other hand, we must modify the proof showing the existence of a flippable execution.

This result relies on the notion of (strong) disjoint-access parallelism, which requires two transactions to be connected (in the conflict graph) even if they both just apply a trivial primitive to the same base object. Since we now put a stronger requirement on disjoint-access parallel STM implementations, Lemma 7.2, assuming a weaker requirement, still holds.

We first show (in Lemma 7.5) that, in a disjoint-access parallel STM implementation, two update transactions executed by different processes on different items do not access a common base object when each of them runs solo from a quiescent configuration. This is used in Lemma 7.6 to prove the existence of a flippable execution, when a read-only transaction of t data items applies non-trivial primitives to at most $t - 2$ base objects.

Lemma 7.5 *Given a disjoint-access parallel STM implementation and a quiescent configuration C , consider the execution of an update transaction U_{j_h} to the item i_h by process p_h , and an update transaction $U_{j_{h'}}$ to the item $i_{h'}$ by process $p_{h'}$, $h \neq h'$, from C . Then, p_h and $p_{h'}$ do not access a common base object when executing U_{j_h} and $U_{j_{h'}}$, respectively.*

Proof: Assume, towards a contradiction, that p_h and $p_{h'}$ access the same base object while executing U_{j_h} and $U_{j_{h'}}$, respectively, from C . Let o be the first base object accessed by p_h that is also accessed by $p_{h'}$. Let $\alpha_h\phi_h$ be the prefix of the execution of U_{j_h} from C , where ϕ_h is the first event in which p_h accesses o (see Figure 7.3(a)). Let $\alpha_{h'}\phi_{h'}$ be the prefix of the execution of $U_{j_{h'}}$ from C , where $\phi_{h'}$ is the first access of $p_{h'}$

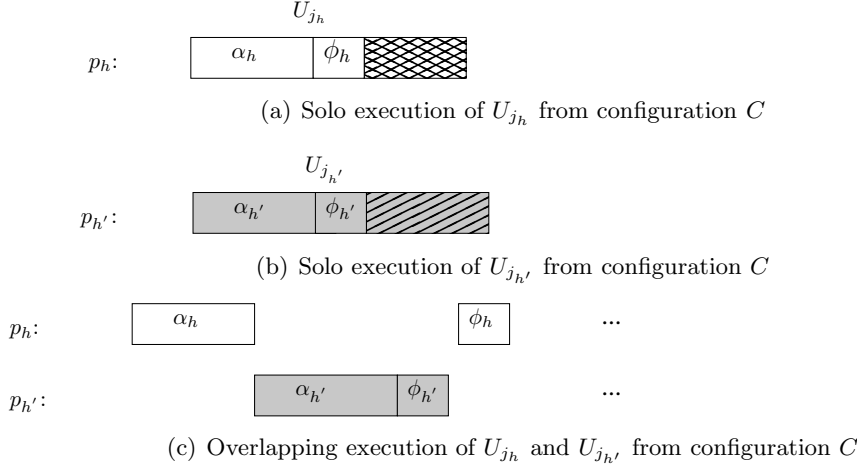


Figure 7.3: Illustration for the proof of Lemma 7.5.

to o (see Figure 7.3(b)). We show how to paste the executions so that the events ϕ_h and $\phi_{h'}$ are concurrently pending.

Consider the execution $\alpha_h\alpha_{h'}$ from C , where p_h executes U_{j_h} until it is about to access o , and then $p_{h'}$ executes $U_{j_{h'}}$ until it is about to access o (see Figure 7.3(c)). By construction, the execution $\alpha_h\alpha_{h'}$ from C is indistinguishable to p_h and $p_{h'}$ from the corresponding executions α_h and $\alpha_{h'}$ from C . Thus, $p_{h'}$ has the event $\phi_{h'}$ pending and p_h has the event ϕ_h pending after $\alpha_h\alpha_{h'}$; that is, $p_{h'}$ and p_h concurrently access o . However, in the conflict graph of the execution interval $\alpha_h\alpha_{h'}\phi_{h'}\phi_h$ from C , there is no path between the data sets of U_{j_h} and $U_{j_{h'}}$, contradicting the assumption that the implementation is disjoint-access parallel. ■

We show that at any point during the execution of the read-only transaction, there is a process that can write to its item without accessing any base object to which q applies non-trivial primitives, thus making the read-only transaction “invisible” to the other processes. Note that, by the definition of a flippable execution, each process always updates the same item. We prove such a process exists by applying a “pigeon hole” argument to show that the process does not access any base object to which the read-only transaction applies non-trivial primitives. Since there are $t - 1$ processes to choose from, each accessing a different item, and since the read-only transaction applies non-trivial primitives to at most $t - 2$ base objects, at least two update transactions by different processes access the same base object, which can be shown to violate disjoint-access parallelism.

Lemma 7.6 *For every $k \geq 0$, a disjoint-access parallel implementation of an STM in which a read-only transaction of $t > 2$ data items applies non-trivial primitives to at*

most $t - 2$ base objects, has a flippable execution $E_k = U_0 s_1 U_1 s_2 \dots U_k$ with t updaters, which is indistinguishable to p_0, \dots, p_{t-1} from the execution $E'_k = U_0 U_1 \dots U_k$ in which only p_0, \dots, p_{t-1} take steps.

Proof: The proof is by induction on the length k of the flippable execution E_k . The base case is when $k = 0$. The lemma holds with a solo execution of an update transaction, U_0 , by process p_0 that writes 1 to i_1 . U_0 successfully terminates since it runs solo from a quiescent configuration.

For the induction step, consider a flippable execution of length k , $E_k = U_0 s_1 U_1 s_2 \dots U_k$, which is indistinguishable to p_0, \dots, p_{t-1} from the execution $E'_k = U_0 U_1 \dots U_k$. We show how to construct a flippable execution of length $k + 1$, which is indistinguishable to p_0, \dots, p_{t-1} from an execution in which only p_0, \dots, p_{t-1} take steps.

By Lemma 7.1, the read-only transaction does not terminate successfully in E_k . Let s_{k+1} be the next step by q and let C_{k+1} denote the configuration at the end of $E_k s_{k+1}$; also, let C'_{k+1} be the configuration at the end of E'_k .

The process p_h to execute U_{k+1} is chosen from p_0, \dots, p_{t-1} such that p_h did not execute U_k and a solo execution of U_{k+1} from C_{k+1} by p_h does not access any base objects to which q applies non-trivial primitives in $E_k s_{k+1}$. Note that this transaction must terminate successfully, by our progress condition; although C_{k+1} is not quiescent, it is indistinguishable from C'_{k+1} , which is quiescent.

We claim such a process exists. Assume, towards a contradiction, that for every process $p_{h_{k+1}}$, $h_{k+1} \neq h_k$, the solo execution by $p_{h_{k+1}}$ from C_{k+1} of the update transaction that writes $k + 2$ to $i_{h_{k+1}}$ accesses a base object to which q applies a non-trivial primitive in $E_k s_{k+1}$. We consider $t - 1$ possible processes, each writing to a different item. Since the read-only transaction applies non-trivial primitives to at most $t - 2$ base objects, at least two update transactions executed by different processes p_h and $p_{h'}$ to different items i_h and $i_{h'}$, starting from configuration C_{k+1} , access the same base object in their first access to a base object to which q applies a non-trivial primitive. Recall that C'_{k+1} is quiescent. Since the execution $E_k s_{k+1}$ is indistinguishable to processes p_h and $p_{h'}$ from the execution E'_k , they access the same base object also when executing the update transactions from C'_{k+1} , which by Lemma 7.5, violates the assumption that the implementation is disjoint-access parallel.

Pick some process $p_{h_{k+1}}$, $h_{k+1} \neq h_k$, that does not access any base objects to which q applies non-trivial primitives in $E_k s_{k+1}$; let U_{k+1} be an update by $p_{h_{k+1}}$ that writes $k + 2$ to $i_{h_{k+1}}$ and denote $E_{k+1} = E_k s_{k+1} U_{k+1}$.

Next, we prove that the execution E_{k+1} is indistinguishable to p_0, \dots, p_{t-1} from the execution E'_{k+1} . This holds for processes other than $p_{h_{k+1}}$ by the inductive assumption and since these processes take no steps in the suffix of this execution. For $p_{h_{k+1}}$, this holds by the inductive assumption and since the solo execution U_{k+1} of an update transaction by $p_{h_{k+1}}$ does not access base objects to which q applies a non-trivial

primitive in $E_k s_{k+1}$.

It remains to prove that for every l , $0 < l \leq k + 1$, the execution E_{k+1} is indistinguishable to all processes from the flipped execution F_l which is either \overleftarrow{F}_l or \overrightarrow{F}_l , as defined in Definition 10. For every l , $0 < l \leq k$, by the inductive assumption, the execution

$$E_k = U_0 s_1 U_1 \dots s_{l-1} U_{l-1} s_l U_l \dots s_k U_k$$

is indistinguishable to all processes from the flipped execution F_l which is either

$$\overleftarrow{F}_l = U_0 s_1 U_1 \dots s_{l-1} U_l U_{l-1} s_l \dots U_k$$

or

$$\overrightarrow{F}_l = U_0 s_1 U_1 \dots s_{l-1} s_l U_l U_{l-1} \dots s_k U_k.$$

In particular, the configurations at the end of the two executions E_k and F_l are the same. Hence, the executions $E_{k+1} = E_k s_{k+1} U_{k+1}$ and $F_l s_{k+1} U_{k+1}$ are indistinguishable to all processes.

For $l = k + 1$, consider the flipped executions \overleftarrow{F}_{k+1} and \overrightarrow{F}_{k+1} . The configuration C'_{k-1} at the end of E'_{k-1} is quiescent. Any STM implementation which is disjoint-access parallel is also weakly disjoint-access parallel, hence we can apply Lemma 7.2 to deduce that p_{h_k} and $p_{h_{k+1}}$ do not contend on, and hence do not access the same base object while executing U_k and U_{k+1} from C'_{k-1} . The indistinguishability property implies that p_{h_k} and $p_{h_{k+1}}$ do not access the same base object while executing U_k and U_{k+1} also in E_{k+1} .

Moreover, if q applies a trivial primitive to some base object o in s_{k+1} , then either at least one of the two processes $p_{h_{k+1}}$ and p_{h_k} does not access o in U_{k+1} and in U_k respectively, or they both apply a trivial primitive to o . In the former case, if $p_{h_{k+1}}$ does not access in U_{k+1} any object that q accesses in s_{k+1} , then

$$\overleftarrow{E}_{k+1} = U_0 s_1 U_1 \dots s_k U_{k+1} U_k s_{k+1}$$

is indistinguishable to all processes from E_{k+1} , while if p_{h_k} does not access in U_k any object that q accesses in s_{k+1} , then

$$\overrightarrow{E}_{k+1} = U_0 s_1 U_1 \dots s_k s_{k+1} U_{k+1} U_k$$

is indistinguishable to all processes from E_{k+1} . If $p_{h_{k+1}}$ and p_{h_k} apply a trivial primitive to o , then both flipped executions are indistinguishable to all processes from E_{k+1} . ■

The lower bound immediately follows:

Theorem 7.7 *In a strict serializable disjoint-access parallel STM implementation for $t + 1$ processes, where all read-only transactions terminate successfully, some read-only*

transaction of $t > 2$ data items applies non-trivial primitives to at least $t - 1$ base objects.

This lower bound holds also for opaque STMs, since opacity implies strict serializability.

7.3 Weaker Consistency Conditions

In this section, we show that both Theorem 7.4 and Theorem 7.7 hold for weaker consistency conditions, namely, snapshot isolation and serializability.

7.3.1 Snapshot Isolation

The weak condition of *snapshot isolation* [21, 69, 83, 100] is known from the database literature, was suggested as an efficient alternative to serializability for STMs [83]; it decouples the consistency of the reads and the writes. In a snapshot isolation implementation, the write sets of any pair of overlapping transactions are disjoint. In addition, for any transaction T , consider a serialization of all the transactions that committed before T started. If T reads item t , then the read operation returns the last value written to t in the serialization, or the initial value if no such write exists. Informally, this means the transaction has a view that is consistent with a snapshot taken when the transaction starts. For a formal definition, see [100, Definition 10.3].

We prove an analogue of Lemma 7.1, that is, we show that the read-only transaction by process q in a flippable execution cannot terminate successfully, also when the implementation provides snapshot isolation.

Lemma 7.8 *Consider a flippable execution of length $k \geq 0$ with t updaters, $E_k = U_0 s_1 U_1 \dots s_k U_k$, of an STM that provides snapshot isolation. The read-only transaction by process q does not terminate successfully.*

Proof: Assume, towards a contradiction, that q successfully terminates its read-only transaction in E_k , with a result (v_0, \dots, v_{t-1}) . Let i_{f_l} be the item written by U_l ; recall that the initial value of all items is zero and that U_l writes $l+1$ to i_{f_l} . By the definition of snapshot isolation, each read operation from an item in the read-only transaction by q returns the most recent committed write operation that updated this item as of the time the read-only transaction starts. The read-only transaction by q returns the most recent values after U_0 is executed and before any other update is executed. Hence, $v_{f_0} = 1$, and for every l , $1 \leq l \leq t-1$, $v_{f_l} = 0$.

The execution E_k is indistinguishable to process q from F_1 , which is either the forward flip

$$\overleftarrow{F}_1 = U_1 U_0 s_1 s_2 U_2 \dots U_k$$

or the backward flip

$$\vec{F}_1 = s_1 U_1 U_0 s_2 U_2 \dots U_k .$$

Hence, the read-only transaction executed by q in F_1 returns the same vector as in E_1 .

However, the definition of snapshot isolation requires different results for the read-only transaction in these executions. In \overleftarrow{F}_1 , the read-only transaction returns the most recent values after U_0 and U_1 are executed and before any other update is executed, hence it returns a vector where $v_{f_0} = 1$, $v_{f_1} = 2$, and $v_{f_l} = 0$, for every l , $2 \leq l \leq t - 1$. In \vec{F}_1 , the read-only transaction returns the most recent values before any update is executed, hence it returns a vector where $v_{f_l} = 0$, for every l , $0 \leq l \leq t - 1$. Thus, the read-only transaction cannot terminate successfully. ■

The proofs of Lemma 7.2 and Lemma 7.5—showing that two update transactions executed by different processes on different items do not access a common base object when one follows the other or when each of them runs solo from a quiescent configuration, respectively—do not rely on the safety property, and thus the lemmas continue to hold for STMs that provide snapshot isolation. The proofs of Lemma 7.3 and Lemma 7.6—showing the existence of the flippable execution—also do not rely on the safety property, and they can be adapted in a straightforward manner to use Lemma 7.8 instead of Lemma 7.1.

Hence, the impossibility result follows from Lemmas 7.3 and 7.8.

Theorem 7.9 *There is no weakly disjoint-access parallel STM implementation with invisible read-only transactions of an STM providing snapshot isolation, in which read-only transactions always terminate successfully.*

The lower bound follows from Lemmas 7.6 and 7.8.

Theorem 7.10 *In a disjoint-access parallel STM implementation for $t + 1$ processes providing snapshot isolation, where all read-only transactions terminate successfully, some read-only transaction of $t > 2$ data items applies non-trivial primitives to at least $t - 1$ base objects.*

7.3.2 Serializability

Recall that an STM is *serializable* if transactions appear to execute sequentially, one after the other; note that we require that transactions of the same process preserve their order (*per-process* order) and that repeatedly reading the same data item eventually returns a non-initial value.

The proof uses an additional process q' . Given a flippable execution $E_k = U_0 s_1 U_1 \dots s_k U_k$, we construct an *augmented flippable execution*

$$\widehat{E}_k = U_0 s_1 S_1^* U_1 \dots s_k S_k^* U_k ,$$

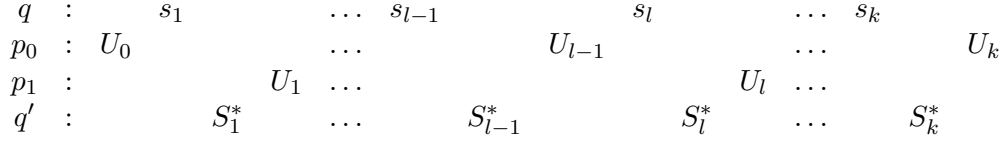


Figure 7.4: An augmented flippable execution \widehat{E}_k derived from the flippable execution E_k of Figure 7.1.

where the additional process q' performs invisible read-only transactions. For every $j \in \{1, \dots, k\}$, q' performs solo a sequence S_j^* of read-only transactions after the event s_j by process q and before the update U_j . Each read-only transaction in S_j^* accesses the items $i_{f_{j-1}}$ and i_{f_j} updated by U_{j-1} and U_j . The result of the last read-only transaction in the sequence S_j^* , denoted S_j , is the value written by U_{j-1} to $i_{f_{j-1}}$ and the last value of i_{f_j} before U_j updates it.

Figure 7.4 shows the augmented flippable execution obtained by augmenting the flippable execution E_k of Figure 7.1 with sequences of read-only transactions performed by process q' .

We rely on the per-process ordering of transactions to prove that the read-only transactions of q' must eventually read the latest value written in U_{j-1} , and thus, S_j^* is finite.

Lemma 7.11 *Consider an augmented flippable execution of length $k \geq 0$, $\widehat{E}_k = U_0 s_1 S_1^* U_1 \dots s_k S_k^* U_k$. In any serialization of \widehat{E}_k that preserves the per-process order, U_0, U_1, \dots, U_k appear in their order of execution.*

Proof: We show, by induction on ℓ , that U_0, U_1, \dots, U_ℓ appear in their order of execution. In the base case, $k = 0$, the serialization of U_0 is trivial.

For the induction step, consider $U_{\ell+1}$. By the induction assumption, the updates U_0, U_1, \dots, U_ℓ are serialized by their execution order in \widehat{E}_k . By construction, $S_{\ell+1}^*$ is a sequence of read-only transactions that access i_{f_ℓ} and $i_{f_{\ell+1}}$, and the last read-only transaction in $S_{\ell+1}^*$, denoted $S_{\ell+1}$, returns the value written by U_ℓ and the last value of $i_{f_{\ell+1}}$ before the one written by $U_{\ell+1}$.

The sequence $S_{\ell+1}^*$ is finite since the STM is serializable and so, eventually, some transaction must return the latest values written to i_{f_ℓ} and $i_{f_{\ell+1}}$, and by the induction assumption, U_ℓ is the last to write to i_{f_ℓ} . Moreover, $S_{\ell+1}$ completes before $U_{\ell+1}$ starts, so it cannot return the value written by $U_{\ell+1}$, since due to serializability, a read operation can not return a value not written.

Since each data item is written by a different process, and due to per-process order, $U_{\ell+1}$ can not be serialized before the last update of $i_{f_{\ell+1}}$ preceding $U_{\ell+1}$.

Moreover, $U_{\ell+1}$ can not be serialized after this update and before $S_{\ell+1}$, since $S_{\ell+1}$ does not return the value written by $U_{\ell+1}$. Hence, $U_{\ell+1}$ is serialized after $S_{\ell+1}$. \blacksquare

We use Lemma 7.11 to prove an analogue of Lemma 7.1.

Lemma 7.12 *Consider an augmented flippable execution of length $k \geq 0$ with t updaters, $\widehat{E}_k = U_0 s_1 S_1^* U_1 \dots s_k S_k^* U_k$. If the read-only transactions by process q' are invisible, then the read-only transaction by process q does not terminate successfully.*

Proof: Assume, towards a contradiction, that the read-only transaction of process q in \widehat{E}_k terminates successfully and returns a value (v_0, \dots, v_{t-1}) , which does not violate serializability. Let the augmented flippable execution $\widehat{E}_k = U_0 s_1 S_1^* U_1 \dots s_k S_k^* U_k$ correspond to a flippable execution $E_k = U_0 s_1 U_1 \dots s_k U_k$.

By Lemma 7.11, the updates in \widehat{E}_k are serialized in the order U_0, U_1, \dots, U_k . The vector (v_0, \dots, v_{t-1}) determines where q 's read-only transaction is serialized. In particular, for some l , $0 < l \leq k$, the read-only transaction of q is serialized after U_{l-1} and before U_l , and for each item i_f in $\{i_0 \dots i_{t-1}\}$, either v_f is zero and no update wrote to i_f before U_l , or the last update to i_f before U_l wrote v_f to i_f . Let S be the serialization of execution \widehat{E}_k .

Since the read-only transactions executed by process q' are invisible, \widehat{E}_k and E_k are indistinguishable to p_0, \dots, p_{t-1} and q . Thus, they will execute the same steps in both executions. Note that S is a serialization also for E_k . Since S preserves the real-time order among transactions, E_k is a flippable execution where the read-only transaction terminates and strict serializability is preserved, contradicting Lemma 7.1. ■

As discussed before the lemma, the existence of a flippable execution (guaranteed by Lemma 7.3) implies there is an augmented flippable execution, and hence, Lemma 7.12 implies the following impossibility result:

Theorem 7.13 *There is no weakly disjoint-access parallel STM implementation with invisible read-only transactions of a serializable STM, in which read-only transactions always terminate successfully.*

When a read-only transaction of $t \geq 2$ data items applies non-trivial primitives to at most $t - 2$ base objects, the read-only transactions of q' in the augmented flippable execution are, in fact, invisible since their read set contains only two data items. As discussed before Lemma 7.12, the existence of a flippable execution (guaranteed by Lemma 7.6) implies there is an augmented flippable execution, and hence, Lemma 7.12 implies the following lower bound:

Theorem 7.14 *In a serializable disjoint-access parallel STM implementation for $t + 2$ processes, where all read-only transactions terminate successfully, some read-only transaction of $t > 2$ data items applies non-trivial primitives to at least $t - 1$ base objects.*

The impossibility result and lower bound also hold for *virtual world consistency*, proposed by Imbs et al. [60]. This consistency condition requires serializability or strict serializability of committed transactions, and ensures that aborted transactions always see a consistent state of the memory, although not necessarily consistent with each other. Since our results do not consider the behavior of aborted transactions, they also hold for virtual world consistency.

Chapter 8

Limitations on STMs with Nontransactional Accesses

In many situations, it is important for STMs to be able to run transactions together with *nontransactional* code accessing the same memory locations. *Strong atomicity* allows to combine nontransactional with transactional accesses even when nontransactional operations (as opposed to transactional operations) are *uninstrumented*. Uninstrumented nontransactional read operations simply read a fixed memory location. The location might depend on the process and the item, e.g., a local copy of the item, but the process applies no manipulation on the value and simply returns the value written in the memory as the value of the item. Uninstrumented nontransactional write operations can be defined analogously (although they are not used in our proofs).

Strong atomicity can be provided by supporting *privatization*. Privatization means a process isolates some shared data, possibly by removing all references to the data, after which the process can privately, nontransactionally, access the data without interference by other processes. Rather than formally defining privatization, we only state a property that is naturally expected out of any notion of privatization.

Process p_j *privatizes* item t_i when p_j commits a transaction privatizing t_i . The private base object that process p_j associates with a data item t_i , after privatizing the item t_i , is denoted m_i^j .

Property 1 (Privatization-safe STM) *An STM with uninstrumented nontransactional operations is privatization safe if after process p_j privatizes item t_i , no process $p_h \neq p_j$ applies a nontrivial primitive to the base object m_i^j .*

The main objective of privatization is to improve performance. Consider, for example, the linked list depicted in Figure 8.1, in which every node points to a root item. Every root item points to some disjoint subgraph, such as a tree, and is the

⁰The results of this chapter have appeared in [13].

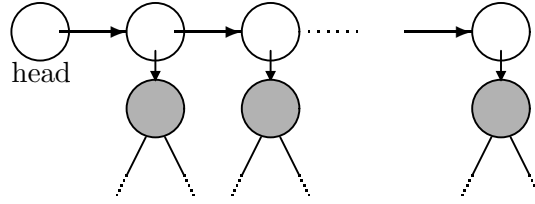


Figure 8.1: In this example, the privatizing transaction sets the head to NULL and privatizes the dark items and their subgraphs. Other transactions write to the dark items; each transaction writes to a different one.

only path to items in the subgraph. When a process privatizes all root items and their subgraphs, it can afterwards access all items in the rooted trees with simple (uninstrumented) reads and writes to the shared memory, which avoids the overhead of transactions and thereby dramatically increasing the efficiency. The hope is that a constant amount of work, e.g., nullifying the head of the linked list, will suffice for privatizing the whole linked list.

In addition to a transaction privatizing all root items, consider a *workload* in which other *updating* transactions read all the nodes of the linked list but write only to one root item that is pointed from the list. Previous work [67] informally assumed that the privatizing transaction must conflict with the other transaction accessing the privatized region. For example, if the privatizing transaction writes to the head of the linked list, and all updating transactions read the head, then the privatizing transaction has a write-read conflict on the head with any updating transaction.

Indeed, it can be easily shown that a weakly progressive STM cannot support privatization *if the read set of every writing transaction is empty*, unless the privatizing transaction accesses all the items it privatizes. Otherwise, if there is an item the privatizing transaction does not access, then a transaction writing to this item executed after the privatizing transaction completes, is unaware of the privatization and may access private locations.

To increase the efficiency, it is desirable that transactions do not observe operations of other non-conflicting transactions, whether or not these operations leave a mark on the memory. This is captured by the following notion of obliviousness (Definition 11).

An ℓ -independent execution contains $\ell \geq 0$ transactions, each executed by a different process, $p_{i_1}, \dots, p_{i_\ell}$, running solo until it is logically committed, on data sets without nontrivial conflicts. An STM is *oblivious* if a transaction running solo after an independent execution, without nontrivial conflicts with the pending transactions, behaves in a manner that is independent of the data sets of the pending transactions.

Definition 11 (Oblivious STM) *An STM is oblivious if for any pair of ℓ -independent executions α_1 and α_2 , each containing ℓ transactions executed by the same processes $p_{i_1}, \dots, p_{i_\ell}$, in the same order, if some transaction T executed by a process p does not*

have nontrivial conflicts with the transactions in α_1 and α_2 , then $\alpha_1 T$ and $\alpha_2 T$ are indistinguishable to p .

Our first main result assumes, in addition to obliviousness, invisible reads. In the linked-list workload example this means that, for every process, the execution of other transactions appears only to write to a single item (either the head of the list or an item pointed by the links). In this case, we show an inherent cost for supporting privatization: a transaction privatizing k items must have a data set of size at least k .

Our second main result removes the assumption of invisible reads, and shows that $\Omega(k)$ memory locations must be accessed by a privatizing transaction, where k is the minimum between the number of privatized items and the number of concurrent transactions guaranteed to make progress, thus capturing the tradeoff between the cost of privatization and the parallelism offered by the STM.

The rest of the chapter is organized as follows: we show in Section 8.1 that *eager* STMs, in which a transaction may update the memory before it is guaranteed to commit cannot support privatization, under the weak 1-progressive assumption. Section 8.2 discusses STMs satisfying the *parameterized opacity* [44] condition that allow uninstrumented operations on items without prior privatization. Section 8.3 includes the lower bound on the size of the data set of privatizing transactions in STMs with invisible reads. Section 8.4, bounds the cost of privatization with visible reads, and sketches an STM that matches this lower bound. Finally, Section 8.6 presents the results for disjoint-access parallel STMs.

8.1 Eager STMs

We first show that in a privatization-safe STM, a transaction applies a nontrivial primitive (e.g., a write) to a base object associated with a privatized item, only after it is already logically committed.

An STM is *eager* if there is a configuration C such that a transaction T is the only pending transaction in C , T is not logically committed, and a process p executing T applies a nontrivial primitive to a base object associated with an item that another process privatizes. It is simple to show that a 1-progressive eager STM is not privatization-safe. This is claimed to be a known flaw in eager STMs that are not strongly atomic [37], but was never proved formally. Theorem 8.1 emphasizes the assumptions required for proving this claim. Note that assuming uninstrumented nontransactional operations implies that the mapping of each privatized item to the corresponding base object is independent of the execution, and it is known in advance.

Theorem 8.1 *An uninstrumented 1-progressive eager STM is not privatization-safe.*

Proof: Since the STM is uninstrumented, let m_i be the private base object which process p_0 associates with item t_i . Assume another process p_1 executing a transaction T_1 applies a nontrivial primitive to m_i before T_1 is logically committed in some configuration C , where only T_1 is pending; call this event τ . Consider a transaction T_0 of p_0 privatizing the item t_i , executed from C .

Since only T_1 is pending in C , α , the execution leading to C , has no incomplete logically committed transaction conflicting with T_0 . Since the STM is 1-progressive, T_0 completes successfully when executed after α , and since the STM is uninstrumented, m_i is private to p_0 after αT_0 . However, τ can be applied to m_i , even after T_0 , in contradiction to privatization safety. ■

In the sequel, we assume that the implementations are uninstrumented and (at least) 1-progressive, therefore a privatization-safe STM is assumed not to be eager.

8.2 Uninstrumented Access without Prior Privatization

Parameterized opacity [44] is a framework for describing the interaction between transactions and nontransactional operations, extending *opacity* [47], and parameterized by a memory model for the semantics of nontransactional operations. Roughly, every transaction appears as if it is executed instantaneously with respect to other transactions and nontransactional operations, and nontransactional operations obey the underlying memory model.

Guerraoui et al. prove that for memory models that restrict the order of some pair of read or write operations to different variables, parameterized opacity cannot be achieved [44, Theorem 1]. Furthermore, they prove that for memory models that allow reordering all operations to different variables, parameterized opacity requires either instrumenting nontransactional operations or using RMW primitives when writing inside a transaction. They also present an uninstrumented STM that guarantees parameterized opacity with respect to memory models that do not restrict the order of any pair of read or write operations. This STM uses a global lock, and is not weakly progressive. Their results assume that items are accessed nontransactionally, without a preceding privatization transaction. In a sense, their results show the implications of not privatizing, while our results complete the picture by showing the cost of privatization.

We show that a 1-progressive, oblivious uninstrumented STM, allowing more than one transaction to proceed concurrently, cannot achieve opacity parameterized with respect to any memory model.

Theorem 8.2 *There is no uninstrumented, 1-progressive, oblivious STM that guarantees opacity with respect to any memory model.*

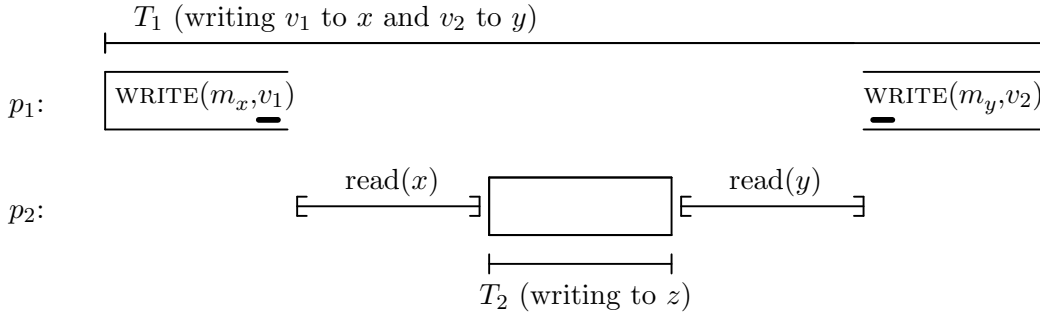


Figure 8.2: The execution used in the proof of Theorem 8.2. The process p_1 executes the transaction T_1 , steps on base objects are denoted by bold lines. The process p_2 invokes nontransactional operations, denoted by square brackets intervals, and the transactions T_2 .

The proof follows the lines of the proof of Theorem 1 in [44], specifically, the part of the proof that handles memory models that restrict the order of pairs of read operations. Their proof constructs an execution, in which one process executes a transaction and another process issues nontransactional read operations, such that there is no serialization of the transaction and nontransactional operation which respects the restriction posed by the memory model. The memory model in our proof may not pose any restriction on the order of the nontransactional read operations, but we can fix their order by inserting an additional transaction between them.

Proof: Consider an execution of a transaction T_1 by p_1 , writing v_1 to item x and v_2 to item y . Denote by m_x , m_y the base objects that p_2 accesses when it reads nontransactionally from x and y respectively.

During the execution of T_1 , p_1 writes v_1 to m_x , and v_2 to m_y . Otherwise, since the STM is uninstrumented, a nontransactional read of p_2 from x (respectively, y) after T_1 is completed, returns the initial value of m_x (respectively, m_y) instead of v_1 (respectively, v_2), so the STM is not parameterized opaque, and we are done.

Without loss of generality, assume p_1 writes v_1 to m_x before writing v_2 to m_y . Denote by C_1 the configuration after p_1 first writes v_1 to m_x . Consider the solo execution of p_2 from C_1 in which it reads x nontransactionally, executes a transaction T_2 in which it writes to item z , and then reads y nontransactionally. T_2 is committed since the STM is 1-progressive. Denote by C_2 the configuration after the solo execution of p_2 , and by α the execution of p_1 and p_2 preceding C_2 . Process p_1 completes T_1 from C_2 (see Figure 8.2). The nontransactional read of x by p_2 returns v_1 , and parameterized opacity implies that T_1 is logically committed at C_1 . Since the STM is 1-progressive, T_1 commits when executed from C_2 .

Claim 8.3 *Process p_2 does not modify the state of m_y in α .*

Proof: Consider an execution α' , such that p_1 executes solo a transaction T_1' writing to item w until the transaction is logically committed at configuration C_1' . Then, p_2 runs solo reading x nontransactionally, executing T_2 , and reading y nontransactionally. Since the STM is uninstrumented p_2 does not modify the state of m_y when reading nontransactionally x and y .

If p_1 or p_2 modify the state of m_y while executing T_1' and T_2 in α' then the nontransactional read operation to y of p_2 after T_2 returns a value that is not the initial value of m_y , whereas no committed transaction writes to y in α' , contradicting parameterized opacity.

By 1-obliviousness, the executions of T_2 from C_1 and C_1' are indistinguishable. Process p_1 does not access m_y neither in α nor in α' , thus p_2 does not apply successfully a nontrivial primitive to m_y also in α . ■

Parameterized opacity implies that T_1 appears before the nontransactional read to x in the ordering, and that T_2 and the nontransactional read to y are ordered after the nontransactional read to x , and thus after T_1 . However, the nontransactional read to y is issued before p_1 first writes to m_y and by Claim 8.3 also p_2 does not modify the state of m_y . Since the STM is uninstrumented, the nontransactional read to y returns the initial value of m_y and not v_2 , thus the STM is not parameterized opaque. ■

Together with [44, Theorem 1], this means that progressive, oblivious, uninstrumented STMs cannot achieve opacity parameterized with respect to *any* memory model, and indicates that a privatizing transaction must precede nontransactional accesses to data items, unless parallelism is compromised.

8.3 Privatization with Invisible Reads

The next theorem shows that in an oblivious STMs supporting privatization, the data set of a privatizing transaction must contain all privatized items. The proof proceeds by creating a scenario in which a privatizing transaction misses the up-to-date value of a privatized item; some care is needed in order to argue about each item separately.

Theorem 8.4 *For any privatization-safe STM that is 1-progressive, oblivious and with invisible reads, there is a workload including a transaction T_0 , privatizing k items, in which transactions have nonempty read sets, for which there is an execution where the size of the data set of T_0 is $\Omega(k)$.*

Proof: Consider two processes p_0 and p_1 : p_0 executes a transaction T_0 that privatizes the items t_1, \dots, t_k . For p_1 , consider a transaction T_1' with an arbitrary read set, writing to an item u that is never accessed by T_0 .

Consider the execution $\alpha' = I_1' T_0$, such that in I_1' , p_1 executes a prefix of the transaction T_1' until it is logically committed (see Figure 8.3(a)). I_1' is indistinguishable

to p_0 from an execution in which p_1 executes a transaction that only writes to u until it is logically committed. After I'_1 , there is no incomplete transaction that has a conflict with T_0 , and since the STM is 1-progressive, T_0 commits when executed after I'_1 .

Assume, by way of contradiction, that the data set of T_0 does not include some item t_i that it privatizes when executed after I'_1 . Consider the execution $\alpha = I_1 T_0$, such that in I_1 , p_1 executes a prefix of a transaction T_1 with the same read set as T'_1 and writing to the item t_i a value different than its initial value, and T_1 is logically committed after I_1 (see Figure 8.3(b)). We show that t_i is not in the data set of T_0 also when executed after I_1 .

Claim 8.5 T_0 does not access t_i also when executed after I_1 .

Proof: The proof uses the following definitions: A transaction includes *access operations*, each operation is a triple: the item which the operation accesses, an indication whether the operation is a read or a write, and the value written or read by the operation. In addition, there are three special operations: the *start*, *commit* and *abort* operations. Every new transaction begins with the start operation, then followed by a sequence of read and write operations. The last operation of a transaction is either an access operation, in which case the transaction is pending, or a commit or abort operation, in which case the transaction is *committed* or *aborted*, respectively.

At any point during the execution of a transaction, the *view* of the transaction is a list of item-value pairs, which includes all the values read by the transaction to this point.

The claim is trivial if that data set is static, so assume that the data set of the transaction is dynamic, i.e., deterministically determined by the view of the transaction before every operation. We prove by induction that the transaction has the same view after applying the same number of operations when executed after I_1 and I'_1 , and that T_0 logically commit when running after I_1 .

The base case is before applying the first operation: the view of the transaction is empty in both cases, and T_0 is neither aborted nor blocked.

For the induction step, assume that after applying $i - 1$ operations when running after I_1 , T_0 has the same view as after applying $i - 1$ operations when running after I'_1 , and that T_0 is neither aborted nor blocked. The i -th operation T_0 applies when running after I_1 , is the same as the i -th operation T_0 applies when running after I'_1 . Recall that T_0 does not access u in any execution. Since T_0 does not access t_i when executed after I'_1 , the i -th operation T_0 applies when running after I_1 also does not access t_i . T_0 does not abort nor blocks when applying the i -th operation after I_1 , and the view after the i -th operation is the same view as after applying the i -th operation when running after I'_1 . ■

Since reads are invisible, I_1 is indistinguishable to p_0 from an execution without

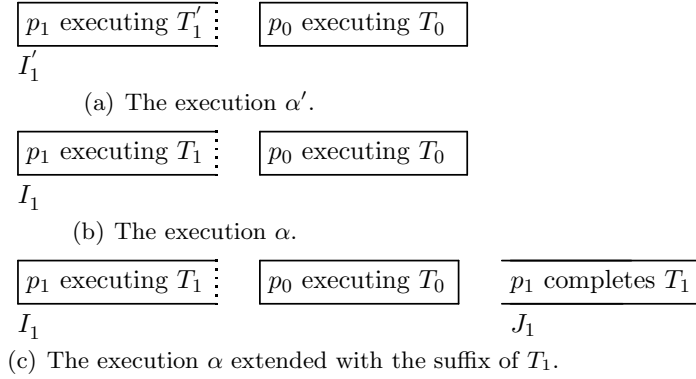


Figure 8.3: The executions used in the proof of Theorem 8.4. A dotted line indicates that the transaction is logically committed.

nontrivial conflicts, and since the STM is oblivious, T_0 commits also when executed after I_1 in α . Let m_1, \dots, m_k be the base objects that are private to p_0 after α (we omit the superscript 0). Since the STM is 1-progressive, T_1 commits when completed after α . Since T_1 is logically committed after I_1 , it writes to t_i . Consider the execution $I_1 T_0 J_1$, such that J_1 is the suffix of the execution of T_1 until it commits (see Figure 8.3(c)).

Claim 8.6 p_1 modifies the state of m_i in J_1 .

Proof: We first show that p_1 does not modify m_i in the first part of its transaction in α . Assume that p_1 applies a nontrivial primitive to m_i in some step when executing I_1 in α , and let τ be the first such step. Let \widehat{I}_1 be the prefix of I_1 preceding τ . I_1 is the shortest prefix of T_1 after which T_1 is logically committed. Hence, T_1 is not logically committed after \widehat{I}_1 , and it is the only transaction that is pending after \widehat{I}_1 . In a solo execution of T_0 after \widehat{I}_1 , T_0 is committed, making m_i private to p_0 . Since the STM is not eager, p_1 does not apply τ after \widehat{I}_1 .

A similar argument shows that p_1 does not apply nontrivial primitive to m_i in α' .

Next, we argue that p_0 does not modify the state of m_i in α . Otherwise, a non-transactional, uninstrumented read operation, to t_i of p_0 after α' returns a value that is not the initial value of m_i , whereas no committed transaction writes to t_i in α' , contradicting our correctness condition. The executions of T_0 after I_1 and after I_1' are indistinguishable, since the STM is oblivious and since T_0 does not access neither u nor t_i . We have shown that p_1 accesses m_i neither in α nor in α' , and hence, p_0 does not successfully apply a nontrivial primitive to m_i also in α .

If p_1 does not modify the state of m_i also in J_1 , then a nontransactional read by p_0 to t_i after $I_1 T_0 J_1$ returns the initial value of m_i , since reads are uninstrumented. This contradicts our correctness condition since T_1 , which writes to t_i a value that is

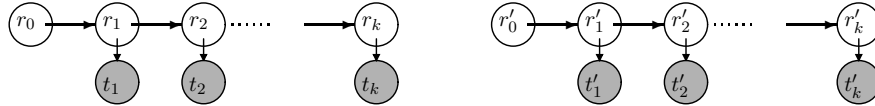


Figure 8.4: An example workload for the proof of Theorem 8.7. The privatizing transaction privatizes the left linked list, while other transactions traverse either the left linked list or the right linked list, and write to respective items.

different from the initial value of m_i , commits before the nontransactional read of p_0 .

■

Therefore, p_1 applies a nontrivial primitive to m_i in J_1 after the execution of T_0 , in contradiction to privatization safety. ■

The proof allows T_1 and T'_1 to have non-empty read sets. Since reads are invisible, this looks to p_0 as if they have empty read sets. Note however, that p_1 does read from the memory and distinguishes its execution of T_1 and T'_1 from executing a transaction with an empty read set. Thus, the result does not follow from the trivial lower bound for transactions with empty read sets.

8.4 Privatization with Visible Reads

A similar lower bound holds for STMs with visible reads, assuming they ensure some degree of parallelism. The cost is stated in terms of low-level accesses by the privatizing transaction, rather than in terms of the high-level aspects of the transaction. Some key ideas in the proof are similar to the proof of Theorem 8.4; however, the technical details are more involved, in order to accommodate visible reads. Therefore, before getting into it, we start with a high-level outline for a workload similar to the linked list of Figure 8.1.

Consider the scenario described in the left side of Figure 8.4: Let r_0, r_1, \dots, r_k be the nodes of the linked list (r_0 is the head node); each node r_i , $1 \leq i \leq k$ points to an item t_i .

We have k *updating* transactions traverse the nodes of a linked list (r_0, r_1, \dots, r_k) , while each transaction writes to a different item pointed by the list (i.e., the i -th transaction writes to t_i); each transaction writes to an item that is not read by other transactions, so these transactions have only trivial conflicts. Later, a transaction by another process, privatizing all items pointed by the linked list (t_1, \dots, t_k) , is shown to miss the up-to-date value of the privatized items, unless it accesses many base objects.

Since reads are visible, however, it is difficult to hide the updating transaction from the privatizing transaction. The challenge is to create an execution in which an updating transaction runs long enough to guarantee that it will commit—even after

the privatizing transaction commits, and even if the privatizing transaction writes to an item it reads—but not long enough to become visible to the privatizing transaction.

The privatizing transaction may write to an item in the read set of an updating transaction (e.g., the head of the list), thus invalidating its read set. Hence, to guarantee that an updating transaction eventually commits in the execution constructed, the updating transaction runs until it is logically committed, before the privatizing transaction starts.

It may seem that, at this point, the privatizing transaction does not need to access many objects to observe a conflict with the updating transactions, and it can abort or at least block until the conflicts are resolved. However, the obliviousness and non-eagerness of the STM can be used to “hide” the updating transactions from the privatizing transaction.

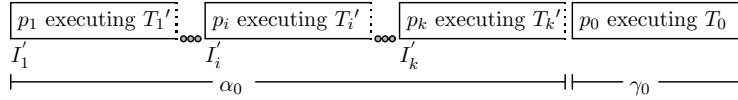
For this, we create a copy of the linked list, having exactly the same structure, which is not connected to the first list in any way. It contains $k + 1$ nodes r'_0, r'_1, \dots, r'_k and k items t'_1, \dots, t'_k . The privatizing transaction does not access this list at all; however, *confusing* transactions, also having only trivial conflicts among them, access this completely disjoint linked list. Due to obliviousness, these transactions are indistinguishable from—and therefore can be swapped with—the original updating transactions.

We start with an execution in which the confusing transactions run one after the other; this execution is k -independent. Then, we swap confusing transactions with updating transactions. Swapping is done inductively: Each inductive step swaps one confusing transaction with an updating transaction by the same process; that is, at each step one additional process executes the updating transaction instead of the confusing transaction, and *incurs an access to at least one additional base object* by the privatizing transaction. This yields an execution in which the privatizing transaction accesses many objects, implying the lower bound.

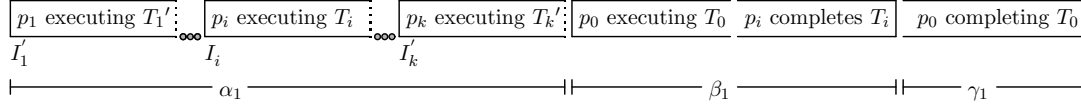
Progressiveness is used to ensure that if at some point the privatizing transaction observes a conflict, the updating transaction causing the conflict may run to completion. This also ensures that the privatizing transaction runs to completion.

A key technical challenge in the proof is in deciding which transaction to swap next, so as not to lose the accesses by the privatizing transaction that appear in the execution we have created so far. Specifically, we need to pick a transaction T such that swapping it is invisible to the privatizing transaction in its execution prefix, at least during the memory accesses incurred due to previous swaps. This is done by choosing T to be the last transaction to modify the next location seen by the privatizing transaction, so that future swaps will not overwrite locations T writes to and that are accessed by the privatizing transaction in its execution prefix. (This is the purpose of Item 5 maintained in the inductive construction.)

The lower bound, which we prove next, holds for any workload satisfying the condition stated in the theorem, including the linked list workload discussed above.



(a) The execution $\alpha_0\beta_0\gamma_0$: α_0 is $I_1' \dots I_k'$; β_0 is the empty execution interval; and γ_0 is a solo execution of T_0 .



(b) The execution $\alpha_1\beta_1\gamma_1$: p_i executes I_i instead of I_i' in α_1 ; in β_1 , p_0 starts executing T_0 and p_i completes T_i ; p_0 completes T_0 in γ_1 .

Figure 8.5: The executions used in the proof of Theorem 8.7. A dotted line indicates that the transaction is logically committed.

Theorem 8.7 *For any privatization-safe STM that is l -progressive and oblivious, and every workload including a transaction T_0 privatizing items t_1, \dots, t_m and m updating transactions T_1, \dots, T_m , such that transaction T_i writes to t_i and does not read from any item t_j , $j \neq i$ (it may read other items), there is an execution where T_0 accesses $\Omega(k)$ base objects, where $k = \min\{l, m\}$.*

Proof: Consider $k + 1$, $k \geq 1$, processes p_0, \dots, p_k , such that p_0 executes T_0 , and for every i , $1 \leq i \leq k$, p_i executes T_i . Create shadow copies of all items in the data sets of T_1, \dots, T_m ; T_0, \dots, T_m do not access any of these shadow items. We denote by t'_1, \dots, t'_k the shadow copies of t_1, \dots, t_k . For every process p_i , $1 \leq i \leq k$, consider an additional (confusing) transaction, T_i' , reading the shadow copies of the items in the read set of T_i , and writing to the item t'_i , which is the shadow copy of t_i . Note that for every i , T_i' does not read any item in $\{t_1, \dots, t_k\} \cup \{t'_1, \dots, t'_k\} \setminus \{t'_i\}$; this ensures that the data sets of the transactions $\mathcal{T}_j = \{T_i\}_{1 \dots k} \cup \{T_i'\}_{1 \dots k} \setminus \{T_j\}$ and $\mathcal{T}'_j = \mathcal{T}_j \cup \{T_j\} \setminus \{T_j'\}$ are nonconflicting, for every j , $1 \leq j \leq k$. Furthermore, T_i' does not read any item written by T_0 ; this ensures that T_i' and T_0 are nonconflicting, for every i .

A process p reads from a process q via a base object o in an execution α if p accesses o , and o was last modified by q . Process p reads from a set of processes P in an execution α if for every process $q \in P$, there is a base object o such that p reads from q via o in α .

Consider the following execution $\alpha_0\beta_0\gamma_0$: α_0 is $I_1' \dots I_k'$, such that p_i executes in I_i' , a prefix of the transaction T_i' and T_i' is logically committed after I_i' ; β_0 is the empty execution interval; and γ_0 is a solo execution of T_0 by p_0 to completion (see Figure 8.5(a)).

For every ℓ , $0 < \ell \leq k$, we show how to perturb $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$ to obtain an execution $\alpha_\ell\beta_\ell\gamma_\ell$, such that

1. p_0 executes T_0 to successful completion in $\beta_\ell\gamma_\ell$.
2. p_0 reads from all processes in P_ℓ , a subset of $\{p_1, \dots, p_k\}$ of size (at least) ℓ , in $\alpha_\ell\beta_\ell$.
3. There is a subset Q_ℓ of P_ℓ , where every process $p_j \in Q_\ell$ executes I_j , a prefix of the transaction T_j , in α_ℓ , such that T_j is logically committed after I_j , and p_j completes T_j in β_ℓ .
4. Every process p_j , $\{p_1, \dots, p_k\} \setminus Q_\ell$, executes I'_j in α_ℓ .
5. For every process p_j from which p_0 does not read in $\alpha_\ell\beta_\ell\gamma_\ell$, α_ℓ^j is a k -independent execution, in which p_j executes I_j instead of I'_j , and all other processes take the same steps as in α_ℓ ; in α_ℓ^j , p_j does not modify any base object o , such that, in $\alpha_\ell\beta_\ell$ p_0 reads o from a process p_h , $h < j$.

For $\ell = k$, we get an execution $\alpha_\ell\beta_\ell\gamma_\ell$, such that p_0 reads from k different processes in P_k (Condition 2). The theorem follows since p_0 accesses k different base objects,

The proof is by induction on ℓ . We first show that all conditions hold for the base case, $\ell = 0$:

1. Since all the transactions in α_0 write to items that are not accessed by T_0 in any execution and all the reads of the transaction in α_0 are from items not written by T_0 , and since the STM is k -progressive, T_0 completes successfully in γ_0 .
2. β_0 is an empty execution interval and p_0 does not take any step in $\alpha_0\beta_0$. Hence, p_0 does not read from any process in $\alpha_0\beta_0$ and P_0 is empty.
3. Q_0 is empty, and the condition vacuously holds.
4. Every process $p_j \in \{p_1, \dots, p_k\} \setminus Q_0 = \{p_1, \dots, p_k\}$ executes T'_j ; there are no non-trivial conflicts in this workload and since the STM is k -progressive the execution interval $\alpha_0 = I'_1 \dots I'_k$, such that p_i executes in I'_i , a prefix of the transaction T'_i and T'_i is logically committed after I'_i , is valid.
5. For every process p_j from which p_0 does not read, α_0^j is a k -independent execution, as all the transactions are nonconflicting, and since the STM is oblivious, α_0^j is a valid execution. Furthermore, β_0 is empty, thus, for every j , p_j does not modify in α_0^j any base object o , accessed by p_0 in β_0 , and the condition trivially holds.

For the induction step, assume an execution $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$ satisfies the above conditions. Consider the subset $V_{\ell-1}$ of $\{p_1, \dots, p_k\} \setminus P_{\ell-1}$ from which p_0 does not read in $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$. If $V_{\ell-1}$ is empty then p_0 reads from all the processes and the theorem holds. Otherwise, one of the processes in $V_{\ell-1}$ is used to construct the next step. For

example, for $\ell = 1$, Figure 8.5(b) shows what happens if p_i is chosen from V_0 , so its execution is perturbed to construct $\alpha_1\beta_1\gamma_1$.

Pick an arbitrary process $p_j \in V_{\ell-1}$ and consider the execution $\alpha_{\ell-1}^j$, in which p_j executes I_j instead of I'_j , and other processes take the same steps as in α_ℓ .

The execution $\alpha_{\ell-1}^j$ is k -independent, since all the transactions are nonconflicting. Since the STM is oblivious, $\alpha_{\ell-1}^j$ is a valid execution, and since the STM is k -progressive T_j is logically committed in $\alpha_{\ell-1}^j$. Since the STM is oblivious, $\alpha_{\ell-1}^j$ is indistinguishable to every process in $\{p_1, \dots, p_k\} \setminus \{p_j\}$ from $\alpha_{\ell-1}$. Furthermore, by the inductive assumption, p_j does not modify in $\alpha_{\ell-1}^j$ any base object o , if in $\alpha_{\ell-1}\beta_{\ell-1}$ p_0 reads o from a process p_h , $h < j$. Thus, p_0 reads the same values as in the execution of $\beta_{\ell-1}$, and there is an execution $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$ such that $\beta_{\ell-1}^j$ and $\beta_{\ell-1}$ are indistinguishable, and p_0 runs solo in $\gamma_{\ell-1}^j$.

Assume that p_0 does not read from p_j also in $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$. Then p_0 takes the same steps in $\gamma_{\ell-1}^j$ and $\gamma_{\ell-1}$ and T_0 is committed in $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$. Let m_1, \dots, m_k be the base objects that are private to p_0 after $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$.

The pending transactions in the execution $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$ are not conflicting. Since the STM is k -progressive and T_j is logically committed after I_j , it commits (writing to t_j) when executed solo after $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$. Consider the execution $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^jJ_j$, such that J_j is the execution of T_j until it commits (see Figure 8.3(c)). In a manner similar to Claim 8.6, we show that p_j must modify the state of m_j in J_j .

Claim 8.8 p_j modifies the state of m_j in J_j .

Proof: We first show that p_j does not apply nontrivial primitive to m_j in $\alpha_{\ell-1}^j$. Otherwise, let τ be the first such step. Let $\hat{\alpha}_{\ell-1}^j$ be the prefix of $\alpha_{\ell-1}^j$ preceding τ . Consider the execution $\hat{\alpha}_{\ell-1}^j\beta$, where each pending transaction preceding I_j in $\alpha_{\ell-1}^j$ executes solo to completion in β ; since there are no nontrivial conflicts in $\hat{\alpha}_{\ell-1}^j\beta$, and the STM is k -progressive, all the transactions running in β complete. Let C be the configuration at the end of this execution. Only T_j is pending in C , however, T_j is not logically committed in C . In a solo execution of T_0 from C , T_0 is committed, making m_j private to p_0 . Since the STM is not eager, p_j does not apply τ in C .

A similar argument shows that p_j does not apply nontrivial primitive to m_j also in $\alpha_{\ell-1}$.

Next, we prove that p_0 does not modify the state of m_j in $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$. Otherwise, a nontransactional read operation to t_j of p_0 after $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$ returns a value that is not the initial value of m_j , whereas no committed transaction writes to t_j in $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$, contradicting our correctness condition. Since p_0 does not read from p_j also in $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$, p_0 applies the same steps in $\beta_{\ell-1}^j\gamma_{\ell-1}^j$ and in $\beta_{\ell-1}\gamma_{\ell-1}$. As we have shown, p_j does not apply a nontrivial primitive to m_j neither in $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$ nor in $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$, thus, p_0 does not modify the state of m_j also in $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$.

If p_j does not modify the state of m_j also in J_j , then a nontransactional read by p_0 to t_j after $\alpha_{\ell-1}^j \beta_{\ell-1}^j \gamma_{\ell-1}^j J_j$ returns the initial value of m_j , since reads are uninstrumented. This contradicts our correctness condition since T_j , which writes to t_j a value that is different from the initial value of m_j , commits before the nontransactional read by p_0 . ■

Therefore, p_j applies a nontrivial primitive to m_j in some step during J_j , after the execution of T_0 , in contradiction to privatization safety. Thus, p_0 must read from p_j in $\alpha_{\ell-1}^j \beta_{\ell-1}^j \gamma_{\ell-1}^j$.

Let s_j be the number of steps until p_0 reads from p_j for the first time in $\gamma_{\ell-1}^j$. Pick a process p_{j_ℓ} such that s_{j_ℓ} is the smallest, and if $s_{j_\ell} = s_{h_\ell}$ then $j_\ell > h_\ell$.

Let the execution interval α_ℓ be $\alpha_{\ell-1}^{j_\ell}$. The execution interval β_ℓ is $\beta_{\ell-1}$ extended with the first $s_{j_\ell} - 1$ steps of p_0 in $\gamma_{\ell-1}^{j_\ell}$, then a solo execution of p_{j_ℓ} completing T_{j_ℓ} , and finally, the s_{j_ℓ} step of p_0 from $\gamma_{\ell-1}^{j_\ell}$, which reads from p_{j_ℓ} . Since T_{j_ℓ} is logically committed in α_ℓ , and the STM is k -progressive, T_{j_ℓ} commits in β_ℓ . The execution interval γ_ℓ is defined as a solo execution of p_0 completing T_0 .

It remains to verify that the conditions hold for $\alpha_\ell \beta_\ell \gamma_\ell$.

1. T_0 completes successfully as there is no incomplete conflicting transaction after $\alpha_\ell \beta_\ell$, and the STM is k -progressive.
2. By the induction assumption, p_0 reads from at least $\ell - 1$ processes, $P_{\ell-1}$, in $\alpha_{\ell-1} \beta_{\ell-1}$, not including p_{j_ℓ} that was chosen in the last iteration. The executions $\alpha_{\ell-1}$ and α_ℓ are indistinguishable to all the processes p_h , for $h < j_\ell$. Furthermore, since the STM is oblivious, $\alpha_{\ell-1}$ and α_ℓ are indistinguishable to all the processes p_h , for $h > j_\ell$. Hence, p_0 reads from at least the same $\ell - 1$ processes in $\alpha_\ell \beta_{\ell-1}$. In addition, p_0 reads from p_{j_ℓ} in $\alpha_\ell \beta_\ell$. Thus, $P_\ell \supseteq P_{\ell-1} \dot{\cup} \{p_{j_\ell}\}$, and $|P_\ell| \geq |P_{\ell-1}| + 1 \geq \ell$.
3. By the induction assumption, $Q_{\ell-1}$ is a subset of $P_{\ell-1}$, such that every process $p_h \in Q_{\ell-1}$ executes I_h in $\alpha_{\ell-1}$, and completes T_h in $\beta_{\ell-1}$. Only p_{j_ℓ} is in $Q_\ell \setminus Q_{\ell-1}$, and it executes I_{j_ℓ} in α_ℓ and completes T_{j_ℓ} in β_ℓ . Since $\alpha_{\ell-1}$ and α_ℓ are indistinguishable to all the processes in $\{p_1, \dots, p_k\} \setminus \{p_{j_\ell}\}$, and since only p_{j_ℓ} switched from I_{j_ℓ}' in $\alpha_{\ell-1}$ to I_{j_ℓ} in α_ℓ , all the processes $p_h \in Q_\ell \setminus \{p_{j_\ell}\}$ execute I_h in α_ℓ and complete T_h in $\beta_{\ell-1}$, which is the prefix of β_ℓ .
4. By the induction assumption, every process $p_h \in \{p_1, \dots, p_k\} \setminus Q_{\ell-1}$, executes I_h' in $\alpha_{\ell-1}$. Since only $p_{j_\ell} \in \{p_1, \dots, p_k\} \setminus Q_{\ell-1}$ switched from I_{j_ℓ}' in $\alpha_{\ell-1}$ to I_{j_ℓ} in α_ℓ , and since $p_{j_\ell} \notin \{p_1, \dots, p_k\} \setminus Q_\ell$, every process $p_h \in \{p_1, \dots, p_k\} \setminus Q_\ell$ executes I_h' in α_ℓ .

5. Assume, by way of contradiction, that for some j , p_j modifies in α_ℓ^j the base object o , which in $\alpha_\ell\beta_\ell$ p_0 reads from p_h , $h < j$. Denote by σ the step during $\alpha_\ell\beta_\ell$, in which p_0 reads from p_h . There is a step, σ' , which is either σ or follows σ during $\alpha_\ell\beta_\ell$, in which p_0 reads from $p_{h_{\ell'}}$ $\in Q_\ell$, since p_0 always reads from a process in Q_ℓ in the last step of $\alpha_\ell\beta_\ell$. Consider the iteration, ℓ' , in which $p_{h_{\ell'}}$ was chosen to switch from $T_{h_{\ell'}}$ to $T_{h_{\ell'}}$. Since the STM is oblivious, the executions $\alpha_{\ell'-1}$ and $\alpha_{\ell'}$ are indistinguishable to the processes in $\{p_1, \dots, p_k\} \setminus \{p_{h_{\ell'}}\}$. Process p_j should have been chosen in the iteration ℓ' , since if σ' follows σ , then $s_j < s_{h_{\ell'}}$, otherwise, σ' is σ , i.e., $h = h_{\ell'}$ and $j > h_{\ell'}$. ■

8.5 Reducing the Cost of Privatization

The previous lower bound, stated as the minimum between the number of privatized items and the level of parallelism, points out a way to reduce the cost associated with privatization, namely, to limit the parallelism offered by the STM. We next show how this tradeoff can be exploited, by sketching a “counter-example” STM, which is a variant of RingSTM [95]. The variant, called *VisibleRingSTM*, reduces the cost of privatization while limiting parallelism.

RingSTM is oblivious and privatization-safe, but not progressive; privatizing k items requires $O(c)$ accesses to base objects, where c is the number of concurrent transactions. RingSTM represents transactions’ read and write sets as Bloom filters [22]. Transactions commit by enqueueing a Bloom filter onto a global ring; the Bloom filter representing the read set of a transaction is used only locally by the transaction. On validation, a transaction T checks for intersections between the read set of T and the write sets of other logically committed transactions in the ring, and aborts in case of a conflict. In the commit phase, T ensures that a write-after-write ordering is preserved. This is done by checking for intersections between the write set of T and the write sets of other logically committed transactions in the ring. In RingSTM a transaction blocks until all concurrent logically committed transactions are completed, therefore RingSTM is not l -progressive, for any $l \geq 1$.

In *VisibleRingSTM*, we make the read set Bloom filter also visible to other transactions, just like the write set filter. In the commit phase, T ensures that a write-after-read ordering is preserved, in addition to the write-after-write ordering, as in RingSTM. This is done by checking for intersections between the write set of T and the (visible) read sets of other logically committed transactions in the ring. In addition, it checks for intersections with the write sets of these transactions, like RingSTM. Intersection between the read set of T and the write set of another transaction is checked by validation. There is no need to check for intersection between read sets, as these are trivial conflicts that should not interfere. Finally, waiting for all logically committed transactions to complete (at the end of the commit phase) is removed in *VisibleRingSTM*,

as the write-after-read and write-after-write ordering ensure that all the concurrent conflicting transactions have completed.

The steps of the commit operation of a transaction T in VisibleRingSTM are:

1. Check intersection of the read set filter with the write set filters of concurrent transactions preceding T in the ring (validating the read set),
2. If there is a read-after-write conflict, then abort.
3. Commit T in the ring; if not successful start again from 1 (otherwise, T is logically committed).
4. Check intersection of the write set filter with the write set *and read set* filters of concurrent transactions preceding T in the ring,
5. Wait until all preceding transactions with write-after-write and *write-after-read* conflicts are completed.
6. Complete T .

In VisibleRingSTM, a transaction aborts only due to read-after-write conflicts with other logically committed transactions, and blocks after it is logically committed only due to write-after-write or write-after-read conflicts with other logically committed transactions. A privatizing transaction accesses the c ring entries of concurrent logically committed transactions, the items in its data set and the global ring index.

The cost of a privatizing transaction can be bounded by $O(c_0)$, for any $c_0 > 1$, by using a ring of size c_0 ; thus, a privatizing transaction needs to access at most c_0 ring entries. In order to commit, a transaction scans the ring for an empty entry. When there are at most c_0 concurrent transactions, it will find an empty entry, become logically committed, and continue as in VisibleRingSTM. This variant is $(c_0 - 1)$ -progressive, but a transaction blocks in executions with more than c_0 concurrent transactions (even if they are not conflicting). Thus, the cost of privatization is reduced by limiting the progress of concurrent transactions.

8.6 Bounds for Disjoint-Access Parallel STMs

This section discusses the relationship between oblivious and disjoint-access parallel STMs. We show that with invisible reads, oblivious STMs are a generalization of disjoint-access parallel STMs.

Recall that disjoint-access parallelism ensures that transactions that are not connected in the conflict graph do not concurrently contend on a (or even access the same) base object. Also, it is important to note that obliviousness ensures a certain behavior of a transaction T only after a k -independent execution without nontrivial conflicts with T . To show that indeed disjoint-access parallel STMs with invisible reads are oblivious, we only need to consider these special executions.

Consider a transaction T by process p_0 and any pair of k -independent executions, α_1 and α_2 , where none of the transactions in these two executions has a nontrivial conflict with T . The invisibility of the reads implies that the execution $\alpha_1 T$ is indistinguishable to p_0 from an execution $\alpha'_1 T$, in which the read sets of all the transactions in α_1 are empty. The same is true for $\alpha_2 T$ and $\alpha'_2 T$, where the read sets of the transactions in α'_2 are empty. Next, consider the conflict graphs of T in the executions $\alpha'_1 T$ and $\alpha'_2 T$. In both executions the data set of T does not intersect with any of the data sets of the transactions preceding it, therefore no transaction is connected to T in both conflict graph. Disjoint-access parallelism implies that in both executions T does not access any base object that was accessed by the transactions preceding it, and therefore the execution interval of T in both executions is the same. This means that $\alpha'_1 T$ and $\alpha'_2 T$ are indistinguishable to p_0 , and by applying the invisibility of reads again, $\alpha_1 T$ and $\alpha_2 T$ are indistinguishable to p_0 . Going back to the definition of obliviousness, and since we considered an arbitrary pair of k -independent executions, this implies the implementation is oblivious.

Hence, Theorem 8.4 implies:

Corollary 8.9 *For any privatization-safe STM that is 1-progressive, disjoint-access parallel and with invisible reads, there is a privatization workload in which transactions have nonempty read sets, for which there is an execution where the size of the data set of a transaction privatizing k items is $\Omega(k)$.*

This generalization is strict, since several clock-based STMs [31, 82, 84] are not disjoint-access parallel, but they are oblivious, so our lower bound applies to them, since they have invisible reads.

When reads are visible, we cannot swap $\alpha_1 T$ with $\alpha'_1 T$, and we need a notion of disjoint-access parallelism that takes into account only nontrivial conflicts. Recall that a conflict is nontrivial only if at least one of the transactions is writing to the item. An edge in a *nontrivial conflict graph* connects two transactions only if they have a nontrivial conflict. An STM is *nontrivial disjoint-access parallel* if two processes executing transactions T_1 and T_2 contend, only if there is a path between T_1 and T_2 in their nontrivial conflict graph.

To show that nontrivial disjoint-access parallel STMs are oblivious we consider again an arbitrary pair of k -independent executions α_1 and α_2 , followed by a nonconflicting transaction T . Instead of swapping α_1 and α_2 with executions with empty read sets, we note that in the nontrivial conflict graphs of executions $\alpha_1 T$ and $\alpha_2 T$, T is not connected to any other transactions. By nontrivial disjoint-access parallelism, the execution interval of T in both executions is the same, and the implementation is oblivious.

Therefore, Theorem 8.7 implies:

Corollary 8.10 *For any privatization-safe STM that is l -progressive and nontrivial disjoint-access parallel there is a privatization workload in which update transactions have nonempty read sets, for which there is an execution where a transaction privatizing m items accesses $\Omega(k)$ base objects, where $k = \min\{l, m\}$.*

Chapter 9

Related Work

Our results join other efforts to chart the boundaries of STM implementations. Guerraoui and Kapalka [46] prove that obstruction-free implementations of software transactional memory cannot ensure *strict* disjoint-access parallelism. This property requires transactions with disjoint data sets not to access a common base object. Our lower bound applies to the notion of disjoint-access parallelism as originally defined in [62]. In contrast, the result of Guerraoui and Kapalka [46] does not hold for this weaker requirement. Indeed, Herlihy et al. [55] present an obstruction-free and disjoint-access parallel STM. Obstruction-freedom does not prevent interfering concurrent processes from starving each other and thus, the implementation presented in [55] does not guarantee that a read-only transaction eventually terminates successfully.

Elsewhere, Guerraoui and Kapalka [47] prove a lower bound on the number of steps a process takes to successfully terminate a transaction, for every implementation that uses invisible reads, is single-version, and never aborts a transaction unless it conflicts with another live transaction.

Multi-Version permissiveness (MV-permissiveness) [79] is a practical notion of permissiveness, usually associated with keeping many versions: it never aborts read-only transactions, and it aborts other transactions only due to a conflicting transaction (which writes to a common item), thereby avoiding spurious aborts. Theorem 7.7 allows multi-version implementations, but requires read-only transactions to terminate successfully, regardless of overlapping transactions. This means that the lower bound in [47] holds for weakly progressive implementations, while our result holds for MV-permissive implementations.

The proof technique in Chapter 7 draws ideas from the lower bounds on the step complexity of update operations in snapshot objects [9, 63]. Israeli and Shirazi [63] prove an $\Omega(m)$ lower bound on the number of steps to update a component in an m -component single-writer snapshot objects, implemented from single-writer registers. Attiya, Ellen and Fatourou [9] extend this lower bound to implementations of m -component multi-writer objects from base objects of any type.

Next, we survey several (strictly serializable) implementations, each providing one or two properties: disjoint-access parallelism or read-only transactions that are either invisible or wait-free; as indicated by our impossibility result (Theorem 7.4), none of the implementations provide all three properties. This discussion is summarized in Table 9.1.

Lazy Snapshot Algorithm (LSA) [82] is a multi-version transactional memory, with invisible, wait-free read-only transactions, which relies on a single shared monotonically increasing counter to determine a unique commit timestamp for transactions, so as to totally order committed transactions writing different versions of the same data item. This allows each read-only transaction to return a consistent snapshot of the memory, despite concurrent writing transactions. *Transactional Locking II* (TL2) [31] is another implementations relying on a centralized mechanism, a global clock. Although having low-cost read-only transactions, using invisible reads, read-only transactions may never commit, as having only a single version per item, the transaction aborts if one of the items it read is overwritten. While the former approach introduces a single hot-spot accessed by all transactions, regardless of their data sets, and is therefore not disjoint-access parallel, the latter approach relies on a single source of synchronous time, which may not be realistic for systems with a large number of processes.

There are other STM implementations without a centralized hot-spot. Avni and Shavit [18] present a *thread-local clock* mechanism that provides a decentralized solution for maintaining a consistent view. The key idea is using Lamport clock (scalar causal timestamps) instead of the real-time global clock. Integrated with TL2, this mechanism provides an STM supporting invisible read-only transactions, without a centralized contention point. A drawback of this algorithm is that transactions that terminated long before the current one may cause it to fail since the timestamp recorded for them is not current enough. Thus, read-only transactions do not necessarily commit after a finite number of steps of the process executing it. Imbs and Raynal [61] propose an opaque single-version STM with no centralized hot-spot. This algorithm has visible reads as for each item, the algorithm maintains the set of transactions reading the item. At commit time, a transaction uses these reading sets to invalidate transactions reading from an item to which the transaction writes.

PermiSTM [15] is a single-version STM that is both strictly disjoint-access parallel and strongly progressive, and it also satisfies MV-permissiveness. In PermiSTM, update transactions are not obstruction-free, since they may block due to other conflicting transactions, matching the impossibility result in [46]; a read-only transaction modifies a number of memory locations that is equal to the size of its read set, in compliance with Theorem 7.7. It has been proved [79, Theorem 2] that a *weakly* disjoint-access parallel cannot be MV-permissive. PermiSTM, satisfying the even stronger property of strict disjoint-access parallelism, shows that this impossibility result depends on a progress condition of individual operations executed by transactions: a transaction *delays only*

Algorithm	Disjoint-access parallel	Invisible read-only txs	wait-free read-only txs
Riegel et al. [82] (LSA)	no	yes	yes
Dice et al. [31] (TL2)	no	yes	no
Avni and Shavit [18] (TLC)	yes	yes	no
Imbs and Raynal [61]	yes	no	no
Attiya and Hillel [15] (PermiSTM)	yes	no	yes
Attiya et al. [10] (Partial scan)	no	no	yes

Table 9.1: Comparison of strict serializable STMs, showing disjoint-access parallelism, and whether read-only transactions are invisible or wait-free.

due to a pending operation (by another transaction). In PermiSTM, a transaction may delay due to another transaction reading from its write set, even if no operation of the reading transaction is pending.

A read-only transaction can be considered as a *partial scan* operation [10]: a *partial snapshot object* is an atomic snapshot object [4], where processes can scan any subset of the components. In the wait-free algorithm for partial snapshot objects [10], scanners announce which components they are currently attempting to scan, i.e., read-only transactions are visible. In this algorithm all transactions access the set of active objects, therefore it is not disjoint-access parallel.

One possible way to circumvent lower bounds is to weaken the assumptions, e.g., the safety condition. *Serializability* provides a weaker guarantee on the ordering of transactions (it does not have to respect the real-time order of non-overlapping ones). Nevertheless, our impossibility results hold also for serializable STMs that preserve the per-process order (Theorem 7.13 and Theorem 7.14). Indeed, none of the serializable STM implementations presented in the literature, e.g., [19, 40, 76, 85], provides disjoint-access parallelism and has invisible read-only transactions that always eventually commit.

In addition to being disjoint-access parallel for allowing scalability and optimizing read-only transactions for efficiency, it is imperative to be able to run nontransactional code together with transactions to gain better performance. Therefore, many STMs support privatization; Table 9.2 compares their obliviousness, visibility of reads and progressiveness.

Oblivious STMs supporting privatization [27, 32, 33, 39, 72, 95] avoid the cost of tracking the read sets of other transactions, especially if they are not conflicting. The visibility of reads is not implied by the obliviousness of the STM: Some oblivious STMs use invisible reads [27, 72, 95], making their read set nonexistent for other transactions. Other STMs, e.g., [33], use *partially visible* reads [70], meaning that other transactions cannot determine which transaction exactly is reading the item. Some oblivious STMs even use visible reads, e.g., [39], however, their execution is unaffected by trivial, read-

Algorithm	Oblivious	Visible reads	ℓ -Progressive
Olszewski et al. [77] (JudoSTM)	yes	no	$\ell = 0$
Spear et al. [95] (RingSTM)	yes	no	$\ell = 0$
Menon et al. [72]	yes	no	$\ell = 0$
Dalessandro et al. [27] (NOrec)	yes	no	$\ell = 0$
Marathe et al. [70]	yes	partially	$\ell = 0$
Dice et al. [33] (Private transactions)	yes	partially	$\ell = 0$
Gottschlich et al. [39] (InvalSTM)	yes	yes	$\ell = 0$
This dissertation (VisibleRingSTM)	yes	yes	$\ell = c_0 - 1$
Dice and Shavit [32] (TLRW restricted to slotted readers)	yes	yes	$\ell = n$
Dice and Shavit [32] (TLRW)	no	partially	$\ell = n$
Lev et al. [67] (SkySTM)	no	partially	$\ell = n$
Attiya and Hillel [15] (PermiSTM)	no	partially	$\ell = n$

Table 9.2: Comparison of STMs supporting privatization, showing obliviousness, visibility of reads, and the progressivenss.

read conflicts. Our lower bounds hold for all these STMs.

TLRW [32] uses read locks, making reads visible. The lock contains a byte per each *slotted* reader, and a reader-count that is modified by other, *unslotted* readers. Slotted readers only write to their slot when reading, so they are unaware of other reads, while unslotted processes read and write to a common counter, and their execution is affected by other reads to the same item (read-read conflict). Therefore, TLRW is oblivious when restricted to slotted readers, and, in accordance with our lower bound, the number of locations accessed by a privatizing transaction is linear in the number of slotted readers.

Our lower bounds indicate that providing efficient privatization requires to compromise parallelism. Inspecting many STMs supporting privatization, e.g., [27, 39, 70, 72, 77, 95], reveals that they limit parallelism, being 0-progressive. These STMs, which we survey next, while having different implementations mechanisms, all sacrifice parallelism by using global synchronization mechanisms, rendering the STM to not being disjoint-access parallel. These mechanisms force the order by which transactions commit, while preventing other transactions to make progress and commit.

JudoSTM [77] transforms code to support transactional execution on-the-fly at runtime. The coarse-grained commit variant uses single lock; when a transaction commits its changes, no concurrent transaction can commit and complete. NOrec [27], like JudoSTM, uses a single sequence lock yet it is livelock-free—a transaction aborts only due to a conflict with concurrent, logically committed transaction.

As described in Section 8.5, RingSTM [95] uses Bloom filters to represent the read and write sets of transactions. Appending an entry to a global ring effects a logical commit of a writing transaction. A logically committed transaction may block due to

nonconflicting concurrent logically committed (write) transactions. The default variant of RingSTM is livelock-free, permitting concurrent disjoint transactions to commit their changes in parallel. The single-writer variant forbids committing the changes of concurrent transactions in parallel, even of disjoint transactions.

InvalSTM [39] applies commit-time *invalidation*. During the commit phase a transaction invalidates all concurrent transactions with which it has nontrivial conflicts. Like RingSTM, it uses Bloom filters to compress the read and write sets of a transaction. To commit, a transaction acquires global locks preventing other transactions from committing or making any progress or new transactions to begin.

Menon et al. [72] suggest two STMs that provide privatization safety. The first uses a global linearization timestamp. It enforces start and commit linearization, meaning the start, commit, and completion order of transactions are the same. Alternatively, ordering is imposed only among conflicting transactions, however, the completion order is similar to the commit order in all transactions. In both cases, this is done by iterating over other processes with concurrent transactions, waiting for their completion.

Marathe et al. [70] use a globally synchronized clock and a linked list containing all active transactions to guarantee transaction consistency. If a transaction identifies a conflict with a concurrent transaction it blocks until all active transactions complete.

Other STMs [15, 32, 67] are progressive. SkySTM [67] is designed to work in hybrid transactional memory systems, combining software and hardware. It uses a global counter to indicate when a writing transaction that has conflicted with a reader commits. It also has partially visible reads. A transaction blocks only due to conflicting transactions and aborts only due to a read-write conflict with another transaction. TLRW [32] is a lock-based STM, in which a transaction aborts only due to a conflict with another transaction and blocks only while waiting for a lock on an item. PermiSTM [15] uses locks, like TLRW, and tracks readers through read counters, like SkySTM. It is strongly progressive, and a transaction T blocks only while another transaction is reading an item to which T is trying to write.

All the STMs discussed so far, support *implicit privatization* [70], meaning the implementation is required to handle all transactions as if they are potentially privatizing items; this incurs excessive overhead for all transactions. In *explicit privatization* [93], the application explicitly annotates privatizing transactions, and the STM implementation can be optimized to handle such transactions efficiently; this approach is error-prone and places additional burden on the programmer, which STM tries to avoid in the first place [67].

Some experiments [36, 93, 101] tested techniques used to support implicit privatization in implementations with invisible reads. The results show a significant impact on the scalability and performance relative to STMs supporting explicit privatization; in some cases, the performance degrades to be even worse than in sequential code.

Private transactions [33] attempt to combine the ease of use of implicit privatization

with the efficiency benefits of explicit privatization. A private transaction inserts a *quiescing barrier* that waits until all active transactions are completed; thus other, non-privatizing transactions avoid the overhead of privatization. The barrier accesses an array whose size is proportional to the maximal parallelism, demonstrating again the tradeoff between parallelism and privatization cost, in oblivious STMs. As in VisibleRingSTM, the size of the active transactions array can be bounded, thereby reducing the overhead of the barrier, but at the cost of limiting the level of parallelism.

An alternative way to provide strong atomicity is thorough *static separation* [2]. This is a discipline in which each data item is accessed either only transactionally or only nontransactionally. In order to access items nontransactionally, a transaction copies them to a private buffer, trivially incurring the cost predicted by our lower bound. *Dynamic separation* [1] allows data to change access modes without being copied, simply by setting a protection mode in the item. Dynamic separation requires the programmer to access all items to become unprotected, i.e., privatized, in accordance with our lower bound.

Chapter 10

Summary

Many core problems in current multiprocessing architectures revolve around the coordination of access to shared resources and can be captured as concurrent data structures. This thesis investigated issues in implementations of concurrent data structures. Our main goal was to indicate effective design choices that enable utilizing the real power provided by multicore and multiprocessing systems.

The first part of the dissertation focused on methodologies for deriving implementations of concurrent data structures. We present, in Chapter 3, a highly-concurrent implementation of k RMW, with improved throughput even when there is contention; it stores a constant amount of information per data item, independently of k . Chapter 4 introduced a new built-in coloring approach for designing high-throughput implementations of linked list data structures. We show a DCAS-based implementation of insertions and removals in a doubly-linked list; for a deque and priority queues, where nodes are removed only at the ends, the implementation uses only CAS.

We believe that DCAS provides critical leverage for implementing concurrent data structures in general, and in particular allowing to implement k RMW, for any $k > 2$, with locality that is difficult, perhaps impossible, to obtain using only CAS. Currently, few architectures provide DCAS in hardware, but DCAS is an ideal candidate to be supported by *hardware transactional memory* [25,29,30,34,56,80], being a short transaction with static data set of minimal size. Alternatively, DCAS can be simulated in software from CAS [8,38], or by applying a simple randomized algorithm [49].

Our algorithms are intended as a proof-of-concept and require further optimizations to make them more practical; it is also necessary to implement a *search* operation in order to support the full functionality of priority queues and lists. Optimizations may include validating the state of the data structure more frequently, and utilizing the strength of DCAS in a way similar to the *two-handed emulation* method [43]. A standard technique can be used to transform the algorithms which apply helping to avoid blocking into wait-free algorithms: after completing its operation the process helps some pending operation before it is allowed to start a new operation. We need

to employ this method carefully in order to maintain the locality of the algorithms. For example, each operation can maintain a queue of its helping operations. After completing, the operation helps all the operation in its queue.

A flexible dynamic variant of our implementation of k RMW can serve as the basis for a *dynamic* STM acquiring ownership at *encounter-time*. Encounter-time semantics enables early detection of conflicts, particularly important when transactions are long; on the other hand, early conflict detection may lead to unnecessary aborts [31, 41]. It is also possible to acquire ownership at *commit-time* with a *speculative execution* scheme [31], first accumulating the data items the operation needs to access, and then acquiring them, with the algorithm presented here. Realizing a full-fledged STM requires to address many additional issues, e.g., memory management, handling read-only data, and optimizing the common case.

The second part of the dissertation studied boundaries and tradeoffs that are inherent in implementations of transactional memory. Chapter 7 showed that no transactional memory implementation can be disjoint-access parallel and have invisible, wait-free read-only transactions. There are implementations that are disjoint-access parallel and have read-only transactions that are invisible but do not always eventually commit [18, 55], while others have invisible, wait-free read-only transactions but are not disjoint-access parallel [82].

In principle, the invisibility of read-only transactions can also be sacrificed in order to keep them wait-free, and the implementation disjoint-access parallel. This can be done by treating the read set together with the write set and adapting a dynamic disjoint-access parallel implementation of multi-word synchronization operator to atomically validate the read set and update the write set of a transaction. For example, Harris et al. [52] give a disjoint-access parallel implementation of a multi-word compare-and-swap operation: the implementation requires locking all items in the data set of the transaction, one by one; if another operation already holds the lock on a word, the operation helps the conflicting operation, thereby guaranteeing progress. (This algorithm is not wait-free, but additional helping can make it wait-free without sacrificing the other properties.)

Thus, each of the assumptions made in our impossibility result (Theorem 7.4) is necessary, since removing either of them admits an implementation with the two remaining properties. Furthermore, the application of [52] in a TM demonstrates our lower bound (Theorem 7.7): A read-only transaction has to lock all items in its read set, and therefore it writes to distinct base objects representing these locks.

Chapter 8 studied the theoretical complexity of privatization that allows uninstrumented nontransactional reads, and shows an inherent cost, linear in the number of privatized items. Privatizing transactions in STMs with invisible reads must have a data set of size k , where k is the number of privatized items. A more involved proof shows that even with visible reads, the privatizing transaction must access $\Omega(k)$ mem-

ory locations, where k is the minimum between the number of privatized items and the number of concurrent transactions that make progress. Both results assume that the STM is oblivious to different non-conflicting executions and guarantees progress in such executions. The specific assumptions needed to prove the bounds indicate that limiting the parallelism or tracking the data sets of other transactions are the price to pay for efficient privatization.

Privatization was informally characterized as suffering from two subproblems: The *delayed cleanup* problem [66], in which transactional writes interfere with nontransactional operations, and the *doomed transaction* problem [99], in which transactional reads of private data lead to inconsistent state. Our definition of privatization safety (Property 1) formalizes the first; our results show that this problem by itself is an impediment to the efforts to provide efficient privatization.

Next we discuss some open questions and suggest future research directions.

It is interesting to explore other applications of our built-in coloring approach, e.g., for tree-based data structures. It is possible to maintain the tree legally colored, i.e., assign distinct colors to a node and its parent, while adding and removing nodes in the tree. Nevertheless, the implementation needs to be tailored to accommodate the semantics of the tree operations.

Chapter 5 considers using the implementation of Attiya and Dagan [8], instead of DCAS in our algorithms, to get implementations that are $O(\log^* n)$ -local nonblocking. It is interesting to obtain locality properties that are independent of n , without using DCAS. A lower bound presented in [8] indicates that this might be impossible, but the exact bounds and tradeoffs should be explored. Even more intriguing is to investigate whether $O(k)$ is the best locality that can be achieved, even with DCAS.

Chapter 7 shows that a disjoint-access parallel implementation with invisible read-only transactions that may terminate unsuccessfully is not permissive with respect to opacity, strict serializability, serializability or snapshot isolation (Theorems 7.4, 7.9, and 7.13, respectively). *Causal serializability* [81] is weaker than serializability since it allows different processes to have a different view of the system. Riegel et al. [85] proposed a causally serializable STM implementation that supports invisible reads and is disjoint-access parallel; in this implementation, read-only transactions may abort infinitely many times. This leaves open the question of whether our results holds for causally serializable STMs, or whether the algorithm of [85] can be refined to have read-only transactions that eventually commit.

It is also interesting to further investigate the connections between our impossibility results and the study of *unnecessary aborts* [40, 45, 65] or *wasted work* in STM implementations.

Chapter 9 discusses some STMs that maintain visible reads, yet they are oblivious [33, 39]. SkySTM [67] has visible reads, and it avoids the cost of the privatizing transaction by not being oblivious; it makes transactions with trivial read-read conflicts

visible to each other. Since SkySTM is not oblivious, our lower bounds do not hold for it. SkySTM, however, demonstrates the alternative cost of not being oblivious, since any writing transaction—not only privatizing transactions—writes to a number of base objects that is linear in the size of its data set, not just the write set. It remains an interesting open question whether this is an inherent tradeoff, or whether there is an STM such that a privatizing transaction accesses $O(1)$ base objects, and any writing transaction writes to a number of base objects that is linear in the size of its write set.

Strong privatization-safety [67] further guarantees that no primitive (including a read) is applied to a private location of a process that completed a privatizing transaction. It formalizes the second problem with privatization, of doomed transactions, and it would be interesting to investigate the cost of supporting it.

Another future research direction is deriving additional quantitative results on the complexity of transactions, in particular, explore tradeoffs of read-only transactions in STM implementations that are not disjoint-access parallel, and the step complexity of both update and read-only transactions in executions with no conflicts.

Taking a broader prospective, our results demonstrate that transactional memory faces significant limitations, and it is worthy to consider other tools and methodologies that are, perhaps, not as sophisticated as transactional memory, yet facilitate writing concurrent application.

Multi-word synchronization, such as our k RMW implementation, are the analogue of static, simple and short transactions. The simple examples in Chapter 1 demonstrate how multi-word operations provide significant leverage for implementing specific data structures.

It is also worth pursuing other general methodologies that are functionality specific; the *map-reduce* paradigm is an excellent example from another field. It is possible to develop a general methodology for deriving implementation of concurrent data structures, possibly using built-in colors. Such a method may have a relatively simple interface, requiring the developer to provide some simple input—much like the input required by map-reduce—that defines the data set and protocols to be executed.

The concurrency revolution is driven by hardware changes. To allow for further theoretical results a stronger model capturing these changes is required. Modeling current multi-core architectures should include the interface for operating transactions without hiding privatization issues; it is also beneficial to expose *power tradeoffs*. These tradeoffs indicate, for example, that accessing the cache is more energy efficient, while aborting many times is less energy efficient. Finally, true scalability of existing implementations is a big question, and there is a great need for a measure that can precisely predict this property, more precisely than disjoint-access parallelism.

Bibliography

- [1] Martín Abadi, Andrew Birrell, Tim Harris, Johnson Hsieh, and Michael Isard. Implementation and use of transactional memory with dynamic separation. In *Proceedings of the 18th International Conference on Compiler Construction (CC)*, pages 63–77, 2009.
- [2] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM SIGPLAN Notices*, 43(1):63–74, 2008.
- [3] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 185–196, 2009.
- [4] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.
- [5] Yehuda Afek, Michael Merritt, Gadi Taubenfeld, and Dan Touitou. Disentangling multi-object operations. In *Proceedings of the 16th annual ACM symposium on Principles of distributed computing (PODC)*, pages 111–120, 1997.
- [6] Ole Agesen, David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul A. Martin, Nir Shavit, Guy L. Steele, and Jr. DCAS-based concurrent dequeues. *Theory of Computing Systems*, 35(3):349–386, 2002.
- [7] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.
- [8] Hagit Attiya and Eyal Dagan. Improved implementations of binary universal operations. *Journal of the ACM*, 48(5):1013–1037, 2001.

- [9] Hagit Attiya, Faith Ellen, and Panagiota Fatourou. The complexity of updating multi-writer snapshot objects. In *Proceedings of the 8th International Conference on Distributed Computing and Networking (ICDCN)*, pages 319–330, 2006.
- [10] Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In *Proceedings of the 20th annual symposium on Parallelism in algorithms and architectures (SPAA)*, pages 336–343, 2008.
- [11] Hagit Attiya and Eshcar Hillel. Built-in coloring for highly-concurrent doubly-linked lists. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 31–45, 2006.
- [12] Hagit Attiya and Eshcar Hillel. Highly-concurrent multi-word synchronization. In *Proceedings of the 9th international conference on Distributed computing and networking (ICDCN)*, pages 112–123, 2008.
- [13] Hagit Attiya and Eshcar Hillel. The cost of privatization. In *Proceedings of the 24th international conference on Distributed computing (DISC)*, pages 35–49, 2010.
- [14] Hagit Attiya and Eshcar Hillel. Highly concurrent multi-word synchronization. *Theoretical Computer Science*, 412(12–14):1243–1262, 2011.
- [15] Hagit Attiya and Eshcar Hillel. Single-version STMs can be multi-version permissive. In *Proceedings of the 12th international conference on Distributed computing and networking (ICDCN)*, pages 83–94, 2011.
- [16] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the 21st annual symposium on Parallelism in algorithms and architectures (SPAA)*, pages 69–78, 2009. To appear in *Theory of Computing Systems*.
- [17] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, 2004.
- [18] Hillel Avni and Nir Shavit. Maintaining consistent transactional states without a global clock. In *Proceedings of the 15th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 131–140, 2008.
- [19] Utku Aydonat and Tarek Abdelrahman. Serializability of transactions in software transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2008.

-
- [20] Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the 5th annual ACM symposium on Parallel algorithms and architectures (SPAA)*, pages 261–270, 1993.
- [21] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O’Neil, and Patrick O’Neil. A critique of ANSI SQL isolation levels. *SIGMOD Record*, 24(2):1–10, 1995.
- [22] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [23] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Communication of the ACM*, 51(11):40–46, 2008.
- [24] Manhoi Choy and Ambuj K. Singh. Efficient fault-tolerant algorithms for distributed resource allocation. *ACM Transactions on Programming Languages and Systems*, 17(3):535–559, 1995.
- [25] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of AMD’s advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems (EuroSys)*, pages 27–40, 2010.
- [26] Richard Cole and Ofer Zajicek. The APRAM: incorporating asynchrony into the PRAM model. In *Proceedings of the 1st annual ACM symposium on Parallel algorithms and architectures (SPAA)*, pages 169–178, 1989.
- [27] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: streamlining stm by abolishing ownership records. *ACM SIGPLAN Notices*, 45(5):67–78, 2010.
- [28] David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Martin, Nir Shavit, and Guy L. Steele, Jr. Even better DCAS-based concurrent dequeues. In *Proceedings of the 14th International Conference on Distributed Computing (DISC)*, pages 59–73, 2000.
- [29] Dave Dice, Yossi Lev, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Simplifying concurrent algorithms by exploiting hardware TM. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 325–334, 2010.
- [30] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the*

- 14th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 157–168, 2009.
- [31] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 194–208, 2006.
- [32] Dave Dice and Nir Shavit. TLRW: return of the read-write lock. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 284–293, 2010.
- [33] David Dice, Alexander Matveev, and Nir Shavit. Implicit privatization using private transactions. In *5th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2010.
- [34] Stephan Diestelhorst and Michael Hohmuth. Hardware acceleration for lock-free data structures and software-transactional memory. In *Proceedings of the Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (EPHAM)*, 2008.
- [35] Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele, Jr. DCAS is not a silver bullet for nonblocking algorithm design. In *Proceedings of the 16th annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 216–224, 2004.
- [36] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a Research Toy. Technical Report LPD-REPORT-2009-003, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, 2009.
- [37] Joe Duffy. A (brief) retrospective on transactional memory. <http://www.bluebytesoftware.com/blog/>, January 2010.
- [38] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free step complexity: Lock-free dcas as an example. In *Proceedings of the 19th international symposium on Distributed Computing (DISC)*, pages 493–494, 2005.
- [39] Justin E. Gottschlich, Manish Vachharajani, and Jeremy G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization (CGO)*, pages 101–110, 2010.
- [40] Vincent Gramoli, Derin Harmanci, and Pascal Felber. Towards a theory of input acceptance for transactional memories. In *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS)*, pages 527–533, 2008.

-
- [41] Vincent Gramoli, Derin Harmanci, and Pascal Felber. On the input acceptance of transactional memory. *Parallel Processing Letters*, 20(1):31–50, 2010.
- [42] Michael Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, August 1999.
- [43] Michael Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *Proceedings of the 21st annual symposium on Principles of distributed computing (PODC)*, pages 260–269, 2002.
- [44] Rachid Guerraoui, Thomas A. Henzinger, Michal Kapalka, and Vasu Singh. Transactions in the jungle. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 263–272, 2010.
- [45] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Permissiveness in transactional memories. In *Proceedings of the 22nd international symposium on Distributed Computing (DISC)*, pages 305–319, 2008.
- [46] Rachid Guerraoui and Michał Kapalka. On obstruction-free transactions. In *Proceedings of the 20th annual symposium on Parallelism in algorithms and architectures (SPAA)*, pages 304–313, 2008.
- [47] Rachid Guerraoui and Michał Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 175–184, 2008.
- [48] Rachid Guerraoui and Michał Kapalka. The semantics of progress in lock-based transactional memory. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 404–415, 2009.
- [49] Phuong Hoai Ha, Philippas Tsigas, Mirjam Wattenhofer, and Roger Wattenhofer. Efficient multi-word locking using randomization. In *Proceedings of the 24th annual ACM symposium on Principles of distributed computing (PODC)*, pages 249–257, 2005.
- [50] Tim Harris and Keir Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, 2003.
- [51] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC)*, pages 300–314, 2001.

- [52] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing (DISC)*, pages 265–279, 2002.
- [53] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.
- [54] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529, 2003.
- [55] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual ACM symposium on Principles of distributed computing (PODC)*, pages 92–101, 2003.
- [56] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *ACM SIGARCH Computer Architecture News*, 21(2):289–300, 1993.
- [57] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
- [58] Maurice Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
- [59] IBM. *IBM System/370 Extended Architecture, Principle of Operation*, 1983. IBM Publication No. SA22-7085.
- [60] Damien Imbs, José Ramon de Mendivil, and Michel Raynal. Virtual world consistency: a new condition for STM systems (brief announcement). In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 280–281, 2009.
- [61] Damien Imbs and Michel Raynal. A lock-based STM protocol that satisfies opacity and progressiveness. In *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS)*, pages 226–245, 2008.
- [62] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th annual ACM symposium on Principles of Distributed Computing (PODC)*, pages 151–160, 1994.

-
- [63] Amos Israeli and Asaf Shirazi. The time complexity of updating snapshot memories. *Information Processing Letters*, 65(1):33–40, 1998.
- [64] Michal Kapalka. *Theory of Transactional Memory*. PhD thesis, EPFL, 2010.
- [65] Idit Keidar and Dmitri Perelman. On avoiding spare aborts in transactional memory. In *Proceedings of the 21st annual symposium on Parallelism in algorithms and architectures (SPAA)*, pages 59–68, 2009.
- [66] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.
- [67] Yossi Lev, Victor Luchangco, Virendra J. Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a scalable software transactional memory. In *4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2009.
- [68] Geoff Lowney. Why Intel is designing multi-core processors. In *Proceedings of the 18th annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 113–113, 2006.
- [69] Shiyong Lu, Arthur Bernstein, and Philip Lewis. Correct execution of transactions at different isolation levels. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1070–1081, 2004.
- [70] Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *Proceedings of the 2008 37th International Conference on Parallel Processing (ICPP)*, pages 67–74, 2008.
- [71] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):17, 2006.
- [72] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for java stm. In *Proceedings of the 20th annual symposium on Parallelism in algorithms and architectures (SPAA)*, pages 314–325, 2008.
- [73] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th annual symposium on Parallelism in algorithms and architectures (SPAA)*, pages 73–82, 2002.
- [74] Maged M. Michael. CAS-based lock-free algorithm for shared dequeues. In *Proceedings of the 9th Euro-Par Conference on Parallel Processing*, pages 651–660, 2003.

- [75] Jaime H. Moreno. Chip-level integration: the new frontier for microprocessor architecture. In *Proceedings of the 18th annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 328–328, 2006.
- [76] Jeff Napper and Lorenzo Alvisi. Lock-free serializable transactions. Technical Report TR-05-04, The University of Texas at Austin, 2005.
- [77] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 365–375, 2007.
- [78] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.
- [79] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in STM. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing (PODC)*, pages 16–25, 2010.
- [80] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. *ACM SIGPLAN Notices*, 37(10):5–17, 2002.
- [81] Michel Raynal, Thia-Ki Gérard, and Mustaque Ahamad. From serializable to causal transactions for collaborative applications. Research Report RR-2802, INRIA, 1996.
- [82] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 284–298, 2006.
- [83] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2006.
- [84] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th annual ACM symposium on Parallel algorithms and architectures (SPAA)*, pages 221–228, 2007.
- [85] Torvald Riegel, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. From causal to z-linearizable transactional memory. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing (PODC)*, pages 340–341, 2007.
- [86] Daniel J. Rosenkrantz, Richard E. Stearns, and II Philip M. Lewis. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198, 1978.

-
- [87] Christopher J. Rossbach, Hany E. Ramadan, Owen S. Hofmann, Donald E. Porter, Bhandari Aditya, and Emmett Witchel. TxLinux and MetaTM: transactional memory and the operating system. *Communications of the ACM*, 51(9):83–91, 2008.
- [88] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th annual ACM symposium on Principles of distributed computing (PODC)*, pages 240–248, 2005.
- [89] Florian T. Schneider, Vijay Menon, Tatiana Shpeisman, and Ali-Reza Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. *ACM SIGPLAN Notices*, 43(10):181–194, 2008.
- [90] Johannes Schneider and Roger Wattenhofer. Bounds On Contention Management Algorithms. In *Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC)*, pages 441–451, 2009.
- [91] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
- [92] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. *ACM SIGPLAN Notices*, 42(6):78–88, 2007.
- [93] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. Ordering-based semantics for software transactional memory. In *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS)*, pages 275–294, 2008.
- [94] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. Technical Report Tr 915, Department of Computer Science, University of Rochester, 2007.
- [95] Michael F. Spear, Maged M. Michael, and Christoph von Praun. RingSTM: scalable transactions with a single atomic instruction. In *Proceedings of the 20th annual symposium on Parallelism in algorithms and architectures (SPAA)*, pages 275–284, 2008.
- [96] H. Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology, 2004.
- [97] H. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based dequeues using single-word compare-and-swap. In *OPODIS 2004*, pages 240–255.

- [98] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS)*, pages 212–222, 1992.
- [99] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 34–48, 2007.
- [100] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.
- [101] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *Proceedings of the 20th annual symposium on Parallelism in algorithms and architectures (SPAA)*, pages 265–274, 2008.

הנגזרים. עקב כך, נדרשים מימושים התפורים למבנה נתונים מסוים. חיבור זה מציג גישה חדשה העושה שימוש בצביעה של האיברים במבנה הנתונים; בגישה זו האלגוריתם מבצע צעדים בדומה למתודולוגיה גנרית, אך יחד עם זאת, מודע לסמנטיקה של הפעולות של מבנה הנתונים הממומש. בסכמה המוצגת הצבעים מוטמעים באיברים, והמימוש משמר צביעה חוקית של האיברים בזמן ביצוע הפעולות על מבנה הנתונים, כלומר שני איברים סמוכים במבנה לעולם צבועים בצבעים שונים. אנו מדגימים את הגישה באמצעות אלגוריתמים חסרי מנעולים למבני נתונים מבוססי רשימה כגון תור דוראשי (deque), תור עדיפויות (priority queue), ורשימה מקושרת דו-כיוונית.

האלגוריתמים שהצגנו מפחיתים את ההתנגשויות בין הפעולות ובכך מגדילים את התפוקה של הביצוע. בפרט, באלגוריתם לרשימות מקושרות רק פעולות על איברים סמוכים ברשימה גורמות הפרעה זו לזו; במימוש של מנגנון סנכרון מרובה-מילים, הפעולות מבודדות מפעולות אחרות שאינם ברדיוס קרוב, כלומר פעולות רחוקות מתקדמות במקביל. החיבור מציע כלים שמאפשרים למדוד את רדיוס ההפרעה של הפעולות ולהוכיח את נכונות האלגוריתמים.

מטרתו של זיכרון עסקאות היא לפשט את התכנון של מערכות מקביליות תוך שמירה על נכונות התוכנית כמו גם שיפור הביצועים ביחס לקוד סדרתי על-ידי ניצול הזדמנויות המדרגיות שמציעות מערכות מרובות ליבה. עם זאת, ברב המקרים, מימושים של זיכרון עסקאות בעלי גרעיניות גסה אינם בעלי מדרגיות כמצופה. לעומתם, מימושים בעלי גרעיניות עדינה כרוכים ברמת סיבוכיות נוספת שאינה מבוטלת. החיבור בוחן האם ניתן לממש את ההבטחה של זכרון העסקאות, כלומר האם קיים מימוש שהוא פשוט, נכון, ובעל מדרגיות גבוהה.

בחיבור זה אנו מוכיחים מגבלות אינהרנטיות של זיכרון עסקאות ומצביעים על בחירות חשובות בתכנונו. הוכחנו שלא קיים מימוש של זכרון עסקאות שנמנע מהתנגשויות של טרנזאקציות על פריטי מידע זרים, ובנוסף מקיים שטרנזאקציות שמבצעות רק גישות קריאה תמיד מסתיימות בהצלחה בלי לכתוב לזכרון. למעשה, אנו מוכיחים שבמימושים אלו טרנזאקציות שמבצעות רק גישות קריאה חייבות לכתוב למספר מקומות בזיכרון, שהוא לינארי במספר פריטי המידע שנקראו.

טרנזאקציות מוגדרות לרוב כרשימה של פעולות שיש לבצע באופן אטומי. במקרים רבים, הרשימה מכילה פעולות שלא ניתנות לשחזור לאחור. כתוצאה מכך, זכרון העסקאות מחויב לתמוך בגישה לאותם פריטי מידע מתוך ומחוץ לטרנזאקציות; לתכונה זו חשיבות מכרעת לצורך יכולת פעולה הדדית עם קוד קיים, ועל מנת לשפר את הביצועים של זכרון העסקאות. הפרטה (privatization) מאפשרת למתכנת להפריט מספר פריטי מידע על-ידי בידודם, ובכך להפוך אותם לפרטיים לתהליך מסוים; לאחר מכן, התהליך יכול לגשת לפריטי מידע אלה שלא מתוך טרנזאקציה וללא הפרעה מתהליכים אחרים. אנו בוחנים את הסיבוכיות התיאורתית של הפרטה, ומראים מחיר אינהרנטי לינארי במספר הפריטים המופרטים. החסם התחתון נשען על הנחות שמלמדות שהגבלה של רמת המקביליות של זכרון העסקאות, או מעקב אחרי הפריטים אליהם ניגשות טרנזאקציות אחרות, הוא המחיר של הפרטה יעילה.

עקב מדיניות רכישת המנעולים, הצורך בהימנעות מחבק, והקושי בניפוי שגיאות. במקרים רבים, מתכנתים שאינם מומחים, חסרים את המיומנויות הדרושות לבניית אפליקציות נכונות ויעילות אשר רותמות את הכוח החישובי של ארכיטקטורות מרובות-מעבדים בצורה מיטבית.

מימושים חסרי-מנעולים עוקפים חלק מהבעיות המוזכרות, ובפרט אינם נחסמים. עם זאת, תכנונם הינו בהרבה מקרים מסובך וקשה כמו זה של מימושים מבוססי מנעולים בעלי גרעיניות עדינה.

זכרון עסקאות (Transactional memory) הינה גישה פופולרית להקלת הקושי בתכנות מקבילי. בהשראת טרנזאקציות ממסדי נתונים קלאסיים, גישה זו מאפשרת תרגום שיטתי של מבני נתונים סדרתיים למימושים מקביליים נכונים, ובכך מאפשרת גמישות בתכנון של אפליקציות מקביליות. טרנזאקציה בזכרון עסקאות עוטפת סדרה של פעולות; מובטח שאם פעולה אחת מתבצעת אזי כל הפעולות מתבצעות, ועבור תהליכים אחרים הביצוע של הפעולות נראה אטומי. מימוש בתוכנה של זכרון עסקאות מתרגם גישה לפריטי מידע ברמה גבוהה, לפעולות בסיסיות על מקומות בזכרון המכילים את הנתונים הדרושים עבור המימוש.

במטרה להשוות את רמת הקושי של תכנות מקבילי לזו של מימוש סדרתי, גישת זכרון העסקאות מסתירה מנגנוני סנכרון מתוחכמים תחת מעטה פשוט. המתכנת נדרש אך לסמן את גבולות ההתחלה והסיום של הטרנזאקציה, ומימוש זכרון העסקאות אחראי לבצע את הקוד המסומן, באופן שידמה כי כל טרנזאקציה מתבצעת באופן אטומי.

חיבור זה חוקר מימושים של מבני נתונים מקביליים, בדגש על מימושים חסרי מנעולים, ונגיעה גם במימושים מבוססי מנעולים בעלי גרעיניות עדינה. המטרה העיקרית של החיבור היא לספק מתודות המקלות על מימוש מבני נתונים מקביליים, ולהציע כלים להוכחת נכונותם, וכלים לניבוי יכולת המדרגיות שלהם בעזרת הערכת המקומיות (locality) שלהם. בנוסף המחקר עוסק במגבלות אינהרנטיות לתכנות מקבילי, בפרט בהקשר של זכרון עסקאות. הפרדיגמה של זכרון עסקאות העלתה את רף הציפיות בהבטחה לאפשר שליטה בקושי של תכנות מקבילי. לכן, יש חשיבות מרובה להבין את יחסי הגומלין והמגבלות האינהרנטיות במימוש של זכרון עסקאות.

מנגנוני סנכרון מרובי-מילים (multi-word synchronization) מקלים את התכנון והמימוש חסר המנעולים של מבני נתונים מקביליים המימוש שמתקבל הינו קל ויעיל יותר ממימושים הנסמכים על מנגנוני סנכרון הניגשים למילה בודדת. לדוגמא, מנגנון הסנכרון קריאה-שינוי-כתיבה הניגש למספר מקומות (kRMW), מאפשר קריאה של מספר פריטי מידע, חישוב של ערכים חדשים, וכתיבתם בחזרה לפריטי המידע, בפעולה אטומית אחת. חיבור זה מציג אלגוריתם לא חוסם למנגנון סנכרון מרובה מילים. האלגוריתם אינו מקבע מראש את מספר פריטי המידע אליהם ניתן לגשת בפעולה, והוא אף ניתן לשינוי כך שיקבל את פריטי המידע באופן דינמי בזה אחר זה. כמו כן, כמות האינפורמציה הנוספת, הנדרשת על מנת לתמוך במימוש, עבור כל פריט מידע הינה קבועה.

שימוש במתודולוגיות גנריות מאפשר לגזור ביעילות מימושים מקביליים למבני נתונים. אולם, שימוש בסכמות אלו כקופסא שחורה יוצר עלות גבוהה בביצועים של המימושים

תקציר

תדירות השעון כמעט ואינה עולה עוד בשנים האחרונות ושיפור בביצועים של יחידות חישוביות אינו מסתמך עוד עד עלייה במהירות השעון. במקום זאת, מרכז הכובד של היצרנים העיקריים של רכיבי המעבדים עובר מהגברת המהירות של מעבד בודד לאכלוס של יותר ויותר ליבות על גבי שבב בודד. הטכנולוגיה של ריבוי ליבות מתיחסת למעבד עם יותר מיחידה חישובית אחת, המאפשר הגברת היעילות, כיוון שעומס העבודה מתחלק בין היחידות. מעבר לטווח הכולל שרתים ומכונות קצה, ריבוי הליבות נפוץ במחשבים אישיים כמו גם במחשבים ניידים, והוא בדרכו לטלפונים ניידים, והתקנים קטנים נוספים.

ככל שריבוי ליבות וריבוי תהליכים נעשה נפוץ יותר, אפליקציות מודרניות נדרשות, לצורך חישוביהם, להשתמש במבני נתונים מקביליים, שהם הנושא המרכזי של חיבור זה. מבני נתונים מקביליים נתונים לגישה בריזומנית של מספר תהליכים הרצים על ליבות שונות. דוגמאות חשובות הן רשימות מקושרות דו-כיווניות, המשמשות לבניית מבני נתונים כגון מחסנית, תור, טבלאות ערבול, ורשימות דילוג; עצי-b המשמשים לרוב כאינדקס המידע במסדי נתונים; ועצי AVL—עצים בינאריים מאוזנים—המאפשרים ביצוע יעיל של חיפוש.

תכנון מבנה נתונים מקבילי והבטחת נכונותו הינה משימה קשה, המאתגרת יותר מתכנון מקבילו הסדרתי של מבנה הנתונים. הקושי, שנובע מהמקביליות, מחמיר עקב העובדה שהתהליכים הינם א-סינכרוניים, ונתונים לפסיקות. על מנת להתמודד עם הקושי שבתכנות מקבילי, אפליקציות מרובות תהליכים דורשות פונקציות סינכרון. מנגנונים אלו מבטיחים את נכונות האלגוריתם על-ידי תיאום הגישות המקביליות של התהליכים. בד בבד, יש חשיבות רבה לאפשר למספר רב של פעולות להתקדם במקביל ולהסתיים ללא הפרעה, על מנת לנצל את יכולות החישוב המקביליות של ארכיטקטורות מודרניות.

המטרה המרכזית של חיבור זה היא לצייד מתכנתים של מערכות מקביליות בכלים, שיקלו על כתיבה, הבנה, ותחזוק של מבני נתונים מקביליים, ואפליקציות מקביליות ככלל.

באופן מסורתי נעשה שימוש במנעולים על מנת להבטיח מניעה הדדית באפליקציות עם מספר תהליכים, לשיטה זו יש מספר חסרונות לא מבוטלים. מימושים מבוססי מנעולים נחסמים, כאשר תהליך אחד מנוע מלהתקדם בעת שהוא ממתין שתהליך אחר ישחרר מנעול. מימושים מבוססי מנעולים בעלי גרעיניות גסה (coarse-grained) הינם פשוטים יחסית אך מצמצמים את היכולת לתמיכה במדרגיות (scalability). לעומתם, מימושים מבוססי מנעולים בעלי גרעיניות עדינה (fine-grained) מסובכים יותר, בין השאר

המחקר נעשה בהנחיית פרופ' חגית עטיה בפקולטה למדעי המחשב.

ברצוני להודות לחגית עטיה על היותה מורת דרך לאורך שנים אלו. התמיכה המקצועית והאישית, הזמן שהקדישה והסובלנות, ההדרכה והנגישות לידע העצום שלה הינם יקרי ערך עבורי. למדתי ממנה רבות, והיא תמיד תהווה עבורי מודל למצוינות.

אני מודה לאלסיה מילאני על שיתוף הפעולה והחברות; אני מודה לדויד חי וקרן צנזור-הלל, אחיי האקדמאיים, על העצות הטובות והעזרה בכל עת שנזקקתי להם.

אני מודה לחברי וועדת הבוחנים שלי: פרופ' מוריס הרליה, פרופ' עדית קידר ופרופ' ארז פטרנק, כמו גם לפרופ' ניר שביט, פרופ' יהודה אפק, פרופ' עודד שמואלי, ופרופ' אסף שוסטר על הערות והצעות מועילות.

פרופ' פיית' אלן, פרופ' יולה פטורו, וינסנט גרמולי, פרופ' רשיד גראווי, דני הנדלר, מיכל קפלוקה, אלכס קוגן, פטר קוזנצוב, אלכס שרייר, פרופ' מייקל ספיר, פרופ' גדי טאובנפלד, ומרטין ווצ'ב נתנו הערות מועילות על גרסאות קודמות של התוצאות המובאות בחיבור זה.

חיבור זה מוקדש לדובי הלל, שתמיד מעודד אותי לכוון גבוה ולשגשג, וגם מזכיר לי להנות מהמסע בדרך להגשמה.

אהבתי שלוחה לילדיי, העונג שבלצפות בהם גדלים מאפשר לכל לקרות עם חיוך. תודות מיוחדות להורי, אחיותיי ומשפחתי המורחבת, על היותם מקור השראה. מעל לכל אני מודה להם על היותם לי קן.

אני מודה לטכניון, לקרן הישראלית למדע, ולקרנות גוטווירט וזף על התמיכה הכספית הנדיבה בהשתלמותי.

מבני נתונים מקביליים: מתודולוגיות ומגבלות אינהרנטיות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

אשכר הלל

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
אדר א' ה'תשע"א חיפה פברואר 2011

מבני נתונים מקביליים: מתודולוגיות ומגבלות אינהרנטיות

אשכר הלל