Scalable Consistency in the Multi-core Era

Thesis approved by
the Department of Computer Science
of the Technical University of Kaiserslautern

for the award of the Doctoral Degree
Doktor der Naturwissenschaften (Dr.rer.nat)
to

Deepthi Devaki Akkoorath

Date of oral defense :	March 26, 2019
Dean :	Dr. habil. Bernd Schürmann
PhD committee	
Chair :	Prof. Dr. Ralf Hinze
Reviewers :	Prof. Dr. Arnd Poetzsch-Heffter
	Prof. Paulo Sérgio Almeida

D 386

Abstract

The advent of heterogeneous many-core systems has increased the spectrum of achievable performance from multi-threaded programming. As the processor components become more distributed, the cost of synchronization and communication needed to access the shared resources increases. Concurrent linearizable access to shared objects can be prohibitively expensive in a high contention workload. Though there are various mechanisms (e.g., lock-free data structures) to circumvent the synchronization overhead in linearizable objects, it still incurs performance overhead for many concurrent data types. Moreover, many applications do not require linearizable objects and apply ad-hoc techniques to eliminate synchronous atomic updates.

In this thesis, we propose the **Global-Local View Model**. This programming model exploits the heterogeneous access latencies in many-core systems. In this model, each thread maintains different views on the shared object: a thread-local view and a global view. As the thread-local view is not shared, it can be updated without incurring synchronization costs. The local updates become visible to other threads only after the thread-local view is *merge*d with the global view. This scheme improves the performance at the expense of linearizability.

Besides the *weak* operations on the local view, the model also allows *strong* operations on the global view. Combining operations on the global and the local views, we can build data types with customizable consistency semantics on the spectrum between sequential and purely mergeable data types. Thus the model provides a framework that captures the semantics of **Multi-View Data Types**. We discuss a formal operational semantics of the model. We also introduce a verification method to verify the correctness of the implementation of several multi-view data types.

Frequently, applications require updating shared objects in an "all-or-nothing" manner. Therefore, the mechanisms to synchronize access to individual objects are not sufficient. Software Transactional Memory (STM) is a mechanism that helps the programmer to correctly synchronize access to multiple mutable shared data by serializing the transactional reads and writes. But under high contention, serializable transactions incur frequent aborts and limit parallelism, which can lead to severe performance degradation.

Mergeable Transactional Memory (MTM), proposed in this thesis, allows accessing multi-view data types within a transaction. Instead of aborting and re-executing the transaction, MTM merges its changes using the data-type specific merge semantics. Thus it provides a consistency semantics that allows for more scalability even under contention. The evaluation of our prototype implementation in Haskell shows that mergeable transactions outperform serializable transactions even under low contention while providing a structured and type-safe interface.

Acknowledgments

First and foremost, I can't thank Dr. Annette Bieniusa enough for guiding me through out the research and supporting me when I most needed. With out her help and support, this thesis would not have become the way it is now.

I am so grateful to Prof. Dr. Arnd Poetzsch-Heffter for his continuous support through out the work. I also thank Prof. Paolo Sérgio for being the co-reviewer. The interest you showed in the topic was a great encouragement for me. Thank you Prof. Carlos Baquero for the brain storming sessions that resulted in one of our co-authored paper.

My time at university would not have been so memorable with out the amazing people I have met there. Many thanks to Peter Zeller, Mathias Weber, Ilham Kurnia, Malte Brunnlieb, and Sebastian Schweizer for being great colleagues. Thank you Marc Shapiro for introducing me to SyncFree project that finally lead me to take up this PhD position and able to work with some great people: Nuno Preguiça, Valter Balegas, Carla Ferreira, Gonçalo Tomás and João Leitão. I had a great time hacking AtindoteDB with Manuel Bravo, Alejandro Tomsic, Zhongmiao Li and Christopher Meiklejohn. Thank you José Brandão for working together with me to implement the prototype.

My research was supported by EU H2020 LightKone project (732505) and European FP7 project SyncFree (609551). These projects gave me the opportunity to work with several expert researchers from around the world.

I would like to thank my friends and family. Thank you Sajith and Vidya for supporting me from the other side of the world.

Finally, thank you Vipin for pushing me to reach my goals.

Contents

List of Figures

List of Algorithms

Chapter 1
Introduction

For years, applications were observing regular performance gains owing to the steady increase in processor clock speed. The manufacturers have hit the physical limits of improving single processor performance; instead, they gravitate towards hyperthreading and multicore processors [1, 4]. Today, multicores are everywhere - from mobile phones, laptops, high-end servers, and special purpose processors. This trend is so pervasive that it is near impossible to find a single-core processor nowadays. With the availability of 100s of cores in a processor, the primary focus of application design now is to exploit parallelism: utilizing parallel threads running on multiple processors to accomplish a single task.

Concurrent programming aids programmers to improve speed-up of the applications by exploiting multicore processors efficiently. However, more threads do not imply unconditionally better performance always. Amdahl's law [13, 76] states that the proportion of sequential execution limits the speed-up of the application. Thus a significant effort in parallelizing an application must be to reduce the amount of such non-parallelizable program segments.

A shared-memory concurrent program consists of a set of processes communicating via shared memory by manipulating a set of shared objects. Unlike in a single-threaded program, we need to guarantee *safety* when there are concurrent accesses to a shared object. Linearizability [55] has turned out to be a fundamental notion on simplifying the reasoning about the correctness of shared objects for programmers. This consistency model formalizes the notion of atomicity for high-level operations. In an execution, every method call is associated with a linearization point, a point in time between its invocation and its response. The call appears to occur instantaneously at its linearization point, behaving as specified by its sequential definition.

Linearizability is the strongest as well as the most used correctness condition because it is composable and easy to reason about the correctness of the application owing to its closeness to the sequential specification of the data structure. Several concurrent data structure design methodologies [71] attempt to improve the scalability of concurrent access while maintaining linearizability. These designs use various synchronization mechanisms such as locks, compare-and-swap, custom lock-free algorithms, and transactional memory to protect the shared locations during concurrent access. The coordination needed for the synchronization contribute to the non-parallelizable segment of the execution [37]. As Amdahl's law recommends, we must direct our efforts to reduce this coordination to extract as much parallelism as possible.

The strict ordering requirement of linearizability hampers the possibility of reducing the synchronization. With the rapid growth of the number of cores and the adoption of NUMA architectures with heterogeneous access latencies, the communication and synchronization cost between the components keeps growing. Thus, restricting to the sequential specification as in linearizability will reach the limits of scalability of concurrent data structure designs.

In practice, programming patterns are emerging that attempt to limit the associated cost of the necessary synchronization on the memory accesses. For example, in the widely-used messaging library ZeroMQ, adding messages to the queue is at the core of the application. While lock-free linearizable queues are fast, the developers observed that enqueuing each new message atomically was affecting the overall performance, especially in high contention workloads [85]. However, only the relative order of messages from a single thread is relevant for the semantics of the message queue; it is not necessary to maintain a strict order of enqueue operations when two independent threads try to insert messages concurrently into the queue. To overcome the performance penalty, the developers re-engineered their message queue such that multiple messages are added as a batch, using only one single atomic operation.

For another example, consider a shared counter that is concurrently updated by several threads. The final value of the counter must include all increments performed, but the order of increments is not relevant since all increments are commutative. If each increment executed by each thread is an atomic operation made visible to all other threads, it can become a bottleneck limiting the performance of the program [19]. In many cases, it is sufficient to execute the increment on some thread-local variable and to apply a combined update to the shared object.

Thus the trends of processor architectures, as well as the requirement for many applications, indicate the need for better scalable concurrent object design methodologies. As a result, many new relaxed objects semantics [81] that favor better scalability of concurrent data structures are widely studied. These models, in general, allow flexible reordering of operations that deviate from linearizability thus enabling better optimizations on the data structure designs.

There are similar concerns in distributed systems, as the network latency between the servers results in higher cost of synchronization. These concerns led to the development of conflict-free replicated data types (CRDTs) [80, 79]. CRDTs are data types with specific semantics that allow concurrent updates on different replicas to execute without any coordination. The updates are propagated to other replicas asynchronously. The properties of CRDTs ensure that all updates are incorporated resulting in a consistent state across all replicas. This property is called Strong Eventual Consistency [79], which is now a widely accepted alternate semantics to linearizability for replicated objects.

CRDTs allow development of replicated systems that limit global synchronization and operate locally when possible thus achieving near linear scalability. At the same time, the multicore architecture is advancing towards a more distributed architecture, where the distance between resources is heterogeneous. The natural question that follows is whether we can apply the synchronization-free techniques from distributed systems in the design of shared-memory concurrent objects.

This thesis explores a novel model for concurrent objects in shared-memory that exploits a relaxed semantics to achieve scalability. The Global-Local View Model exploits the heterogeneous access latencies in many-core systems. In this model, each thread maintains different views on the shared object: a thread-local view and a global view. As the thread-local view is not shared, it can be updated without incurring synchronization costs. The local updates become visible to other threads only after a thread merges its thread-local view with the global view. This scheme improves the performance at the expense of linearizability. Besides the *weak* operations on the local view, the model also allows *strong* operations on the global view. Combining operations on the global and the local views, we can build data types with customizable consistency semantics on the spectrum between sequential and purely mergeable data types. Thus, the model provides a framework that captures the semantics of *Multi-View Data Types.*

Linearizability and other relaxed semantics we have discussed so far is used to reason about the correctness of individual objects. In many cases, applications require updating multiple shared objects in an “all-or-nothing” manner. Therefore, the mechanisms to synchronize access to individual objects are not sufficient. Transactional Memory [45] is a mechanism that helps the programmer to synchronize access to multiple mutable shared data correctly. Akin to linearizability of individual objects, the correctness condition that is often applied to TM is serializability. Serializability [78] guarantees that two concurrent transactions behave as if they are executed sequentially one after the other. An optimistic algorithm executes the operations in a transaction without coordinating with others and buffers the modifications locally. During the commit, it checks for conflicting operations, which are concurrent updates that modified the same objects that it has accessed. In case of conflicts, the transaction is aborted and re-executed. Even with transactions with relaxed semantics, conflicts and hence aborts can still arise when updates cannot be serialized leading to degradation of performance.

We propose a novel semantics for Transactional Memory exploiting the "mergeable" semantics of Multi-view Data Types. Instead of aborting and re-executing the transaction, Mergeable Transactional Memory merges its changes using the data-type specific merge semantics. It provides a relaxed consistency semantics that allows for more scalability even under contention.

1.1 Contribution of this thesis

In this thesis, we study how relaxing the strong consistency requirement (linearizability) of concurrent objects helps to achieve better scalability in concurrent shared-memory programs. As a result, the contributions of this thesis are:

- **Global-Local View Model**, a novel model that serves as a base framework to design concurrent objects with relaxed semantics.
- **Multi-view Data Types** (MDT), a set of concurrent data types designed based on the Global-Local View Model.
- A proof method to verify the correctness of data type implementations.
- **Mergeable Transactional Memory** (MTM), a Software Transactional Memory that allows better scalability than serializability leveraging the properties of Multi-view Data Types.
- An evaluation that shows the scalability of the proposed MDT designs and MTM.

1.2 List of publications

Publications directly contributed to the thesis:

- Deepthi Devaki Akkoorath and Annette Bieniusa. Transactions on mergeable objects. In *Programming Languages and Systems - 13th*

Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings, pages 427–444, 2015.

- Deepthi Devaki Akkoorath and Annette Bieniusa. Highly-scalable concurrent objects. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016*, pages 13:1–13:4, 2016.

- Deepthi Devaki Akkoorath, José Brandão, Annette Bieniusa, and Carlos Baquero. Multi-view data types for scalable concurrency in the multi-core era. In *Proceedings of the Workshop on Programming Models and Languages for Distributed Computing*, PMLDC '17, New York, NY, USA, 2017. ACM.

- Deepthi Devaki Akkoorath, José Brandão, Annette Bieniusa, and Carlos Baquero. Global-local view: Scalable consistency for concurrent data types. In *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, pages 492–504, 2018.

Other relevant papers published during the PhD:

- Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*, pages 405–414, 2016.

- Deepthi Devaki Akkoorath, Viktória Fördős, and Annette Bieniusa. Observing the consistency of distributed systems. In *Proceedings of the 15th International Workshop on Erlang, Nara, Japan, September 18-22, 2016*, pages 54–55, 2016.

- Gonçalo Tomás, Peter Zeller, Valter Balegas, Deepthi Devaki Akkoorath, Annette Bieniusa, João Leitão, and Nuno M. Preguiça. Fmke: a real-world benchmark for key-value data stores. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2017, Belgrade, Serbia, April 23 - 26, 2017*, pages 7:1–7:4, 2017.

1.3 Overview

In Chapter 3, we describe the global-local view model and present the operational semantics of an abstract execution model. We discuss the design of several multi-view data types and outline the correctness proof. We present the novel mergeable transactional memory in Chapter 4. We discuss the operational semantics of MTM and provides an algorithm. We also discuss the implementation of MTM in haskell defining a structured and type-safe interface. In Chapter 5, we present several experiments to evaluate the performance and scalability of MDT and the global-local view model. Further, Chapter 6 discusses several related works. We conclude the thesis in Chapter 7.

Chapter 2

Concurrency, Consistency and Scalability

Concurrent programming has reached all programmers and programs. It is inevitable when providing responsive GUIs, non-blocking HW drivers, and parallelizing network connections. Further, the advent of multi-core systems has made concurrent programming wide-spread in order to utilize additional computing capacity. The benefits of concurrent programming are shadowed by the difficulties that arise to reason about the correctness of concurrent programs, in particular when managing shared memory access. Classical lock-based synchronization relies on the programmer's expertise to avoid deadlock and livelocks, in particular when access to multiple shared objects is required simultaneously. Transactional Memory (TM) [45] provides an abstraction for concurrent access to multiple objects, akin to transactions in a database systems. Several consistency models have been defined to reason about the correctness of shared objects in the presence of concurrent access.

In this chapter, we discuss the commonly used correctness conditions of concurrent objects and transactional memory such as linearizability, serializability and snapshot isolation. These semantics are categorized as *strong consistency* because they have strict restrictions on the allowed behavior of a concurrent object. We also discuss weaker consistency semantics named as Eventual Consistency, widely used in distributed systems to achieve better scalability.

2.1 System model

We assume a system of n threads independently executing at arbitrary speeds. Each thread executes a single sequential program. Multiple threads can execute concurrently. Communication between the threads is done exclusively via shared memory objects; we assume that there are no side channels.

2.2 Shared mutable objects

A shared object has a state and supports a set of operations to read or update its state. Concurrent threads can read and update the shared object using these operations. However, to ensure that concurrent access does not lead to incorrect state, the threads may have to use additional synchronization mechanisms. A concurrent object is a shared object that guarantees that concurrent access to the object is “correct”. We explain in the next section what correctness means in this context.

2.2.1 Correctness conditions

A sequential specification of an object defines, given a particular state of the object, what will be the resulting state after executing an operation and its return value. Given an initial state, and a sequence of operations, we can determine the resulting state when the operations are executed in sequential order. Unfortunately, for a concurrent object, following the sequential specification may not give its correct behavior because multiple threads might be invoking operations on the object concurrently. Sequential specification needs to be adapted to cover concurrent, i.e. non-sequential, execution of operations.

To specify the correctness of the concurrent object, several consistency models have been defined. The consistency models are typically defined on a history of the concurrent system. A history consists of a sequence of method invocations ($inv(m, obj, t)$) and response events ($res(m, obj, t)$). In a concurrent history, method invocations and response from different threads may be interleaved. A history is sequential if every method invocation is immediately followed by its matching response event. A response matches an invocation if they have the same object and thread. A method call is a pair consisting of an invocation and the next matching response.

Sequential Consistency A history is sequentially consistent if it is equivalent to a history S, where 1) S is a sequential history, and 2) the method invocations in S follow the program order, i.e. the order in which they appear in the program.

Sequential consistency specifies that the result of the execution is the same as if the operations of all the processes were executed in some sequential order, and the operations of each individual process appear in the order specified by its program [64]. In sequential consistency, the ordering restrictions are only between the operations within a thread. The operations from different threads may occur in any order disregaring their real time order.

Linearizability A history H, is linearizable if it is equivalent to some sequential history S, and if method calls m_0 precedes method call m_1 in H, then the same is true in S, i.e., if one method call precedes another, then the earlier call must have taken effect before the later call. If two methods are concurrent, they can be ordered in any suitable way.

Unlike sequential consistency, linearizability considers the real-time order of the operation. Informally, every method call is associated with a linearization point, a point in time between its invocation and its response. The call appears to occur instantaneously at its linearization point.

Hardware vendors and most programming languages do no guarantee sequential consistency or linearizability. Traditional hardware and compilers designed for sequential programs perform a number of transformations, such as re-ordering memory read/write operations, to improve the performance of the program. In case of a multi-threaded program, the threads can observe this out-of-order execution. Hence, enforcing sequential consistency or linearizability restricts these transformations, thus significantly reducing the opportunities for performance optimizations. Besides, processors usually have multiple layers of cache to improve the speed of accessing objects by serving the access requests locally without contending the shared memory bus. The processors that provide strong consistency (e.g. sequential consistency) guarantee that all cached copies have the same value at any point in time. Other processors that provide weaker consistency models offer special memory barrier instructions to make the local updates visible to each other.

Programming languages also offer relaxed consistency for normal operations and provide special "synchronization" operations when stronger consistency is needed. For example, in the Java memory model, all threads have access to a common memory, where the shared objects are stored [41]. Besides, conceptually, each thread has its local memory area, where it caches objects. The thread is guaranteed to propagate the updates to cached objects to the common memory immediately only when an explicit synchronization operation happens. The synchronization operation has the effect of flushing the (hardware) cache to main memory and/or invalidating the cache so that the objects in main memory and local cache are up-to-date. Depending on the underlying processor architecture, these synchronization operations may execute a memory barrier operation in order to make the updates by the processors visible to each other.

In Java, there are different mechanism to coordinate access to shared data.

- Intrinsic locks [41]. Java provides a built-in locking mechanism via *synchronized* blocks implemented using *monitors*. A *synchronized* block specifies an object that will serve as the lock and a code fragment to be guarded by the lock. When a thread executes a synchronized block, the lock on the *monitor* associated with the object is automatically acquired and automatically released when exiting the block. The intrinsic locks act as mutual exclusion locks, thus executing the code within atomically.
- Explicit locks. Java provides explicit locking mechanism via the Lock interface and Reentrant locks. Reentrant locks provide similar mutual exclusion semantics as synchronized blocks.
- Volatile fields [41]. Java memory model guarantees that all accesses to volatile fields are totally ordered. However, a simple volatile variable cannot provide read-modify-write atomicity. Java provides *atomic* wrappers for the primitive types in `java.util.concurrent.atomic` that implements atomic compare-and-set in addition to the properties of the volatile fields.

Since linearizability must guarantee that the events are observed in a specific order, it demands some kind of synchronization for each operation. Using any of the above synchronization mechanisms would result in overhead due to the execution of memory barriers or flushing caches, thus affecting the performance of the application.

2.3 Transactional memory

A transaction is a sequence of actions that appears to execute instantaneously [45]. Transactions were initially introduced in databases to exploit parallelism of hardware with relatively simple programming model when executing database queries. Database transactions provides four properties known as ACID properties. Atomicity requires that either all actions in a transaction execute successfully or none of them execute. Consistency relates to the correctness of the state as a result of executing the operations. Isolation says how two concurrent transactions interfere with each other. The strongest isolation guarantee that they do no interfere with each other. Durability requires that the result of a committed transaction is persistently stored.

Inspired by the simplicity of transaction abstractions in database, Transactional Memory (TM) was introduced in concurrent programming to ensure the consistency of objects shared among multiple threads. With out TM, programmers has to use synchronization mechanisms such as locks, mutexes etc. to protect the shared object during concurrent access. These low level synchronization primitives are often error-prone and difficult to compose. Programming languages that support TM allow programmers to wrap a piece of code in

```
atomic {
  //code to read, modify, write shared objects
  read var
  ...
  write var
}
```

to execute the operations as a single unit of operation with out the need for explicitly acquiring locks. When two transactions concurrently access the same memory locations, with atleast one thread updating the location, the operations conflict. The TM implements various mechanisms to detect and resolve such conflicts.

A pessimistic TM prevents any conflicting access to the same data by acquiring locks for all objects that are accessed with in the transaction prior to its execution.

In an optimistic approach, multiple transactions can access an object concurrently. When the transaction commits, it checks for conflicts. In case of conflicts, it is aborted and re-executed; otherwise its modifications will be written to memory.

2.3.1 Correctness conditions

Akin to linearizability for concurrent objects, the basic correctness condition for concurrent transactions is serializability. Serializability requires that the result of executing a group of concurrent transactions must be equivalent to some serial execution of them [78]. With serializability, two transactions conflict if they access the same object and atleast one of them modifies the object. These conflicting transactions must be executed serially.

Snapshot Isolation [75] is a correctness condition weaker than serializability. In snapshot isolation, two operations conflict only if both of them modify the same object. Snapshot isolation requires that all operations in the transactions see a consistent snapshot, taken at the start of the transaction. A transaction is successfully committed when there are no write-write conflict. In STM, snapshot isolation is often implemented by maintaining multiple versions of the objects [26]. Multi-versioning is a way to increase the concurrency and performance by allowing transactions to read from several last versions of the object. Unlike serializability, in snapshot isolation, read-only transactions never conflict and thus improve the concurrency of application that have transactions with high read-write ratio.

The code snippet in Figure 2.1 demonstrates the difference between serializability and snapshot isolation. Snapshot Isolation is weaker than serializability as it allows more observable behavior than serializability.

Thread 1	Thread 2

`atomic{`
`  if(x == 0)`
`    y = 1;`
`}`
	`atomic{`
	`  if(y == 0)`
	`    x = 1;`
	`}`

Serializability:	x = 0, y = 1 or
	x = 1, y = 0.

Snapshot isolation:	x = 0, y = 1 or
	x = 1, y = 0 or
	x = 1, y = 1

Figure 2.1: Example: Expected behavior in Serializability and Snapshot Isolation.

2.4 Eventual consistency

The PACELC theorem [7], which is an extended form of CAP theorem [40], states that there is a trade-off between consistency and latency in a distributed system. In order to provide strong consistency in a replicated database, the system requires communication and coordination between participating servers that are located across the globe. The round-trip communication time as well as the delays due to network interruptions add to the latency perceived by the user for a single operation. Thus many modern databases [5, 2, 3] provide Eventual Consistency [87]. With eventual consistency, it is not guaranteed that an update to an object in one replica is immediately visible at others; but when all updates are delivered to all replicas (in any order), the replicas converge to the same state. Concurrent updates may conflict. The systems implement various conflict resolution mechanism; Conflict-free

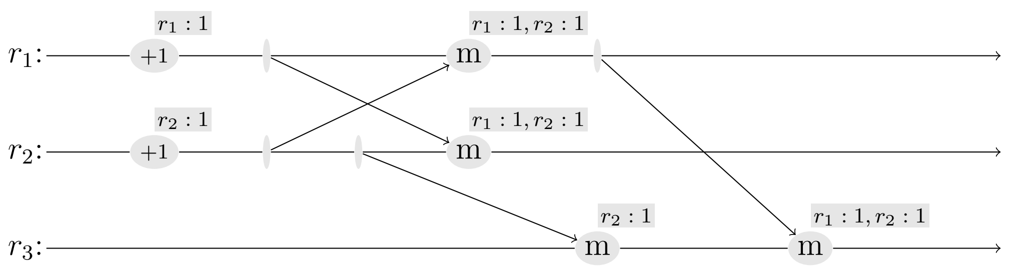


Figure 2.2: Evolution of the state of a CRDT counter in presence of concurrent updates. +1: increment, m: merge, r_i: replica i. The states shown in rectangle boxes converges on all replicas.

Replicated Data Types (CRDT) [80] being one of them.

Conflict-free Replicated Data Types are data types with simple mathematical properties that ensure eventual consistency. An update to a CRDT requires no synchronization, and converges to a same correct state. A state-based CRDT grows over a monotonic semi-lattice. Two states of the same objects are merged by taking the least upper bound in the semi-lattice. A replicated data store that uses state-based CRDTs executes an update on the CRDT object locally and store the resulting state; it eventually communicates the new state to other replicas asynchronously. When a replica receives a value from a remote replica, it merges the new value with the local value. Since the merge is commutative and transitive, it guarantees that both replicas converge to the same value when both of them receives all modified states in any order. For example, a state based CRDT counter consists of a map, where each replica id is associated with an integer representing the number of increments it executed. The value of the counter is the sum of all integers in the map. The merge takes the maximum value of each entry; thus the resulting states assume values from a monotonic semilattice. Figure 2.2 shows an example execution timeline.

Op-based CRDTs, on the other hand, exploit commutativity of updates to deterministically converge the states of two replicas. A replicated data store that uses op-based CRDTs requires reliable broadcast communication with a well-defined delivery order.

While eventual consistency is now well studied and a popular choice in distributed and replicated systems, similar relaxed consistency semantics are not well explored in the multi-core programs. This thesis proposes a model for designing and implementing CRDT like mergeable types for a shared memory concurrent program.

Chapter 3

Multi-view Data Types

Concurrent programming on shared-memory architectures is notoriously difficult. A concurrent system consists of a set of processes communicating implicitly through shared data structures. The visibility of updates on these data structures depends on the intricate interplay of synchronization mechanisms as defined by the memory model. Linearizability [55] has turned out to be a fundamental notion on simplifying the reasoning about the correctness of shared data structures for programmers. This consistency model formalizes the notion of atomicity for high-level operations. In an execution, every method call is associated with a linearization point, a point in time between its invocation and its response. The call appears to occur instantaneously at its linearization point, behaving as specified by the sequential semantics.

While linearizability is very useful for reasoning about the correctness of concurrent data structures, its implementation can be prohibitively expensive. As the number of cores increases in a multi-core system, the synchronization cost becomes more apparent that it favors the relaxation of the concurrent objects semantics for scaling the programs [81]. In practice, programming patterns are emerging that attempt to limit the associated cost of the required synchronization on the memory accesses. For example, in the widely-used messaging library ZeroMQ, adding messages to the queue is at the core of the application. While lock-free linearizable queues are fast, the developers observed that enqueuing new messages were affecting the overall performance, especially in high contention workloads [85]. However, only the relative order of messages from a single thread is relevant for the semantics of the message queue; it is not necessary to maintain a strict order of enqueue operations when two independent threads try to insert messages concurrently into the queue. To overcome the performance penalty, the developers re-engineered their message queue such that multiple messages are added as a batch, thus using only one single atomic operation.

For another example, consider a shared counter that is concurrently updated by several threads. The final value of the counter must include all increments performed, but the order of increments is not relevant since all increments are commutative. If each increment executed by each thread is an atomic operation made visible to all other threads, it can become a bottleneck limiting the performance of the program [19]. In many cases, it is sufficient to execute the increment on some thread-local variable and to apply a combined update to the shared object.

In this chapter, we introduce the Global-Local View Model that serves as a base framework for designing concurrent objects. The model leverages the fast local state and distant global state to allow concurrent operations to execute in parallel. We also introduce Multi-View Data Types (MDT) that are concurrent data types whose design is based on the Global-Local View Model.

Overview

- We begin with an informal discussion of the global-local view model in Section 3.1.
- In Section 3.2, we identify several applications that can benefit from the use of the global-local view model and MDTs.
- In Section 3.3, we discuss the specification of several MDTs.
- Section 3.4 gives a formal execution model and the operational semantics.
- In Section 3.5, we discuss the implementations of several MDTs and identify the properties of MDTs that provide guidelines on how to design other types.
- In Section 3.6, we discuss a proof method to validate the correctness of MDT implementations.

3.1 Global-Local View Model

The Global-Local View Model is a novel model for concurrent shared objects that leverages different views of an object. In this section, we give an informal description of the model and its properties.

In the global-local view model, an object has multiple copies of its own state, which are called *views*. The object has a *global view* accessible by all threads. Each thread sees a *local view* of the object which is not accessible by other threads. Such an object is called a *multi-view object*. Figure 3.1 shows the multiple views on a shared object accessed by concurrent threads.

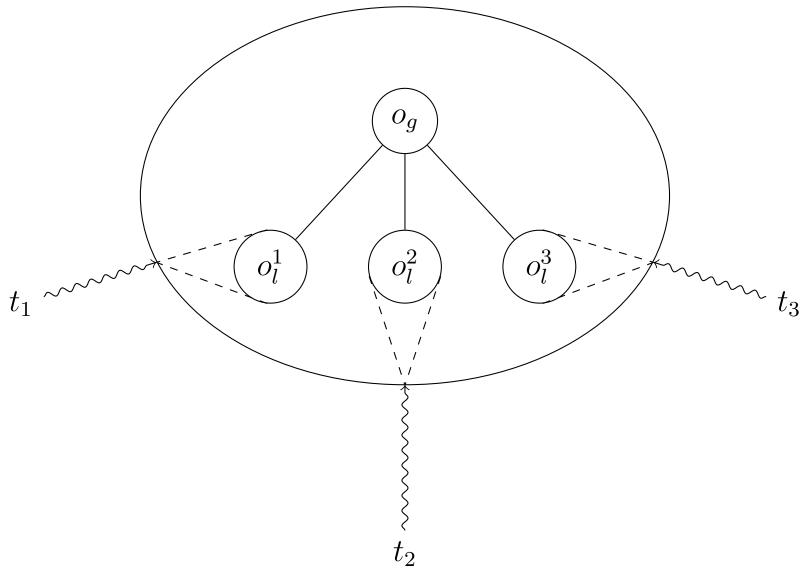


Figure 3.1: A multi-view object o, o_g is the global view, thread T_i sees its local view o_l^i.

In this model, an operation issued by a thread is executed on its local view without changing the global view or other local views. The local updates, though visible in the local view, are made visible on a global view only after the local view is merged to the global view. The other threads observe these changes once they synchronize their respective local view with the global view.

A merge must incorporate the local changes while not overriding the concurrent updates merged to the global view. For example, consider an integer that has a global view of value 0, and two local views of value 0 each (See Figure 3.2). Thread t_1 sets its local view to 1. Concurrently, thread t_2 also sets its local view to 1. Thread t_1 merges its local view to obtain a global view of value 1. After thread t_2 merges, the global view remains 1.

What should be the global view if the object type was a counter instead of an integer? Instead of setting to a specific value, a counter provides an increment operation. The result of executing two $set(1)$ operations are different from two $inc(1)$ operations. In Figure 3.3, threads t_1 and t_2 execute an increment resulting in local views of value 1. If the merge just overwrites the value, the global view will be 1 after t_2 merges. As it is a counter, it is meaningful if the merge results in value 2 because there were two increments executed in total. Hence, the merge must be a type-specific operation which can incorporate concurrent updates to the global view in a semantically meaningful way. In Section 3.3 and Section 3.5, we discuss the specification and

$$\begin{tikzcd}
t_1 : 0 & t_1 : 1 & t_1 : 1 & t_1 : 1 \\
t_2 : 0 \arrow[r, "t_1 : set(1)", "t_2 : set(1)"'] & t_2 : 1 \arrow[r, "t_1 : merge"] & t_2 : 1 \arrow[r, "t_2 : merge"'] & t_2 : 1 \\
g : 0 & g : 0 & g : \mathbf{1} & g : \mathbf{1}
\end{tikzcd}$$

Figure 3.2: Expected semantics of an **integer** in the global-local view model. t_i - local view of thread t_i. g - global view.

$$\begin{tikzcd}
t_1 : 0 & t_1 : 1 & t_1 : 1 & t_1 : 1 \\
t_2 : 0 \arrow[r, "t_1 : inc(1)", "t_2 : inc(1)"'] & t_2 : 1 \arrow[r, "t_1 : merge"] & t_2 : 1 \arrow[r, "t_2 : merge"'] & t_2 : 2 \\
g : 0 & g : 0 & g : \mathbf{1} & g : \mathbf{2}
\end{tikzcd}$$

Figure 3.3: Expected semantics of a **counter** in the global-local view model. t_i - local view of thread t_i. g - global view.

the implementation of several types.

As the local view is private, the operations on it can be performed without any coordination among the threads. Coordination is required only when the global view is accessed. Threads can execute many local updates without synchronizing with the global view or coordination with other threads, thus enabling better performance, albeit at the expense of linearizability [55].

In addition to the local operations, the model also provides synchronous operations on the global view. Consider, for example, a queue where the enqueue operations have been executed on the local view. To guarantee that the elements are dequeued only once, dequeues are executed atomically on the global view. We call the operations that perform only on local view *weak* operations and those on global view *strong* operations. Combining operations on the global and the local views, we can build data types with customizable semantics on the spectrum between sequential and purely mergeable data types.

List of operations. The global-local view model defines the following operations that are executed on a multi-view object.

- pull updates the local view of the executing thread with the global view.
- weakRead q returns the result of a type-specific query q on the local view.
- strongRead q returns the result of a type-specific query q on the global view.

- weakUpdate u applies an update operation u on the local view without any coordination.

- strongUpdate u applies an update operation u on the global view. This operation may require coordination.

- merge incorporates the local updates to the global view using a type-specifc merge operation and updates the local view.

While weakRead and weakUpdate act exclusively on the local copy, strongRead and strongUpdate act on the global state. The combination of these two operations supports flexible optimizations on each data type. For example, a queue can guarantee that an element is dequeued only once by executing dequeues in strongUpdate. At the same time, enqueues can be applied as weakUpdate and merged later for better performance (See Section 3.5 for full specification and implementation). For a counter, we may want to enforce a weak limit on the maximum value, i.e., its value should not diverge arbitrarily from the defined maximum value. Such a counter can use a strongRead to check the global value to adapt the merge interval or to switch to a fully synchronized version.

A *multi-view data type (MDT)* is a data type designed using the global-local view model that defines a merge operation in addition to other operations. A *mergeable data type* is a multi-view data type where are all updates and reads are weak operations. A *hybrid mergeable data type* is a multi-view data type where some updates and/or reads are strong operations. The types with only strong operations offer a sequential semantics and define neither the local views nor the merge operation.

3.2 Use cases

We identified several applications where linearizable data types are not required for the correctness of the application. In this section, we discuss several such applications in which using the global-local view model is better regarding programmability and/or scalability.

A work-stealing queue.

A task queue is used to distribute tasks among threads running in parallel. The threads produce new tasks and push them into the task queue. When a thread is out of work, it gets a new task from the task queue. A typical implementation of a task queue uses a linearizable queue, where the push is implemented by *enqueue* on the queue and pop by *dequeue*. However, such a task queue requires synchronization for each push and pop operation. In most cases, a linearizable queue is not needed for task distribution because the order of task execution is not critical as long as the tasks are done eventually.

Task queues are used in many work schedulers. As an example, consider the Cilk language [18]. Cilk is a multithreaded language for parallel programming that generalizes the semantics of C. The programs written for Cilk employ fork-join parallelism. A pool of threads, called workers, is available to execute the tasks. A worker, when it encounters the keyword *spawn*, creates a new child task and adds it to the task queue. The child task can be immediately executed if resources are available. The worker can continue executing the parent task without waiting for the child task. If the parent needs a result from the child, it uses the sync keyword which suspends the parent task; the worker thread then retrieves more tasks from the task queue.

The task queue in Cilk is implemented using a work-stealing queue [39]. Each thread has a deque (a double ended queue), which has the following operations: *popBack*, *pushTop*, *popTop*. A thread uses it as a stack, adding newly created tasks using *pushTop*, and retrieving more work when the current task completes using *popTop*. If a worker's queue is empty when it looks for more work, it will attempt to steal a task from other queues, using *popBack*. The Cilk runtime uses a lock-free protocol to handle concurrent access to the queue by the owner and the thief.

A work stealing queue with this semantics is a natural fit to the global-local view model. Instead of a queue per thread, we have a multi-view queue with a global view and a local-view per thread. Workers can add and retrieve tasks using *pushTop* and *popTop* that execute on the thread-local views without any concurrency control, and remote threads can steal tasks from the shared global view.

One disadvantage of this design is that it may prevent threads from stealing tasks when the global view is empty even if there are unmerged tasks in the local views. To avoid this, we may set a minimum number of items in the global view. The threads periodically check the global view and merge their local view when the number of tasks in the shared queue drops below this threshold.

In-memory multi-core databases.

One way to exploit the parallelism of a many-core system is by partitioning the data among the processes running in each core. Non-overlapping partitions allow executing requests to different partitions in parallel. However, concurrent requests to the same partition or objects will be executed sequentially by the process responsible for the partition. An in-memory database with or without a partitioning system employs various concurrency control mechanisms to guarantee serializable executions of concurrent conflicting transactions. A high number of conflicting transactions may impact the performance of such systems. Unfortunately, as reported in [72], such high contention workloads are common.

In high contention workloads, we can achieve high performance by allowing concurrent conflicting transactions to proceed in parallel on different cores. Instead of serializing the access to the objects, the transactions can update a per core copy of the object and merge them later. Doppel [72] uses a phase reconciliation mechanism that automatically parallelizes high contention transactions. When the system detects high contention on data items in-memory databases, it switches to a split phase where the transactions update a local per-core copy of the contended data in parallel. After the split phase, the per-core copies are merged, and the transactions proceed to execute using classical concurrency control techniques. Whether transactions can be executed in the split phase, is decided based on the commutativity of operations, thus preserving sequential consistency.

A multi-view data type implemented in the global-local view model is a natural fit for a system that uses phase reconciliation. MDTs supports the per-core copy of the objects through its local views. During the normal execution, it can use the *strong* operations of the MDT that updates the shared global view. During the split phase, it switches to using the *weak* operations that update the local view. The MDT implements the *merge* operation which is immediately available for the reconciliation phase. The design of MDTs is tuned to avoid the use of any concurrency control while accessing per-core copies. The merge executes faster in comparison to executing the operations in sequence, thus fully exploiting the benefits of phase reconciliation.

Message queues.

In the widely-used messaging library ZeroMQ [6], adding messages to the queue is at the core of the application. While lock-free linearizable queues are fast, the developers observed that enqueuing new messages were affecting the overall performance, especially in high contention workloads [85]. However, only the relative order of messages from a single thread is relevant for the semantics of the message queue; it is not necessary to maintain a strict order of enqueue operations when two independent threads try to insert messages concurrently into the queue. To overcome the performance penalty, the developers re-engineered the message queues such that multiple messages are added as a batch, thus using only one single atomic operation for a batch of messages.

Message queues where multiple messages can be batched together and added to the shared queue is a direct application of the hybrid queue described in Section 3.5.2. The fast merge provides the expected performance as well as the desired semantics required by the messaging library.

Aggregation counters and other statistical data types

The applications that use aggregation counters that are computed by parallel threads are amenable to using the mergeable counter. Similarly, the objects that store statistical measures such as sums, min, max etc. that are computed by parallel threads will benefit from the global-local view model. The commutativity and associativity properties of these computations allow threads to do computations on thread-local view and later aggregate the results during the merge without re-computing.

Software Transactional Memory.

In software transactional memory, we can use mergeable objects to avoid unnecessary aborts where the conflicting updates can be meaningfully merged [11]. A detailed semantics and implementation of such a transactional memory algorithm are described in Chapter 4.

3.3 Specification of Multi-view Data Types

Multi-view Data Types (MDT) define a subset of the basic operations from the global-local view model, depending on the semantics needed. A purely mergeable MDT defines only the weak operations and merge, while a hybrid MDT defines strong operations in addition to the weak operations and merge. In this section, we define an abstract specification of multi-view data types in relation to the corresponding sequential data type.

A sequential data type τ is specified by a tuple

$$S_\tau = (\mathcal{U}_\tau, \mathcal{Q}_\tau, query_\tau, apply_\tau, \mathcal{R}_\tau)$$

where $\mathcal{U}_\tau$ is a set of update operations, $\mathcal{Q}_\tau$ is a set of query operations, $query_\tau$ is an evaluation function that evaluates the query operation. Given a sequence of update operations σ and a query $q \in \mathcal{Q}_\tau$, $query_\tau(q, \sigma)$ returns a result $r \in \mathcal{R}_\tau$. Given a sequence of update operations σ, $apply_\tau(\sigma)$ returns the result of applying the operations in σ on the initial state of the object with type τ.

Given a sequential specification for type τ, we can define a multi-view specification by defining a merge operation. A multi-view data type τ is specified by a tuple

$$M_\tau = (S_\tau, \mathcal{U}^w_\tau, \mathcal{Q}^w_\tau, \mathcal{U}^s_\tau, \mathcal{Q}^s_\tau, merge_\tau)$$

where

- S_τ is the sequential specification of the data type
- $\mathcal{U}^w_\tau \subseteq \mathcal{U}_\tau$ defines the set of weak updates.
- $\mathcal{Q}^w_\tau \subseteq \mathcal{Q}_\tau$ defines the set of weak reads.
- $\mathcal{U}^s_\tau \subseteq \mathcal{U}_\tau$ defines the set of strong updates.
- $\mathcal{Q}^s_\tau \subseteq \mathcal{Q}_\tau$ defines the set of strong reads.
- $merge_\tau(\sigma_c, \sigma_l)$ defines the semantics to merge the concurrent updates σ_c and σ_l. It returns a sequence of updates σ_r which can be appended to the sequence of operations in global view $\sigma_g$that already included σ_c. (See abstract execution semantics in Figure 3.13 to see how this function is used by the global-local view model.)

Following the above specification method, we discuss the specification of several multi-view data types in relation to the sequential specification of the corresponding sequential data types.

Counter Figure 3.4 gives the specification for the sequential counter and the specifications for purely mergeable counter and hybrid counter. A sequential counter defines updates *inc* (increment by 1) and *dec* (decrement by 1), and query *getValue* (returns the current value). The specification of $merge_{counter}$ given by the multi-view specification returns the sequence of local operations

Sequential Counter:
$S_{counter} = (\mathcal{U}_{counter}, \mathcal{Q}_{counter}, query_{counter}, \mathcal{R}_{counter})$

$$\begin{aligned} \mathcal{U}_{counter} &= \{inc, dec\} \\ \mathcal{Q}_{counter} &= \{getValue\} \\ \mathcal{R}_{counter} &= \mathbb{Z} \\ query_{counter}(getValue, \sigma) &= \big|\{op \in \sigma | op = inc\}\big| - \big|\{op \in \sigma | op = dec\}\big| \end{aligned}$$

Multi-view Counter:
$M_{counter} = (S_{counter}, \mathcal{Q}^{w}_{counter}, \mathcal{U}^{w}_{counter}, \mathcal{Q}^{s}_{counter}, \mathcal{U}^{s}_{counter}, merge_{counter})$

$$merge_{counter}(\sigma_c, \sigma_l) = \sigma_l$$

Purely Mergeable Counter	Hybrid Counter
$\mathcal{U}^{w}_{counter} = \{inc, dec\}$	$\mathcal{U}^{w}_{counter} = \{inc, dec\}$
$\mathcal{Q}^{w}_{counter} = \{getValue\}$	$\mathcal{Q}^{w}_{counter} = \{getValue\}$
$\mathcal{U}^{s}_{counter} = \emptyset$	$\mathcal{U}^{s}_{counter} = \{inc, dec\}$
$\mathcal{Q}^{s}_{counter} = \emptyset$	$\mathcal{Q}^{s}_{counter} = \{getValue\}$

Figure 3.4: Specification of Multi-view counter.

(σ_l) which can be appended to the global view as defined in Rule Merge in Figure 3.13. Since all update operations of the counter are commutative, we can simply append the local operations to the global sequence.

The specification of a purely mergeable counter differs from that of a hybrid counter in that the former defines only *weak* operations and the later defines bother *weak* and *strong* operations. The applications using a hybrid counter can choose *weak* or *strong* operations dynamically based on desired semantics. For example, an application that uses weak increments initially can switch to use strong increments when it observes that the value reached a threshold.

Queue The queue data type has operations *enqueue*(e) (enqueues an item e) to the queue), and *dequeue* (dequeues an item from the queue). A hybrid mergeable queue is defined with a weak enqueue and strong dequeue (Figure 3.5). The weak enqueues can be executed concurrently on the local-view and merged later for better performance. At the same time, strong dequeues guarantee that an item is dequeued only once. In the above semantics, if the global copy is empty, *dequeue* returns null even if there are local enqueue operations by the same thread which have not been merged yet.

A queue with weak enqueue and weak dequeue may be useful if a redundant

$$\begin{aligned} merge_{queue}(\sigma_c, \sigma_l) &= \sigma_l \\ \mathcal{U}^w_{queue} &= \{enqueue(e) | \forall\ e\} \\ \mathcal{U}^s_{queue} &= \{dequeue\} \end{aligned}$$

Figure 3.5: Specification of Multi-view hybrid queue.

dequeue is semantically acceptable for the application. A queue with only strong enqueue and strong dequeue behaves as a linearizable queue.

Grow-only Bag A grow-only bag is a set that provides only an *add* operation, and allows duplicate elements. A purely mergeable bag implements a weak add, weak lookup and merge (Figure 3.6).

$$\begin{aligned} merge_{bag}(\sigma_c, \sigma_l) &= \sigma_l \\ \mathcal{U}^w_{bag} &= \{add(e) | \forall e\} \\ \mathcal{U}^s_{bag} &= \emptyset \\ \mathcal{Q}^w_{bag} &= \{lookup(e) | \forall e\} \\ \mathcal{Q}^s_{bag} &= \emptyset \end{aligned}$$

Figure 3.6: Specification of Multi-view Grow-only Bag.

Add-wins set Add-wins set (AW-set) is a set that allows both add and remove operations. When there are concurrent add and remove of the same item, the add wins during the merge, i.e., the resulting set after the merge contains the item.

$$\begin{aligned} & merge_{awset}(\sigma_c, \sigma_l) = \sigma_l', \text{ where} \\ & \qquad \sigma_l' = \{m \mid m \in \sigma_l \wedge \\ & \qquad\qquad (m = add(a) \wedge \nexists m' \in \sigma_l : m' = rem(e) \wedge m \rightsquigarrow m') \\ & \qquad\qquad \vee (m = rem(a) \wedge (\nexists m' \in \sigma_c : m' = add(a) \\ & \qquad\qquad \wedge \exists m'' \in \sigma_l : m'' = add(a) \wedge m \rightsquigarrow m''))\} \\ & m \rightsquigarrow m' \implies \text{index}(\sigma_l, m) < \text{index}(\sigma_l, m) \\ & \mathcal{U}^w_{awset} = \{add(e), remove(e) | \forall e\} \\ & \mathcal{U}^s_{awset} = \emptyset \\ & \mathcal{Q}^w_{awset} = \{lookup(e) | \forall e\} \\ & \mathcal{Q}^s_{awset} = \emptyset \end{aligned}$$

Figure 3.7: Specification of Multi-view Add-wins Set.

3.4 Abstract execution model

The system we consider is built upon a classical shared-memory architecture as supported by specifications such as the C++ or Java memory models. We assume that the system consists of a dynamic number of threads. Any thread can spawn new threads that may outlive their parent thread. The system distinguishes two types of memory: local memory is associated with a single thread and can only be accessed by this thread; any thread can access the shared memory. Communication and coordination between the threads are done exclusively via shared-memory objects. We assume that there are no side channels. In particular, spawned threads do not inherit local objects from their parents.

The state of the objects are modeled as sequence of updates (σ). A sequence σ_i can be concatenated with another sequence σ_j, denoted by $\sigma_i \cdot \sigma_j$. The global view of a multi-view object is a sequence of updates represented as σ_g. The local view is a tuple $\langle \sigma_s, \sigma_l \rangle$, where σ_s, the local snapshot, is the state of the global view when this thread last synchronized with the global view through a merge or pull operation, and σ_l is the sequence of local operations that have not been merged yet.

In later sections, we describe the language (Section 3.4.1) for the global-local view model and its execution semantics (Section 3.4.2). The semantics assumes the abstract specification of multi-view data types defined in Section 3.3.

3.4.1 Common language syntax

We define a high-level language $\mathscr{A}_{\text{MDT}}$ in which the programs using the global-local view model are written. We use $\mathscr{A}_{\text{MDT}}$ to define the abstract operations of the global-local view model and to express their semantics. Later, we use the same language to express the implementations of MDTs.

Figure 3.8 shows the syntax of $\mathscr{A}_{\text{MDT}}$. $\mathscr{A}_{\text{MDT}}$ is defined as an imperative language extended with the operations defined by the global-local view model. To facilitate the use of the same language for the abstract execution model and the MDT implementations, we distinguish the expressions and the statements common to both as Exp_{imp} and Stmt_{imp}. Exp_{gl} and Stmt_{gl} are the expressions and the statements that define the abstract operations in the global-local view model.

We use the following naming conventions: V is a variable name, e an expression, v a value, s a statement, M a method name, S a C-like structure with multiple fields and F a field name in this structure. A value is either a reference r, a number n, a sequence σ of MDT operations op, a tuple of values $\langle v_1, v_2 \rangle$ or unit $()$. Exp_{imp} are expressions defined in a common imperative language, given as values, variables, numerical expressions, field access, method invocation, and object creation. Exp_{gl} denote the expressions which are part of the global-local view model operations. The expressions marked in gray do not appear in source programs but represent dynamically generated locations and intermediate system states. Statements comprise variable assignment, field assignment, conditional statements, loops, method invocation, return statement, lock access, thread fork, and skip. The operations from the global-local view model are defined in Stmt_{gl}. `weakUpdate` and `strongUpdate` can be used both as expression and statement because the update operations may return values which can be used in other expressions or statements. For example, *dequeue* on a queue is an update operation that returns an item. Statements can be sequentially composed as $s_1; s_2$.

$$
\begin{array}{rcl}
V & \in & \text{Var} \\
r & \in & \text{Ref} \\
n & \in & \text{Number} \\
op & \in & \mathcal{Q}_\tau \cup \mathcal{U}_\tau, \text{ for all multi-view types } \tau \\
\sigma & \in & \text{Sequence of operations } op \\
\langle v_1, v_2 \rangle & \in & \text{Tuple} \\
\tau & \in & \text{Multi-view Types} \\
e & \in & \text{Exp}_{imp} \cup \text{Exp}_{gl} \\
s & \in & \text{Stmt}_{imp} \cup \text{Stmt}_{gl} \\
\text{Val} & ::= & \colorbox{lightgray}{r} \mid n \mid \colorbox{lightgray}{σ} \mid \mathit{true} \mid \mathit{false} \mid \langle v_1, v_2 \rangle \mid () \\
\text{Exp}_{imp} & ::= & v \mid V \mid e + e \mid e * e \mid \ldots \mid \\
 & & e.F \mid M(\bar{e}) \mid \texttt{new}\ \tau \\
\text{Stmt}_{imp} & ::= & s; s \mid V := e \mid e_1.F := e_2 \mid \\
 & & \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{endif} \mid \\
 & & \texttt{while}\ e\ \texttt{do}\ s\ \texttt{end} \mid M(\bar{e}) \mid \texttt{return}\ e \mid \\
 & & \texttt{lock}\ e \mid \texttt{unlock}\ e \mid \texttt{fork}(\overline{V})\{s\} \mid \texttt{skip} \\
\text{Exp}_{gl} & ::= & \texttt{weakRead}\ op\ e \mid \texttt{strongRead}\ op\ e \mid \\
 & & \texttt{weakUpdate}\ op\ e \mid \texttt{strongUpdate}\ op\ e \\
\text{Stmt}_{gl} & ::= & \texttt{weakUpdate}\ op\ e \mid \texttt{strongUpdate}\ op\ e \mid \\
 & & \texttt{pull}\ e \mid \texttt{merge}\ e
\end{array}
$$

Figure 3.8: Syntax of $\mathscr{A}_{\text{MDT}}$.

An $\mathscr{A}_{\text{MDT}}$ program is a list of state definitions, a list of method definitions and a list of statements. A *state definition* defines a structure with a type name and a list of field definitions. A *method definition* consists of a method name, parameter types and names, and the method body which consists of statements.

Figure 3.9 is a program in $\mathscr{A}_{\text{MDT}}$ with two threads that access a shared multi-view counter ($M_{counter}$). The program does not have additional state definitions or method definitions. It consists of only a sequence of statements. Here, `inc` (increments the counter by one) and `getValue` (returns the value of the counter) are the operations defined by the counter data type. At the end of the execution of this program, the value of the counter can be either 2 or 3 depending on the scheduling of steps from the two concurrent threads. If the second thread executes the `strongRead` after the first thread's `merge`, the value of b is not 0 and the thread 2 skips the merge. A formal operational semantics that leads to this behavior is explained in the next section.

```
c := new Counter;                        ▷ instantiate a counter
fork (c) {                          ▷ increment by 2 and merge.
   pull c;
   weakUpdate inc c;
   weakUpdate inc c;
   merge c;
};
fork (c) { ▷ increment by 1, then read its value and merge.
   pull c;
   weakUpdate inc c;
   b := strongRead getValue c;
   if b=0 then
      merge c;
   else
      skip;
   end if
};
```

Figure 3.9: A program in $\mathscr{A}_{\text{MDT}}$ accessing a $M_{counter}$.

3.4.2 Operational semantics

Memory Model The memory model distinguishes three types of memory:

1. a global heap $\Gamma : \text{Ref} \rightharpoonup \text{Val}$ accessible by all threads,
2. a local heap $\Lambda : \text{Ref} \rightharpoonup \text{Val}$ for every thread t, and
3. a local stack $\Theta = \bar{\vartheta}$ for every thread, $\vartheta : \text{Var} \rightharpoonup \text{Val}$.

A reference r corresponds to a location allocated in heap Γ or Λ. $\Gamma(r)$ returns the value on the global heap associated with the reference r. $\Gamma[r \mapsto v]$ returns a heap identical to Γ, except that it maps r to v. $\Lambda(r)$ and $\Lambda[r \mapsto v]$ behaves similarly.

Θ is a stack of memory blocks represented as $\vartheta;\bar{\vartheta}$, where ϑ is the top of the stack and $\bar{\vartheta}$ represents the rest of the memory blocks. ϑ represents the memory block of the current method incarnation and $\bar{\vartheta}$ the rest of the blocks from the parent methods. $\vartheta(V)$ denotes the value associated with the variable V and $\vartheta[V \mapsto v]$ returns an identical block to ϑ, but with the mapping of V updated to v.

State. The state of the system is represented by a pair $\Pi;\Gamma$. Π is a thread system of active threads, and Γ represents the global heap. The state of a thread t is a pair $\langle s, \Theta, \Lambda\rangle$, where s is the statements to evaluate and Λ is the local heap accessible only by thread t. The thread system Π maps a thread identifier t to the state of the thread.

$$\begin{aligned} t &\in \text{ThreadIds} \\ \Pi &\in \text{ThreadIds} \rightharpoonup \langle \text{Stmt}, \Theta, \Lambda\rangle \end{aligned}$$

Evaluation Figure 3.10 introduces the evaluation contexts that the language constructs define. $\oplus$ is a placeholder for binary operators such as $+, *$, etc.

The evaluation of a program with body s starts with an initial state $\Pi\{0 \mapsto \langle s, [\ ], [\ ]\rangle\}; [\ ]$, with an empty global heap and a main thread with id 0 and empty local heap and local stack, and ends when there are no more statements to be evaluated in any thread ($\Pi\{\forall t : t \mapsto \langle \texttt{skip}, \Theta_t, \Lambda_t\rangle\}; \Gamma$). Figure 3.11 shows the evaluation steps of the thread. Each reduction step $\rightarrowtail$ non-deterministically selects a thread from Π, thus modeling an arbitrary thread scheduling.

Rule Spawn shows the steps to `fork` a thread. There are no global variables. If a thread needs to access any variables defined in the parent, the parent thread must explicitly pass the values of the variables to the child at the time of child's creation. Hence the first argument of `fork` is $\overline{V}$ which is a list of variables to be passed to the new thread. The `fork` creates a new local heap ϑ' with the variables $\overline{V}$ mapped to its values from ϑ ($\vartheta|_{\overline{V}}$). The second argument is a statement s to be evaluated by the new thread. A new thread-id t' maps to the state of the new thread which consists of ϑ', an empty

Evaluation Contexts:

$$\begin{gathered}
\mathbb{E} = [\cdot] \oplus e \mid v \oplus [\cdot] \mid [\cdot].F \mid M\,[\cdot] \mid \\
[\cdot]; s \mid V := [\cdot] \mid r.F := [\cdot] \mid \\
\texttt{if}\ [\cdot]\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{endif} \mid \texttt{while}\ [\cdot]\ \texttt{do}\ s\ \texttt{end} \mid \\
\texttt{return}\ [\cdot] \mid \texttt{lock}\ [\cdot] \mid \texttt{unlock}\ [\cdot] \mid \\
\texttt{weakUpdate}\ op\ [\cdot] \mid \texttt{weakRead}\ op\ [\cdot] \mid \\
\texttt{strongUpdate}\ op\ [\cdot] \mid \texttt{strongRead}\ op\ [\cdot] \mid \\
\texttt{pull}\ [\cdot] \mid \texttt{merge}\ [\cdot]
\end{gathered}$$

Figure 3.10: Operational Semantics of $\mathscr{A}_{\text{MDT}}$: Evaluation contexts.

$$\frac{t' \text{ fresh} \qquad \Theta = \vartheta; \bar{\vartheta} \qquad \vartheta' = \vartheta|_{\overline{V}} \qquad \Theta' = \vartheta'; \Theta}{\Pi\{t \mapsto \langle \mathbb{E}[\texttt{fork}\ (\overline{V})\{s\}], \Theta, \Lambda\rangle\}; \Gamma \rightarrowtail \Pi\{t \mapsto \langle \mathbb{E}[()], \Theta, \Lambda\rangle, t' \mapsto \langle \mathbb{E}[s], \Theta', [\,]\rangle\}; \Gamma} \text{ SPAWN}$$

$$\frac{\langle \mathbb{E}[s], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[s'], (\Theta', \Lambda', \Gamma'))\rangle}{\Pi\{t \mapsto \langle \mathbb{E}[s], \Theta, \Lambda\rangle\}; \Gamma \rightarrowtail \Pi\{t \mapsto \langle \mathbb{E}[s'], \Theta', \Lambda'\rangle\}; \Gamma'} \text{ THREADSTMT}$$

$$\frac{\langle \mathbb{E}[e], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[e'], (\Theta', \Lambda', \Gamma'))\rangle}{\Pi\{t \mapsto \langle \mathbb{E}[e], \Theta, \Lambda\rangle\}; \Gamma \rightarrowtail \Pi\{t \mapsto \langle \mathbb{E}[e'], \Theta', \Lambda'\rangle\}; \Gamma'} \text{ THREADEXP}$$

Figure 3.11: Operational Semantics for $\mathscr{A}_{\text{MDT}}$: Thread Evaluation $\rightarrowtail$

local heap and s.

All other statements (and expressions) are evaluated by $\hookrightarrow$. A statement or expressions can modify the local heap and/or the global heap given by the state transformation

$$\langle s, (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle s', (\Theta', \Lambda', \Gamma'))\rangle$$

as shown in Rules ThreadStmt and ThreadEval. The reduction rules $\hookrightarrow$ for evaluating expressions (Exp_{imp}) and statements (Stmt_{imp}) are defined in Figure 3.12, and expressions (Exp_{gl}) and statements (Stmt_{gl}) in Figure 3.13.

Figure 3.12 shows the evaluation steps of expressions and statements. Rule Seq show sequential composition. Rule BinOp shows the steps for the evalu-

$$\begin{array}{c}\langle \mathbb{E}[v; s], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[s], (\Theta, \Lambda, \Gamma)\rangle \\ \langle \mathbb{E}[\texttt{skip}; s], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[s], (\Theta, \Lambda, \Gamma)\rangle\end{array}\ \textsc{Seq}$$

$$\langle \mathbb{E}[v_1 \oplus v_2], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[\mathit{binop}(\oplus, v_1, v_2)], (\Theta, \Lambda, \Gamma)\rangle\ \textsc{BinOp}$$

$$\frac{\Theta = \vartheta; \bar{\vartheta} \qquad \vartheta(V) = v}{\langle \mathbb{E}[V], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[v], (\Theta, \Lambda, \Gamma)\rangle}\ \textsc{VarAccess}$$

$$\frac{\Theta = \vartheta; \bar{\vartheta} \qquad \vartheta' = \vartheta[V \mapsto v] \qquad \Theta' = \vartheta'; \bar{\vartheta}}{\langle \mathbb{E}[V := v], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[\texttt{skip}], (\Theta', \Lambda, \Gamma)\rangle}\ \textsc{Assign}$$

$$\frac{\Gamma(r) = S \qquad S(F) = v}{\langle \mathbb{E}[r.F], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[v], (\Theta, \Lambda, \Gamma)\rangle}\ \textsc{FieldAccess}$$

$$\frac{\Gamma(r) = S \qquad S' = S[F \mapsto v] \qquad \Gamma' = \Gamma[r \mapsto S']}{\langle \mathbb{E}[r.F := v], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[\texttt{skip}], (\Theta, \Lambda, \Gamma')\rangle}\ \textsc{FieldAssign}$$

$$\frac{\texttt{newaddr}(\Gamma) = r}{\langle \mathbb{E}[\texttt{new}\ \tau], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[r], (\Lambda, \Gamma[r \mapsto \emptyset])\rangle}\ \textsc{Alloc}$$

$$\frac{s = \begin{cases} s_1, \text{if } v = \mathit{true} \\ s_2, \text{if } v = \mathit{false} \end{cases}}{\langle \mathbb{E}[\texttt{if}\ v\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{endif}], (\Theta, \mathcal{H})\rangle \hookrightarrow \langle \mathbb{E}[s], (\Theta, \mathcal{H})\rangle}\ \textsc{Conditional}$$

$$\frac{w = \texttt{while}\ e\ \texttt{do}\ s\ \texttt{end}}{\langle \mathbb{E}[w], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[\texttt{if}\ e\ \texttt{then}\ s; w\ \texttt{else skip endif}], (\Theta, \Lambda, \Gamma)\rangle}\ \textsc{Loop}$$

$$\frac{\begin{array}{c}\mathit{fundef}(M) = (\overline{pn}, s) \\ \mathit{length}(\overline{pn}) = \mathit{length}(\bar{v}) \qquad \vartheta = \mathit{mapall}[\overline{pn} \mapsto \bar{v}]\end{array}}{\langle \mathbb{E}[M(\bar{v})], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[s], (\vartheta; \Theta, \Lambda, \Gamma)\rangle}\ \textsc{MethodCall}$$

$$\langle \mathbb{E}[\texttt{return}\ v], (\vartheta : \Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[v], (\Theta, \Lambda, \Gamma)\rangle\ \textsc{Return}$$

$$\frac{\Gamma(\mathit{lock}(r)) = \mathit{false} \qquad \Gamma' = \Gamma[\mathit{lock}(r) \mapsto \mathit{true}]}{\langle \mathbb{E}[\texttt{lock}\ r], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[\texttt{skip}], (\Theta, \Lambda, \Gamma')\rangle}\ \textsc{Lock}$$

$$\langle \mathbb{E}[\texttt{unlock}\ r], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[\texttt{skip}], (\Theta, \Lambda, \Gamma[\mathit{lock}(r) \mapsto \mathit{false}])\rangle\ \textsc{Unlock}$$

Figure 3.12: Operational semantics for $\mathscr{A}_{\text{MDT}}$: Evaluation of expressions and statements (Stmt_{imp} and Exp_{imp}).

ation of binary operations such as $+, *$ respectively. Rules Conditional and Loop show conditional statements and while loops.

Variable Access and Assignment A variable V is always stored in the local stack. The scope of a variable is limited to the stack of the current method, which is ϑ. Evaluating a variable access V returns a value v from ϑ (Rule VarAccess). Evaluating an assignment of a value to a variable v updates the local stack memory (Rule Assign).

Field Access and Assignment To access a field, first, we look up the structure in the global heap. Next, the value of the field is looked up in the structure (Rule FieldAccess). $S(F)$ returns the value associated with the field F in the structure S. The assignment is done similarly, which updates the structure in the global heap (Rule FieldAssign). $S[F \mapsto v]$ returns a structure identical to S except the value of F updated to v.

Allocation Rule Alloc shows the steps for creating a new object. `newaddr`(Γ) returns an unused reference in Γ. The heap is updated with reference r mapping to an empty value denoted by $\emptyset$. The result of the expression is the newly allocated reference r.

Method Call The definition of method M is retrieved by an auxiliary function *fundef*, which is a pair consisting of a list of parameter names $\overline{pn}$ and a method body consisting of statements s to be executed. An auxiliary function *mapall* maps the parameter names to parameter values, which is then assigned to a new memory block ϑ. ϑ is added to the local stack (Rule MethodCall).

When a method execution finishes, and it returns, the local memory block ϑ is removed from the stack. The result of the evaluation is the return value v.

Locks *lock(r)* returns a location in the heap that represents the lock associated with r. Rule Lock and Rule Unlock shows the evaluation of lock and unlock operations.

The global-local view operations Figure 3.13 shows the evaluation steps for the operations defined by the global-local view model.

The operation `pull` synchronizes the local view with the global view (Rule Pull). The global view referenced by r in the global heap is σ_g. The local

$$\frac{\Gamma(r) = \sigma_g}{\langle \mathbb{E}[\texttt{pull}\ r], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[\texttt{skip}], (\Theta, \Lambda[r \mapsto \langle \sigma_g, \emptyset\rangle], \Gamma)\rangle}\ \textsc{Pull}$$

$$\frac{\begin{array}{c}\Lambda(r) = \langle \sigma_s, \sigma_l\rangle \\ \mathit{type}(r) = \tau \qquad q \in \mathcal{Q}^w_\tau \qquad v = \mathit{query}_\tau(q, \sigma_s \cdot \sigma_l)\end{array}}{\langle \mathbb{E}[\texttt{weakRead}\ q\ r], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[v], (\Theta, \Lambda, \Gamma)\rangle}\ \textsc{WeakRead}$$

$$\frac{\begin{array}{c}\Lambda(r) = \langle \sigma_s, \sigma_l\rangle \\ \mathit{type}(r) = \tau \qquad \mathit{op} \in \mathcal{U}^w_\tau \qquad v = \mathit{apply}_\tau(\sigma_s \cdot \sigma_l, \mathit{op})\end{array}}{\langle \mathbb{E}[\texttt{weakUpdate}\ \mathit{op}\ r], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[v], (\Theta, \Lambda[r \mapsto \langle \sigma_s, \sigma_l \cdot \mathit{op}\rangle], \Gamma)\rangle}\ \textsc{WeakUpdate}$$

$$\frac{\begin{array}{c}\Gamma(r) = \sigma_s \cdot \sigma_c \qquad \Lambda(r) = \langle \sigma_s, \sigma_l\rangle \\ \mathit{type}(r) = \tau \qquad \mathit{merge}_\tau(\sigma_c, \sigma_l) = \sigma_r \qquad \sigma_g = \sigma_s \cdot \sigma_c \cdot \sigma_r\end{array}}{\langle \mathbb{E}[\texttt{merge}\ r], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[\texttt{skip}], (\Theta, \Lambda[r \mapsto \langle \sigma_g, \emptyset\rangle], \Gamma[r \mapsto \sigma_g])\rangle}\ \textsc{Merge}$$

$$\frac{\Gamma(r) = \sigma_g \qquad \mathit{type}(r) = \tau \qquad q \in \mathcal{Q}^s_\tau \qquad v = \mathit{query}_\tau(q, \sigma_g)}{\langle \mathbb{E}[\texttt{strongRead}\ q\ r], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[v], (\Theta, \Lambda, \Gamma)\rangle}\ \textsc{StrongRead}$$

$$\frac{\begin{array}{c}\Gamma(r) = \sigma_g \\ \mathit{type}(r) = \tau \qquad \mathit{op} \in \mathcal{U}^s_\tau \qquad v = \mathit{apply}_\tau(\sigma_g, \mathit{op})\end{array}}{\langle \mathbb{E}[\texttt{strongUpdate}\ \mathit{op}\ r], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[v], (\Theta, \Lambda, \Gamma[r \mapsto \sigma_g \cdot \mathit{op}])\rangle}\ \textsc{StrongUpdate}$$

Figure 3.13: Operational semantics for $\mathscr{A}_{\text{MDT}}$: Evaluation of expressions and statements (Stmt_{gl} and Exp_{gl}).

view referenced by r in the local heap is set to the tuple $\langle \sigma_g, \emptyset \rangle$, updating the local snapshot with global state and overwriting all the local updates.

`weakRead` returns the result of a type-specific read-only operation q on the state obtained by applying local updates on the local snapshot ($query_\tau(q, \sigma_s \cdot \sigma_l)$). Rule WeakRead evaluates the `weakRead` operation. $q \in \mathcal{Q}^w_\tau$ is a query operation defined as weak read on the type τ of the reference r. The query q is evaluated according to the type specification (see Section 3.3). The result of this expression is the result of the query operation. Since this is a weak read operation, it uses only the local heap. The local and global heaps remain unchanged.

`weakUpdate` applies the update method on the local view by appending the update operation op to σ_l (Rule WeakUpdate). $op \in \mathcal{U}^w_\tau$ is a weak update operation on type τ of reference r (Section 3.3). The local view referenced by r in local heap Λ is updated. Weak updates do not modify the global view.

The `merge` operation merges the local state $\langle \sigma_s, \sigma_l \rangle$ to the global state $\sigma_s \cdot \sigma_c$ (Rule Merge). σ_s is the snapshot seen by the thread when it last executed `merge` or `pull`. σ_c is the sequence of concurrent operations merged to the global view. σ_l is the list of updates executed on the local view that are not yet merged. σ_g is the result of type specific merge operation as defined by its specification (see Section 3.3). Merge updates the reference r in the global heap to σ_g and the local heap with $\langle \sigma_g, \emptyset \rangle$. The `merge` operation thus atomically merges the local view to the global view and updates the local view with the result.

Similar to `weakRead`, `strongRead` evaluates a query q on type τ (Rule StrongRead). However, $q \in \mathcal{Q}^s_\tau$ is a query operation defined as strong read. Instead of using the local view, Rule StrongRead evaluates q on the global view σ_g referenced by r in the global heap. The expression returns the result of $query_\tau(q, \sigma_g)$. Neither the global view nor the local view is modified.

Similarly Rule StrongUpdate evaluates the `strongUpdate` operation. It applies the update operation op on the global view by appending op to σ_g and updates the reference r in global view. $op \in \mathcal{U}^s_\tau$ is a strong update operation on type τ defined by the type specification of τ (Section 3.3).

3.5 Implementation

The abstract model expresses the high level semantics of the global-local view model. The MDT implementations define the concrete state in terms of the underlying data structures and the algorithms to modify those data structures.

First, we discuss the properties that we need to consider when implementing an MDT. Then, we present the implementations of several MDTs.

3.5.1 Properties

Exploiting object semantics to define the merge function has been successfully applied in Conflict-free Replicated Data Types (CRDTs) [80] in the context of distributed database systems. State-based CRDTs rely on lattice-based monotonic data values where the merge computes the least upper bound. Operation-based CRDTs re-execute updates that were issued on the local object instance against the global object, therefore requiring commutativity of concurrent updates to achieve consistency despite different orders of update application at the different replicas.

In general, CRDTs employ various mechanisms to achieve deterministic results for objects with non-commutative operations, e.g., maintaining tombstones for sets where elements can be added and removed. While CRDTs have been successful in avoiding costly synchronization in replicated data stores, employing the known specifications of CRDTs in multi-/many-core programs seems prohibitively expensive. In this section, we identify two main properties for MDTs - *persistence* and *mergeability*, that is critical for efficient and scalable implementations of MDTs.

Persistence A multi-view object maintains multiple versions: thread-local versions which reflect the modifications done by the threads and a global version updated by merge. Threads concurrently access and modify these versions. Thus, multiple thread-local versions and global versions must coexist. This property is called persistence (as used in the domain of functional data structures [35]). A data structure is said to be persistent if multiple versions of it are accessible. It is partially persistent if only some versions are accessible, and fully persistent if all versions are accessible [35]. Persistence is an essential property for MDTs because it allows concurrent access to multiple versions.

There are different mechanisms to achieve persistence, such as copy-on-write or path copying [35]. Depending on the data type semantics, their implementations may use specific techniques to make them efficiently persistent. Hence, it is essential to make persistence a property of the MDT rather than a universal mechanism implemented by a generic version manager.

Typically, not all versions have to be persisted. Old versions that are known not to be required and never be reaccessed can be garbage-collected.

For example, a local version which is already merged to the global view will never be read again.

Mergeability A shared object may be concurrently accessed and updated by multiple threads resulting in different versions of the object. These versions must be merged together to create a new version. Extending the Abstract Data Type definitions that specify their semantics, MDTs must be mergeable. This means that the MDT must define the semantics of concurrent updates on different versions and the semantics of the merge of two versions.

Which versions must be mergeable? In our case, every thread-local version of an object must be mergeable with the latest global version. On the other hand, local versions from different threads do not have to be mergeable. In our setting, all thread-local versions are branched of the global version and merged back to it. This requirement is in contrast to state-based CRDTs [80, 79] in eventually consistent systems where all versions residing on different replicas must be mergeable.

There are two aspects of mergeability, namely *semantic mergeability* and *structural mergeability.*

Semantic mergeability tackles the semantics of the data type operations with respect to the merge operation; i.e., whether the objects behave "as expected" after merging. Recall the illustrations in Figure 3.3 and Figure 3.2. We do not want to lose the local increments to the counter when merging. Similar to CRDTs, thread-local updates should be reflected semantically in the merged version. A deterministic merge guarantees that any concurrent updates can merge their results and obtain a consistent deterministic state.

Structural mergeability is related to how efficiently two versions can be merged. This property refers to the low-level implementation details of the data structure. Copy-on-write for large data structures such as lists, trees, and sets is not efficient regarding run-time and memory usage. Hence, it is essential to share parts of the data structure from multiple versions and only keep distinct parts which are required to distinguish different versions. Several mechanisms are employed in persistent data structures in functional programs [73]. Our notion of structural mergeability is inspired by these implementations but having an efficient merge as an additional requirement.

3.5.2 A portfolio of MDTs

The implementation of an MDT τ consists of a set of state definitions to define the underlying data structure that captures the state, and a list of method definitions. Each method implements the operations given in the type specification. To specify which abstract operation from the global-local view model it implements, each method definition is annotated using `@` notation.

The state of multi-view data types consists of two parts – an object variable for the local view and another for the global view. Local view and global view may or may not have the same representation. A generic pattern for implementing a mergeable data type MDT is given by the following pseudocode:

```
type MDT = {ThreadLocal T1 localview, T2 globalview}

function OP1(MDT mdt) @weakUpdate op1
   lv := getLocalObject(localview);
   //update lv
end function

function OP2(MDT mdt) @weakRead op2
   lv := getLocalObject(localview);
   //read lv
end function

function OP3(MDT mdt) @strongUpdate op3
   atomic { //update globalview }
end function

function MERGE(MDT mdt) @merge
   lv := getLocalObject(localview);
   atomic {
      merge(globalview, lv)
      //update localview
   }
end function
```

The types of local view and global view (T1, T2) may or may not be the same. Local views are thread-local instances as identified by `ThreadLocal`. A variable specified as `ThreadLocal` exists per thread in the thread's private storage. `getLocalObject` returns the reference to the thread-local object. Many programming languages support some form of thread-local storage (TLS) without the need for explicitly calling `getLocalObject`. A mergeable data type can also implement its thread local storage by mapping thread ids to different instances of the object.

`atomic` refers to any synchronization mechanism such as using a mutex or lock-free techniques such as compare and swap or transactional memory that atomically executes the code block within. OP1, OP2, OP3 refers to the methods implementing the object's update or query operations. Each of them is annotated using `@` to specify if it implements a weakRead, strongRead, weakUpdate or strongUpdate together with the operation defined by its type.

weak operations are executed on the local view. The `ThreadLocal` descriptor guarantees that each thread is accessing its private view.

For some data types, local views are isolated from each other and the global view, by maintaining a full copy of the object in each view. For large data structures, such as list or trees, keeping a full copy is not efficient. Thus the local views may contain references to parts of the data structures that are shared by other local views or global view. In most cases, the shared parts are not directly updated by the weak updates but only read. For example, a *lookUp* on a list may first traverse the locally added items and then the shared parts of the list which are conceptually part of its local view. The mechanisms to make sure that an update on the global view does not change the local views, if it is updating the shared part, depends on the data type semantics and the underlying data structure. We show designs of a few data types where this can be done efficiently and correctly without copying the entire data structure.

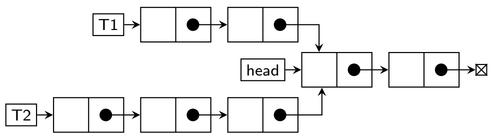
Counter The global view of a mergeable counter is an integer g. The local view consists of a pair of integers (s, l). The weak increments are collected in the variable l and added to g during the merge. This design is inspired by *sloppy counters* [19], while using a local counter per thread instead of per core. Algorithm 3.1 shows the implementation of the mergeable counter.

It is easy to extend this implementation to allow decrements, explicit arguments for increments/decrements, and generalize to other commutative monoids.

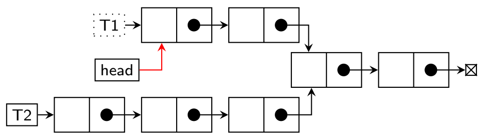
Grow-only bag While the merge operation for the counter can be implemented in a simple and efficient way, we have to employ different strategies for larger, composed data structures such as lists and sets. We adapt techniques that have been developed in the context of persistent data structure [30]. A persistent data structure is a mutable data structure that offers access to multiple versions. This technique is widely used to implement purely functional data structures efficiently, in particular, linked data structures such as lists and trees. When multiple threads modify the data structure, each thread

Algorithm 3.1 Mergeable Counter.

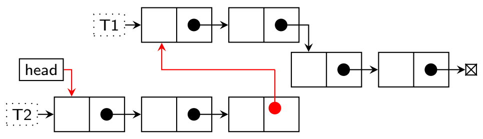
```
type Counter = {int gcntr, ThreadLocal CounterLocal lcntr, Lock l}
type CounterLocal = {int s, int l}

function weakGetValue(Counter cntr) @weakRead getValue
    lcntr := getLocalObject(cntr);   ▷ get reference to the threadlocal object of CounterLocal
    v := lcntr.s + lcntr.l;
    return v;
end function

function weakInc(Counter cntr) @weakUpdate inc
    lcntr := getLocalObject(cntr);   ▷ get reference to the threadlocal object of CounterLocal
    x := lcntr.l + 1;
    lcntr.l := x;
end function

function strongInc(Counter cntr) @strongUpdate inc
    lcntr := getLocalObject(cntr);   ▷ get reference to the threadlocal object of CounterLocal
    lock(cntr.l);
    gcntr := gcntr + 1;
    unlock(cntr.l);
end function

function merge(Counter cntr) @merge
    lcntr := getLocalObject(cntr);   ▷ get reference to the threadlocal object of CounterLocal
    lock(cntr.l);
    gcntr = gcntr + lcntr.l;
    lcntr.l := 0;
    lcntr.s = gcntr;
    unlock(cntr.l);
end function
```



(a) Two threads with different local views.



(b) After T_1's local view is merged.

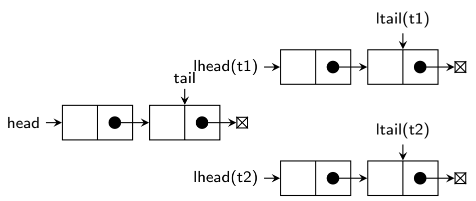


(c) After T_2's local view is merged.

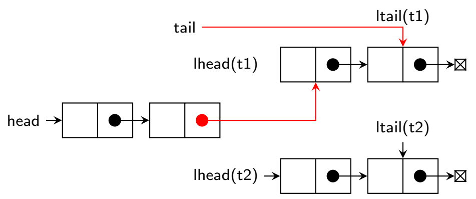
Figure 3.14: Mergeable grow-only bag.

executes updates on a thread-local version of the object, without the need for copying the entire data structure into thread-local storage. The merge operation is then reduced to adjusting pointers in the local and global version to incorporate the updates in the global version. The merge operation must preserve the semantics of the abstract data type by resolving potential semantic conflicts due to concurrent updates.

A bag is a set that allows duplicate elements. A grow-only bag is a bag that allows only add operation. Here, threads can concurrently add elements without violating its semantical correctness. A grow-only bag is implemented as a persistent linked list. An implementation of the bag is illustrated in Figure 3.14. The head points to the first node of the global version accessible to all threads. Adding an element to the bag adds a new node at the head of the linked list local to the thread. This results in a multi-headed list. Figure 3.14a shows the bag after threads T_1 and T_2 have added two and three elements, respectively, and before merging. The list pointed to by T_1 represents the view of the bag to thread T_1, similarly for T_2. Both versions share the nodes of the elements that have been added before the threads started. A lookup that traverses the list starting from the local head will never see an item that is concurrently added or merged. When merging T_1, it updates the global head to point to T_1 (Figure 3.14b). When merging T_2, it has to update both the global head and the local tail of T_2 to include changes of T_1 in the merge (Figure 3.14c). The merge of an add-only bag is efficient because it requires manipulation of only two pointers.



(a) Two threads with local unmerged enqueues.



(b) Thread 1 merges its local queue.

Figure 3.15: Hybrid Mergeable Queue.

Queue A hybrid mergeable queue can be implemented using a singly-linked list similar to a linearizable queue. The global view consists of a singly-linked list where the items enqueued are added to the tail of the list and dequeue is performed from the head. The local view also consists of a singly-linked list, which collects the items enqueued by the thread that are not yet merged (See Figure 3.15a). A mergeable queue instance contains a global view – (`head, tail`), which points to the head and tail nodes respectively of the global list and local view – (`ThreadLocal lhead, ThreadLocal ltail`), which are the head and the tail of the local list of each thread. The merge atomically appends the local list to the global list (Figure 3.15b). The time needed to merge a group of nodes is the same as the time needed to enqueue a single node. By batching the enqueues, we can reduce the number of synchronization operations, thus improving the overall throughput.

The *dequeue* operation directly updates the shared part of the list. For some data types, an update on the shared part of the data structure should preserve the old version, because local views may be keeping a reference to it. However, there is no weakRead, such as a weak lookup, defined on a queue that must observe a version before a concurrent dequeue. Hence, there is no need to keep those versions, which simplifies the implementation.

Add-wins set Add-wins set (AW-set) is a set in which if there are concurrent add and remove operation of the same item, the resulting set after the merge contains the item. i.e. in a concurrent add and remove, add wins. In this section, we discuss the implementation of an add-wins set. Unlike the other types that we have seen in this chapter, AW-set has conflicting operations that need to be resolved during the merge.

The AW-set is implemented using a binary search tree as the underlying data structure. A node of the binary search tree stores one item that was added. No two nodes have the same item. Similar to the bag, the thread-local views and the global view share a single tree structure. In order to allow multiple-local versions and global version to share the internal binary-search tree structure, each node keeps meta-information about the versions of the set that the item belongs to.

The structure of the AW-set is defined in Algorithm 3.2. The complete algorithm implementing the AW-set operations is given in Algorithm 3.3 - 3.6.

Algorithm 3.2 AW-set: state definition

```
1: type GSet = {Node root, Vid vid}
2: type LSet = {Node root, Vid vid, List(<Node, Val>) added,
   List(<Node, Vid>) removed}
3: type Node = {Val item, List(VersionInfo) vinfo, Node
   left, Node right}
4: type Vid = int
5: type VersionInfo = {Vid first, Vid last, Vid removed}
6: type AWSet = {GSet setg, ThreadLocal LSet setl}
```

The global view `setg` consists of a pointer to the root of the binary search tree and a version id `vid` that denotes the version number generated by the last merge operation. The local view `setl` consists of a pointer to the root of the tree and version id that denotes the version of the global view it has observed when it last synchronized with the global view, either by a merge or pull operation. The local view also keeps a information about the items that are added and removed locally, but not yet merged.

A nodes `VersionInfo` consists of three elements: `first` denotes the version when the item was first added, `last` denotes the version id when the item was last added, `removed` denotes when the item was removed. An item is contained in all versions between `first` and `removed`. When an item is re-added after a removal, a new record of `VersionInfo` is added to the list.

A lookup (Algorithm 3.3) on the thread local view returns true if item is added locally or the snapshot version `setl.vid` contains the item. When the item is not present in the local added list, lookup traverses the tree to find the node containing the item. When the node is present, it iterates over the version info list to find one record with $\texttt{first} \leq \texttt{setl.vid} < \texttt{removed}$. If such a record is present, the version `setl.vid` contains the item, and the lookup returns true.

Algorithm 3.3 AW-set: pull and lookup (Algorithm 3.2 continued).

```
/* AW-set functions*/
function PULL(AWSet set)
    setl := getLocalObject(set.setl);
    setl.root := set.setg.root;
    setl.vid := setg.vid ;
    setl.localadded := [];
    setl.localremoved := [];
end function

function LOOKUP(AWSet set, Val item) @weakRead
lookup(item)
    setl := getLocalObject(set.setl);
    if item is in setl.removed then
        x:= false
    else if item is in setl.added then
        x:= true
    else
        node := FINDNODE(item, setl.root)
        if node = $null$ then
            x := false
        else
            x := false
            k := node.vinfo.length()
            while k $> 0 \wedge$ x = false do
                k:=k-1
                v := node.vinfo[k]            ▷ item at index k
                if  v.removed = 0 $\wedge$ v.first $\leq$ setl.vid then
                    x := true;
                else if v.first $\leq$ setl.vid $<$ v.removed then
                    x := true;
                else if setl.vid $>$ v.first then
                    k := 0
                end if
            end while
        end if
    end if
    return x;
end function

```

Algorithm 3.4 AW-set: weak add and remove (Algorithm 3.2 continued).

```
45: function ADD(AWSet set, Val item) @weakUpdate add(item)
46:    setl := getLocalObject(set.setl);
47:    p := FINDPARENT(item, setl.root) ;
48:    LISTADD(⟨item, p⟩, setl.added);
49:    DELETEREMOVED(item, setl.removed)
50: end function
51:
52: function REMOVE(AWSet set, Val item) @weakUpdate remove(item)
53:    setl := getLocalObject(set.setl);
54:    p := FINDNODE(item, setl.root) ;
55:    if p ≠ null then
56:       LISTADD(⟨p, setl.vid⟩, setl.removed);
57:    end if
58:    DELETEADDED(item, setl.added);
59: end function
```

When a thread executes a weak add operation (Algorithm 3.4), it traverses through the tree to find a potential parent node where the item can be inserted. It records this information in the list `setl.added`. During the merge, it can start traversing from this node rather than from the root.

When a thread executes a weak remove operation (Algorithm 3.4), it traverses through the tree to find the node that the item belongs to. It records the reference to the node in the list `setl.removed`.

The merge (Algorithm 3.5) acquires a lock on the tree, and generates a new version number by incrementing the current version number. Each merge thus produces a monotonic version number. It uses this version number to mark if an item was added or removed during this merge operation.

The merge first merges the local adds (Lines 66 - 89). For each added item, it tries to insert the item as the child of the potential parent which was recorded during the weak add operation. If the potential parent already has children, it traverses down the tree to find the right place to insert the item. If a node corresponding to the item already exists, it only updates the version info of that node `vinfo.last` to the new version number. Otherwise, it inserts a node as a leaf. The property of binary search tree guarantees that there is a unique node corresponding to an item.

During the merge of removed items, if there was any concurrent add of the same item, (i.e if `vinfo.last` $>$ the snapshot that the thread observed `setl.vid`), then the remove operation takes no effect. Otherwise, the item

Algorithm 3.5 AW-set: merge (Algorithm 3.2 continued).

```
function MERGE(AWSet set) @merge
   setg := set.setg;
   setl := getLocalObject(set.setl);
   lock setg
   newvid := setg.vid+1;
   /* Merge adds */
   for $\langle$item,parent$\rangle$ in setl.added do
      if setg.root = null then
         n := NEWNODE(item)
         LISTADD(n.vinfo, $\langle$ newvid, newvid, 0 $\rangle$)
         setg.root := n
      else
         if parent = null then
            parent := setg.root
         end if
         n := FINDNODE(item, parent)
         if n $\neq$ $null$ then
            if LASTNODE(n.vinfo).removed $>$ 0 then
               LISTADD(n.vinfo, $\langle$newvid, newvid, 0$\rangle$)
            else
               LASTNODE(n.vinfo).last := newvid
            end if
         else
            n := NEWNODE(item)
            LISTADD(n.vinfo, $\langle$newvid, newvid, 0$\rangle$)
            INSERTNODE(n, parent)
         end if
      end if
   end for

   /* Merge removes */
   for $\langle$n,obvid$\rangle$ in setl.removed do
      v := LASTNODE(n.vinfo)
      if v.removed $= 0$ $\wedge$ v.last $\leq$ obvid then
         v.removed := newvid
      end if
   end for
   setg.vid := newvid
   unlock setg
end function
```

Algorithm 3.6 AW-set: Auxiliary functions (Algorithm 3.2 continued).

```
/* Auxiliary functions */
function FINDPARENT(VAL item, NODE root): NODE
    //starts traversal from node 'root', use binary search tree traversal
    //returns the node which will be the parent of 'item' if it is added, or null if root is null
end function

function FINDNODE(VAL item, NODE root): NODE
    //starts traversal from node 'root', use binary search tree traversal
    //returns the node which contains 'item' or null if the node doesnot exist.
end function

function LISTADD(x, LIST list)
    //Append x to list
end function

function DELETEADDED(VAL item, LIST list)
    //delete ⟨item, _⟩ from list
end function

function DELETEREMOVED(VAL item, LIST list)
    //remove ⟨node, _⟩ from list, where node contains item
end function

```

is marked as removed by setting the `vinfo.removed` to the new version number.

3.6 Correctness

The implementations of the operations of an MDT have more fine-grained steps for each atomic operation in the abstract model, resulting in executions with steps of an operation from a thread interleaved with the steps of operation from other threads. In the presence of the interleaved steps, the implementation must guarantee the semantics of the global-local view model as defined by its abstract execution. In this section, we outline a proof method to verify the implementations of *purely mergeable* MDTs. We refrain from proving the correctness of hybrid MDTs.

Concrete and Abstract programs A concrete program $\mathcal{C}_p$ is an $\mathscr{A}_{\text{MDT}}$ program that consists of MDT implementations in addition to the other state and method definitions. A concrete program does not contain any of the statements from Stmt_{gl}, i.e., it does not have any of the abstract operations defined by the global-local view model. A concrete program is valid only if it accesses the instances of multi-view types τ only through the method calls defined in its implementation.

An abstract program $\mathcal{A}_p$ corresponding to a concrete program $\mathcal{C}_p$ is obtained by replacing the method calls defined in the MDT implementation with the corresponding operations from the global-local view model. Every method definition in an MDT implementation is annotated with its corresponding operation in the global-local view model (Section 3.5.2). Thus, there is a one-to-one translation from a concrete program to its corresponding abstract program.

For example, Algorithm 3.7 shows a concrete program that accesses a shared counter whose implementation is given in Algorithm 3.1. Algorithm 3.8 shows its corresponding abstract program. Each method call (such as WEAKINC, MERGE) in the concrete program is replaced by the equivalent abstract operations (such as **weakUpdate** inc, merge), with out modifying any other expressions or statements in the program.

Algorithm 3.7 Concrete program for a shared counter.

```
c := new Counter;
fork (c) {
    WEAKINC(c);
    WEAKINC(c);
    MERGE(c);
};
fork (c) {
    INC(c);
    b := WEAKGETVALUE(c);
    MERGE(c);
};
```

Concrete and Abstract execution An execution is a sequence of steps of the form

$$\Pi; \Gamma \rightarrowtail \Pi'; \Gamma' \rightarrowtail \Pi''; \Gamma'' \rightarrowtail ...$$

Algorithm 3.8 Abstract program for a shared counter.

```
c := new Counter;                          ▷ instantiate a counter
fork (c) {
    weakUpdate inc c;
    weakUpdate inc c;
    merge c;
};
fork (c) {
    weakUpdate inc c;
    b := weakRead getvalue c;
    merge c;
};
```

obtained by evaluating the program as defined by the operational semantics of $\mathscr{A}_{\text{MDT}}$. $\mathcal{E}_{\mathscr{A}_{\text{MDT}}}(\mathcal{A}_p)$ denotes the set of all valid executions of $\mathcal{A}_p$ according to the reduction steps defined in Section 3.4.2. $\mathcal{A}_{\mathcal{E}}$ denotes a valid abstract execution. Similary, $\mathcal{C}_{\mathcal{E}}$ denotes a valid concrete execution of a concrete program.

Correctness The implementation of an MDT is correct if the concrete programs using the MDT do not have any additional observable behavior compared to the corresponding abstract program. In our case, the observable behavior is the return value of a data type operation. To capture the return values to be used in the verification, we use an auxiliary variable rval. The read operations (`weakRead`) in the abstract and concrete program are annotated with the assignment to the rval.

We establish a simulation ensuring that for every externally observable behavior made by the concrete program execution, there is a corresponding one in the abstract execution. To that end, we define the equivalence relation between $\mathcal{C}_{\mathcal{E}}$ and $\mathcal{A}_{\mathcal{E}}$ in the following way.

- An Inv relation relates the concrete low-level representation of the data structure to the corresponding abstract representation.
- Starting from Inv-related states, each step of the concrete program corresponds to zero or one step of the abstract program and resulting in states that are again Inv-related.
- rval captures the observable behavior in abstract and concrete execution. At each step, the value of rval in abstract state and concrete state must be same.

The abstract and concrete states, $\mathcal{A}_{state}$ and $\mathcal{C}_{state}$, are obtained from global heap (Γ), thread-local heaps (Λ) and stacks (Θ) of the abstract execution and the concrete execution, respectively.

Definition 3.1. A concrete execution is equivalent to an abstract execution, $\mathcal{C}_{\mathcal{E}} \cong \mathcal{A}_{\mathcal{E}}$, iff for each step of concrete execution,

$$\Pi_c\{t \mapsto \langle e, \Theta_c, \Lambda_c\rangle\}; \Gamma_c \rightarrowtail \Pi_c\{t \mapsto \langle e', \Theta'_c, \Lambda'_{c'}\rangle\}; \Gamma'_c$$

if $\textsc{Inv}(\mathcal{A}_{state}, \mathcal{C}_{state})$ holds before the step, then there exists a step in the abstract execution

$$\Pi_a\{t \mapsto \langle e, \Theta_a, \Lambda_a\rangle\}; \Gamma_a \rightarrowtail \Pi_a\{t \mapsto \langle e', \Theta'_a, \Lambda_a\rangle\}; \Gamma'_a$$

or an empty transition

$$\Pi_a\{t \mapsto \langle e, \Theta_a, \Lambda_a\rangle\}; \Gamma_a \overset{\varepsilon}{\rightarrowtail} \Pi_a\{t \mapsto \langle e, \Theta_a, \Lambda_a\rangle\}; \Gamma_a$$

such that $\textsc{Inv}(\mathcal{A}'_{state}, \mathcal{C}'_{state}) \wedge \Gamma_a(\text{ rval }) = \Gamma_c(\text{ rval })$ holds after each step.

Definition 3.2. A concrete program $\mathcal{C}_p$ is *equivalent* to an abstract program $\mathcal{A}_p$, if for every concrete execution $\mathcal{C}_{\mathcal{E}} \in \mathcal{E}_{\mathscr{A}_{\text{MDT}}}(\mathcal{C}_p)$, there exists an $\mathcal{A}_{\mathcal{E}} \in \mathcal{E}_{\mathscr{A}_{\text{MDT}}}(\mathcal{A}_p)$ such that $\mathcal{C}_{\mathcal{E}} \cong \mathcal{A}_{\mathcal{E}}$.

Definition 3.3. An MDT implementation is correct if all concrete programs using the MDT are equivalent to their corresponding abstract programs.

Thus, to prove the correctness of an MDT implementation, we proceed as follows.

1. We define an $\textsc{Inv}$ relation between the abstract and concrete state.
2. We show that the concrete implementation of an operation is correct with respect to the abstract operation semantics using the simulation.
3. The challenge here is to show that the interleaving of the steps from operations in concurrent threads does not affect the correctness of this operation. To prove the correctness of one operation, we assume that no other concurrent step violates $\textsc{Inv}$. Then, we show that this assumption is valid for each operation.

The simulation, thus, shows that for all executions of a concrete program, there is an equivalent abstract execution. Thus we can show that the given implementation of the MDT is correct.

Additional Definitions. We assume the program that we are verifying accesses only one shared object referenced by r.

Given the global and local memories of the abstract execution $(\Gamma_a, \forall t : \Lambda_a^t)$, the state of an abstract execution is defined as

$$\mathcal{A}_{state} = (\sigma_g, L_a)$$

where $\sigma_g = \Gamma_a(r)$ and $L_a[t] = \Lambda_a^t(r)$. $L_a[t]$ is a tuple of sequence of updates $\langle \sigma_s, \sigma_l \rangle$.

Similarly, given the global and local memories of the concrete execution $(\Gamma_c, \forall t : \Lambda_c^t, \Theta_c^t)$, we define the state of a concrete execution as

$$\mathcal{C}_{state} = (g_c, L_c)$$

where $g_c = \Gamma_c(r)$ is the global view and $L_c[t] = \Gamma_c(lref(r,t))$ refers to the local copies. $lref(r,t)$ is the dereferencing function that yields the reference to a local copy.

We also annotate each operation in a sequence σ_g (and σ_s) with a version number. Two operations have the same version number iff they were merged in the same merge operation. A version of a state is defined as the version of the last operation in sequence $v(\sigma \cdot op) = v(op)$.

It follows from the abstract execution model that the local snapshots (σ_s) are always a prefix of the global snapshot. Hence $\forall \langle \sigma_s, \sigma_l \rangle \in L_a, \sigma_s \preceq \sigma_g \wedge v(\sigma_s) \leq v(\sigma_g)$.

3.6.1 Verification of mergeable counter

The concrete state of a counter is $\mathcal{C}_{state} = (gcntr, L_c)$ where $gcntr$ is an integer, and L_c is the set of thread-local counter objects.

Given the concrete state of the counter as

```
type Counter = {int gcntr, ThreadLocal CounterLocal lcntr}
type CounterLocal = {int s, int l}
```

we define the invariant $\textsc{Inv}(\mathcal{A}_{state}, \mathcal{C}_{state})$ as follows:

$$\begin{aligned}
\textsc{Inv}_g &:= gcntr = |\sigma_g| \\
\textsc{Inv}_l(t) &:= L_a[t] = \langle \sigma_s, \sigma_l \rangle \wedge L_c[t] = lcntr \\
&\qquad \iff lcntr.s = |\sigma_s| \wedge lcntr.l = |\sigma_l| \\
\textsc{Inv}(\mathcal{A}_{state}, \mathcal{C}_{state}) &:= \big(unlocked(gcntr) \implies \textsc{Inv}_g \wedge \forall t : \textsc{Inv}_l(t)\big) \\
&\qquad \wedge \big(locked(gcntr) \wedge lock\_owner(gcntr) \neq t \implies \textsc{Inv}_l(t)\big)
\end{aligned}$$

To simplify the predicates, we assume that we can only increment a counter, but not decrement it[1]. The predicate $locked(gcntr)$ yields $true$ iff a thread has acquired the lock on $gcntr$ and $unlocked(gcntr)$ yields $true$ iff no thread has acquired the lock. $lock\_owner(gcntr)$ returns the thread id of the thread that has acquired the lock.

Algorithm 3.1 shows the implementation of the counter. From the abstract execution model, we can assume the following:

- Two threads cannot concurrently invoke INC/GETVALUE on the same $lcntr$, because $lcntr$ is a thread-local object.
- Two threads can concurrently execute MERGE. However, in the given implementation, the merge operation is protected using locks. Hence, we can ignore the effects of the interleaving of steps from concurrent merges.

This simplifies the proof because we only need to show that for each operation (inc, value, merge), there is a valid simulation between an abstract execution and the concrete execution as defined in Definition 3.1. That is we need to show that the invariant INV is maintained after each step in the operation. (Note that we use INV without the parameters $\mathcal{A}_{state}$ and $\mathcal{C}_{state}$ to make the predicates less verbose.)

getValue: This is a read-only operation. Hence, we have to show that the value returned is the same as for a `weakRead` $getValue\ r$. Let us assume that INV holds before the start of the execution of this operation. Then, we know that $lcntr.s = |\sigma_s| \ \wedge\ lcntr.l = |\sigma_l|$. Thus $v = |\sigma_s| + |\sigma_l|$ is the return value in both abstract and concrete execution.

increment: First, we assume that INV holds before the start of the increment operation. We need to show that INV holds after each step and at the end of the execution. Following Hoare-style notation, we can annotate the implementation of INC with the predicates that hold before and after each step. t denotes the thread-id of the thread that is executing this operation.

$\textsc{Inv} \wedge L_a[t] = \langle \sigma_s, \sigma_l \rangle \wedge lcntr.l = |\sigma_l|$

```
x := lcntr.l + 1;
```

$\textsc{Inv} \wedge x = |\sigma_l| + 1$

[1]We can extend the invariant with decrement operation by defining the equivalence relation between σ and an integer n as, $n =$ (the number of increments in σ- the number of decrements in σ). This does not modify the proof method discussed in this section.

```
lcntr.l := x; weakUpdate inc cntr
```
$lcntr.l = |\sigma_l| + 1 \wedge L_a[t] = \langle \sigma_s, \sigma_l \cdot inc \rangle \wedge \textsc{Inv}$

At the second step, where x is assigned to lcntr.l, the abstract program also executes one increment operation modifying the predicate to $L_a[t] = \langle \sigma_s, \sigma_l \cdot inc \rangle$ after this step. Thus the invariant holds after the step. As discussed before, no other concurrent threads modifies `lcntr.l` or `lcntr.s`. Hence the correctness of this operation is not affected by the presence of any concurrent steps.

merge: Similarly, we need to show that Inv holds after each step in the merge. Assuming Inv holds before the merge, we can annotate the code for merge with the predicates that hold before and after each step. The predicate $unlocked(gcntr) \wedge \textsc{Inv}$ indicates that immediately before the successful execution of `lock gcntr`, `gcntr` is unlocked and the invariant holds.

$unlocked(gcntr) \wedge \textsc{Inv}$
```
lock gcntr;
```
$locked(gcntr) \wedge \textsc{Inv} \wedge L_a[t] = \langle \sigma_s, \sigma_l \rangle \wedge gcntr = |\sigma_g| \wedge lcntr.l = |\sigma_l|$
```
gcntr = gcntr + lcntr.l;
```
$locked(gcntr) \wedge \textsc{Inv} \wedge gcntr = |\sigma_g| + |\sigma_l|$
```
lcntr.l := 0;
lcntr.s = gcntr;
unlock gcntr; merge cntr
```
$unlocked(gcntr) \wedge \mathcal{A}_{state} = \langle \sigma_g \cdot \sigma_l, \_ \rangle \wedge L_a[i] = \langle \sigma_g \cdot \sigma_l, \emptyset \rangle \wedge$
$gcntr = lcntr.s = |\sigma_g \cdot \sigma_l| \wedge lcntr.l = |\emptyset| \wedge \textsc{Inv}$

Thus we proved that the implementation of the mergeable counter as given in Algorithm 3.1 is correct.

3.6.2 Verification of AW-set

The implementation of AW-set consists of a binary search tree whose nodes are in the global heap. The tree can be accessed and updated by all threads. Each thread has a local object that has references to the shared tree in the global heap. Thus the implementation of AW-set differs from that of the counter in that the counter has no references to the shared object in its local view. The proof for the correctness of the AW-set is hence more complex.

The concrete state of the tree is given by:

```
type GSet = {Node root, Vid vid}
type LSet = {Node root, Vid vid, List(<Node, Val>) added,
   List(<Node, Vid>) removed}
```

```
type Node = {Val item, List(VersionInfo) vinfo, Node left,
   Node right}
type Vid = int
type VersionInfo = {Vid first, Vid last, Vid removed}
type AWSet = {GSet setg, ThreadLocal LSet setl}
```

The binary search tree that stores the item in the set is shared by the global view `setg` and the local objects `setl` (of each thread). The full pseudocode of the operations is given in Section 3.5.2.

We can assume the following:

- Two threads can concurrently invoke the merge function. However, the merge operation is protected using locks. Hence, we can ignore the interleaving of steps from two concurrent merges.
- Concurrent lookup and merge can happen. While lookup is traversing the tree, the concurrent merge can modify it. We must prove that lookup returns a correct result even in the presence of a concurrent merge. To show that lookup returns the correct result, we assume that Inv is maintained at each step. Later, we prove that the merge does not violate the invariant.
- The update operations, add and remove, traverse the shared tree, but never modify it. They only modify the thread-local objects, which are not shared with other threads. Hence the update operations do not affect any concurrent lookups and update operations in other threads. A concurrent merge may affect the state observed during the traversal in add and remove operations.
- A node is inserted into a binary search tree only during the merge. By the property of the binary search tree, there is a unique node for an item that can be found by the traversal on the tree. We do not prove this property but assume this from the underlying tree implementation.

Definitions Similar to the counter, abstract state $\mathcal{A}_{state}$ consists of global state σ_g and a local state $L_a[t]$ per thread t. To simplify the notations in the predicates, we use $(\bar{\sigma_s}[t], \bar{\sigma_l}[t])$ to represent the local state $L_a[t]$. We annotate each operation in σ_g and $\bar{\sigma_s}[t]$ with a version number. Two operations have the same version number when they were merged in the same merge operation. The operations in σ_l do not have version numbers. A version of a snapshot is defined as the version of the last operation in the sequence. $v(\sigma \cdot op) = v(op)$. The local snapshot $\bar{\sigma_s}[t]$ is always a prefix of the global state σ_g.

Concrete state $\mathcal{C}_{state}$ consists of global object *setg* and for each thread t, a local view *setl*.

Proof We define an Invariant INV which is a predicate over the abstract state and concrete state. First, we assume that INV holds at every point in the concrete execution. With that assumption, we can prove that any read operation (here lookup) returns the correct result. Then we prove that no step in the concrete execution of the operations that modify the state violates INV.

Figure 3.16 shows the invariants defined for the AW-set.

The invariant INV consists of two predicates INV_g, a predicate over global states σ_g and *setg*, and INV_{ls}, a predicate over local states. $\text{INV}_{1..5}$ are parameterized with σ and *root*, which are instantiated in INV_g and INV_{ls}.

Informally, $\text{INV}_1(\sigma, root)$ and $\text{INV}_{1a}(\sigma, root)$, establish a relation between existing nodes in the tree in the concrete state to the corresponding operations in the abstract state. It says, if there is a *node* for item e in the tree representing the set, then for every version info $\{f, l, r\}$ of *node*, there exists an $add(e)$ operation at version f and version l in the abstract state, and if $r = 0$, then there is no $rem(e)$ after version l. If $r \neq 0$, there is $rem(e)$ at version r.

Similarly, INV_2 and INV_3 say that, if there is an $add(e)$ or a $rem(e)$ in the abstract state, then there is a node with item e and version info $\{f, l, r\}$ corresponding to those operations. INV_4 defines a property of the version info list of a node.

Correctness of lookup Algorithm 3.9 shows the lookup function with annotated predicates that hold before and after each step. We assume that INV holds before the start of the lookup and after each step. Later we show that the operations that modify the state maintains the INV after each step. Thus, the concurrent operations do not affect the correctness of the lookup. Since lookup does not modify the state, it does not violate the invariant INV.

setl represents the local view of the current thread t that executes the operation.

By INV we know that $setl.vid = v(\sigma_s)$. We have to show that $P(\sigma_s, \sigma_l) \iff x = true$, where x is the result of the `lookup(e)` and

$$P(\sigma_s, \sigma_l) \triangleq \big(\exists m_1 \in \sigma_s \cdot \sigma_l : m_1 = add(e) \wedge \\ \nexists m_2 \in \sigma_s \cdot \sigma_l : m_2 = rem(e) \wedge m_1 \rightsquigarrow m_2\big)$$

$$
\begin{aligned}
\textsc{Inv}_1(\sigma, \mathit{root}) \triangleq\ & \forall \mathit{node} \lhd \mathit{root} : \mathit{node.item} = e \implies \\
& \forall \{f, l, r\} \in \mathit{node.vinfo} : \\
& f \le l \land (r = 0 \lor l < r) \land \\
& \quad \big(f \le v(\sigma) \implies \exists m_1 \in \sigma : f = v(m_1) \land m_1 = \mathit{add}(e)\big) \land \\
& \quad \big(l \le v(\sigma) \implies \exists m_2 \in \sigma : l = v(m_2) \land m_2 = \mathit{add}(e)\big) \land \\
& \quad \big(r = 0 \land f \le v(\sigma) \implies \\
& \qquad \nexists m_3 \in \sigma : f < v(m_3) \le v(\sigma) \land m_3 = \mathit{rem}(e)\big) \land \\
& \quad \big(r \ne 0 \land r \le v(\sigma) \implies \\
& \qquad \exists m_4 \in \sigma : v(m_4) = r \land m_4 = \mathit{rem}(e)\big) \land \\
& \quad \big(r \ne 0 \implies \nexists m_5 \in \sigma : f < v(m_5) < r \land m_5 = \mathit{rem}(e)\big) \\
\textsc{Inv}_{1a}(\sigma, \mathit{root}) \triangleq\ & \exists m_1 \in \sigma : m_1 = \mathit{add}(e) \implies \\
& \quad (\exists \mathit{node} \lhd \mathit{root} : \mathit{node.item} = e \land \\
& \quad \exists \{f, l, r\} \in \mathit{node.vinfo} : f \le v(m_1) \le v(l)) \\
& \qquad \exists m_4 \in \sigma : v(m_4) = r \land m_4 = \mathit{rem}(e)\big) \land \\
& \quad \big(r \ne 0 \implies \nexists m_5 \in \sigma : f < v(m_5) < r \land m_5 = \mathit{rem}(e)\big) \\
\textsc{Inv}_2(\sigma, \mathit{root}) \triangleq\ & \exists m_1 \in \sigma : m_1 = \mathit{add}(e) \implies \\
& \quad (\exists \mathit{node} \lhd \mathit{root} : \mathit{node.item} = e \land \\
& \quad \exists \{f, l, r\} \in \mathit{node.vinfo} : f \le v(m_1) \le v(l)) \\
\textsc{Inv}_3(\sigma, \mathit{root}) \triangleq\ & \exists m_2 \in \sigma : m_2 = \mathit{rem}(e) \implies \\
& \quad (\exists \mathit{node} \lhd \mathit{root} : \mathit{node.item} = e \land \\
& \quad \exists \{f, l, r\} \in \mathit{node.vinfo} : f \le l < r \land r \le v(m_2) \land \\
& \quad \nexists \{f', l', r'\} : l < f' \le v(m_2)) \\
\textsc{Inv}_4(\mathit{root}) \triangleq\ & \forall \mathit{node} \lhd \mathit{root} : i < j \land \{f, l, r\} = \mathit{node.vinfo}[j] \land \\
& \{f', l', r'\} = \mathit{node.vinfo}[i] \implies f' \le l' < r' < f \le l \\
\textsc{Inv}_5(\sigma, \mathit{root}) \triangleq\ & \textsc{Inv}_1(\sigma, \mathit{root}) \land \textsc{Inv}_{1a}(\sigma_c, \mathit{root}) \land \textsc{Inv}_2(\sigma, \mathit{root}) \land \\
& \textsc{Inv}_3(\sigma, \mathit{root}) \land \textsc{Inv}_4(\sigma, \mathit{root}) \\
\textsc{Inv}_g(\sigma_g, \mathit{setg}) \triangleq\ & v(\sigma_g) = \mathit{setg.vid} \land \textsc{Inv}_5(\sigma_g, \mathit{setg.root}) \\
\textsc{Inv}_l(\sigma_l, \mathit{setl}) \triangleq\ & (\forall m \in \sigma_l : m = \mathit{add}(e) \land \nexists m' \in \sigma_l : m' = \mathit{rem}(e) \land \\
& m \rightsquigarrow m' \iff \langle e, \_ \rangle \in \mathit{setl.added}) \land \\
& (\forall m \in \sigma_l : m = \mathit{rem}(e) \land \nexists m' \in \sigma_l, m' = \mathit{add}(e) \land \\
& m \rightsquigarrow m' \iff \langle \mathit{node}, \_ \rangle \in \mathit{set.removed} \land \mathit{node.item} = e) \\
\textsc{Inv}_{ls}(\bar{\sigma_s}, \bar{\sigma_l}, \mathit{setl}) \triangleq\ & \forall t : \mathit{setl}[t].\mathit{vid} = v(\bar{\sigma_s}[t]) \land \\
& \quad \textsc{Inv}_5(\bar{\sigma_s}[t], \mathit{setl}[t].\mathit{root}) \land \textsc{Inv}_l(\bar{\sigma_l}[t], \mathit{setl}[t]) \\
\textsc{Inv}(\mathcal{A}_{state}, \mathcal{C}_{state}) \triangleq\ & \textsc{Inv}_g(\sigma_g, \mathit{setg}) \land \textsc{Inv}_{ls}(\bar{\sigma_s}, \bar{\sigma_l}, \mathit{setl})
\end{aligned}
$$

Figure 3.16: Invariants of AW-set. $\mathit{node} \lhd \mathit{root}$ denotes *node* exists in the subtree with root node *root*. $m \rightsquigarrow m'$ shows that the operation m precedes the operation m' in the given sequence.

We can rewrite the predicate to consider different cases as follows:
When the item was removed in the local updates.

$$P_1(\sigma_s,\sigma_l) \triangleq \exists m_2 \in \sigma_l : m_2 = rem(e) \wedge$$
$$\nexists m_1 \in \sigma_l : m_1 = add(e) \wedge m_2 \rightsquigarrow m_1$$

When the item was added in the local updates.

$$P_2(\sigma_s,\sigma_l) \triangleq \exists m_1 \in \sigma_l : m_1 = add(e) \wedge$$
$$\nexists m_2 \in \sigma_l : m_2 = rem(e) \wedge m_1 \rightsquigarrow m_2$$

When the item was neither added nor removed in the local updates, then lookup the item in the snapshot.

$$P_3(\sigma_s,\sigma_l) \triangleq \exists m_1 \in \sigma_s : m_1 = add(e) \wedge$$
$$(\nexists m_2 \in \sigma_s : m_2 = rem(e) \wedge v(m_1) < v(m_2)$$

$$P(\sigma_s,\sigma_l) \iff x = true \cong$$

$$(P_1(\sigma_s,\sigma_l) \implies x = false) \wedge (P_2(\sigma_s,\sigma_l) \implies x = true) \wedge$$
$$((\neg P_1(\sigma_s,\sigma_l) \wedge \neg P_2(\sigma_s,\sigma_l) \implies (P_3(\sigma_s,\sigma_l) \iff x = true)))$$

Case 1: Algorithm 3.9 Line 2-3. By $\textsc{Inv}_l(\sigma_l, setl)$, P_1 implies the item is in `setl.removed`. Thus $P_1 \implies x = false$.

Case 2: Similarly, following $\textsc{Inv}_l(\sigma_l, setl), P_2$ implies the item is in `setl.added`. Thus $P_2 \implies x = true$.

Case 3: At line 7 $(\neg P_1 \wedge \neg P_2)$ holds. Hence, we need to show $P_3 \iff x = true$. When there is no node for an *item*, the lookup returns false (Line 11). By $\textsc{Inv}_2$

$$\nexists node : node.item = item \implies \neg P_3$$

Case 4: At line 17, by $\textsc{Inv}_2(\sigma_s, setl.root) \wedge \textsc{Inv}_3(\sigma_s, setl.root)$, the following predicate holds.

$$(\exists m_1 = add(e) \in \sigma_s \implies \exists \{f,l,r\} \in node.vinfo[0..n-1]$$
$$\wedge\, f \le v(m_1) \le l) \quad \wedge$$
$$(\exists m_2 = rem(e) \in \sigma_s \implies \exists \{f,l,r\} \in node.vinfo[0..n-1]$$
$$\wedge\, l < r \le v(m_2))$$

Algorithm 3.9 Lookup operation annotated with predicates. We assume Inv holds at each step, but not included in the predicates.

```
function LOOKUP(AWSet set, Val item)
   setl := getLocalObject(set.setl);
   if item is in setl.removed then
      x:= false
   else if item is in setl.added then
      x:= true
   else
      node := FINDNODE(item, setl.root)
      if node = null then
         $\nexists node : node.item = item$
         x := false
         $\neg P_3 \wedge x = false$
      else
         x := false
         k := node.vinfo.length()
         $\neg B_1 \wedge \neg B_2 \wedge x = false$
         while k $> 0 \wedge$ x = false do
            $\neg B_1 \wedge \neg B_2 \wedge x = false$
            k:=k-1
            v := node.vinfo.at(k)          ▷ item at index k
            if  v.removed = 0 $\wedge$ v.first $\leq$ setl.vid then
               x := true;
               $B_1 \wedge x = true$
            else if v.first $\leq$ setl.vid $<$ v.removed then
               x := true;
               $B_2 \wedge x = true$
            else if setl.vid $>$ v.first then
               $\neg(B_1 \vee B_2)$
               k := 0
               $\neg(B_1 \vee B_2) \wedge k = 0 \wedge x = false$
            end if
            $(B_1 \vee B_2) \iff x = true$
         end while
         $(B_1 \vee B_2) \iff x = true$
      end if
      $(\nexists node : node.item = item \wedge x = false \wedge \neg P_3) \vee$
      $\exists node : node.item = item \wedge ((B_1 \vee B_2) \iff x = true))$
   end if
   return x;  rval = weakRead lookup e
end function
$P \iff x = true$
```

The above condition is maintained for any concurrent modification because we assume Inv holds at each step. Thus we can safely iterate over the list of vinfo.

Let

$$B_1 \triangleq \exists\{f, l, r\} \in node.vinfo[k..N-1] : r = 0 \land f \leq vid$$

$$B_2 \triangleq \exists\{f, l, r\} \in node.vinfo[k..N-1] : f \leq vid < r$$

We can show that $k = 0 \implies (B_1 \lor B_2 \iff P_3)$.

By $\textsc{Inv}_1$:

$$\begin{aligned} B_1 \implies &\exists m_1 : m_1 = add(e) \land v(m_1) \leq v(\sigma_s) \land \\ &\nexists m_2 : m_2 = rem(e) \land v(m_1) < v(m_2) \leq v(\sigma_s) \end{aligned}$$

Thus $B_1 \implies P_3$. Similarly by $\textsc{Inv}_1$ $B_2 \implies P_3$.

To prove $P_3 \implies B_1 \lor B_2$, we show that $\neg(B_1 \lor B_2) \implies \neg P_3$

$$\begin{aligned} \neg(B_1 \lor B_2) &\implies \nexists\{f, l, r\} : (r = 0 \land f \leq vid) \lor (f \leq vid < r) \\ &\implies \nexists\{f, l, r\} : (r = 0 \land f \leq vid) \lor \\ &\qquad (r \neq 0 \land v \leq vid \land vid < r) \end{aligned}$$

By $\textsc{Inv}_2 \land \textsc{Inv}_3$

$$\neg(B_1 \lor B_2) \implies \neg P_3$$

Algorithm 3.9 shows the predicates that hold after each step of execution in the lookup operation. At the end of the execution of the while loop, it has been shown that $B_1 \lor B_2 \iff x = true$. Thus we show that $P \iff x = true$. We consider the execution point of the lookup operation in the abstract execution at line 39, where the result x is returned.

Thus we proved that the correctness of lookup operation. Next, we show that the operations that modify the states, such as add, remove and merge, maintain the invariant Inv at every step in their concrete execution.

Correctness of merge The merge operation is executed by a thread to merge its local operations to the global set. While the thread executes merge, the other threads can execute any operation concurrently, including merge. However, as shown in Algorithm 3.10, the thread acquires the lock on global set before executing the steps in the merge. Hence we can ignore the interleaving of two concurrent merges.

However, the merge is modifying the shared tree, while other threads concurrently execute a lookup or traversal on the tree. Hence we must show that $\textsc{Inv}$ is maintained by each step in the merge. Since merge does not modify the local objects (*setl.added* and *setl.removed*) of the other threads, we have to consider only the predicates of $\textsc{Inv}$ that affect the snapshots (σ_s and the shared tree). Thus, we show that $\textsc{Inv}_g \wedge \forall t : \textsc{Inv}_5(\sigma_s[t], \mathit{setl}[t])$ is maintained. Since no other thread modifies the global state concurrently, we can safely assume that this predicate is true at the start of the *merge*, and is not affected by the concurrent threads.

There are four lines, where the tree is modified during the merge of adds.

Case 1: *root* is null. Hence, by $\textsc{Inv}_2$ the tree is empty. Thus all existing snapshots ($\forall t : \sigma_s[t]$) are empty. At line 11, after a new node is added with vinfo $\{\mathit{newvid}, \mathit{newvid}, 0\}$, $\forall t : \textsc{Inv}_1(\sigma_s[t], \mathit{setl}[t])$ remains unchanged, because $\mathit{newvid} > v(\bar{\sigma}_s[t])$.

Case 2: There exists a *node* for *item*. At line 21, a new tuple is added to $vinfo$. Since the existing vinfo tuples are not modified, $\forall t$: $\textsc{Inv}_2(\sigma_s[t], \mathit{setl}[t]) \wedge \textsc{Inv}_3(\sigma_s[t], \mathit{setl}[t])$ is maintained. Since $\mathit{newvid} > v(\bar{\sigma}_s[t])$, $\textsc{Inv}_1(\bar{\sigma}_s[t], \mathit{setl}[t])$ is also maintained.

Case 3: There exists a *node* for *item*. At line 24, an existing vinfo tuple $\{f, l, 0\}$ is modified to $\{f, \mathit{newvid}, 0\}$. By $\textsc{Inv}_1$, $\{f, l, 0\}$ indicates that there exists an operation $\mathit{add}(\mathit{item})$ at version l. Since $\mathit{newvid} > l$, we can see that $\textsc{Inv}_2$ is maintained. Other existing vinfo tuples are not modified. Hence $\textsc{Inv}_3$ and $\textsc{Inv}_1$ are maintained.

Case 4: A *node* for the *item* does not exist. Thus by $\textsc{Inv}_2$, there is no $\mathit{add}(e)$ operation in any snapshot σ_s, where $v(\sigma_s \leq v(\sigma_g))$. Adding a new node with vinfo $\{\mathit{newvid}, \mathit{newvid}, 0\}$ does not affect the invariants on existing snapshots because $\mathit{newvid} > v(\sigma_s[t])$.

In all the 4 cases, we can see that $\textsc{Inv}_4$ is also maintained.

Similarly, during the remove, the only modification is to update the last vinfo tuple $\{f, l, 0\}$ to $\{f, l, \mathit{newvid}\}$, which maintains the invariant.

Now, we must show that the merge follows the specification of the AW-set merge operation. We assume that the methods findNode and insertNode follows the properties of Binary Search Tree, thus ensuring unique nodes for the items added to the tree.

The merge operation is specified as

$$merge_{\texttt{awset}}(\sigma_s \cdot \sigma_c, (\sigma_s, \sigma_l)) = \sigma_s \cdot \sigma_c \cdot \sigma_l'$$

where

$$\begin{aligned}\sigma_l' = \{&m \mid m \in \sigma_l \wedge \\ &(m = add(a) \wedge \nexists m' \in \sigma_l : m' = rem(e) \wedge m' \text{follows } m) \\ &\vee (m = rem(a) \wedge (\nexists m' \in \sigma_c : m' = add(a) \\ &\wedge \exists m'' \in \sigma_l : m'' = add(a) \wedge m'' \text{follows } m))\}\end{aligned}$$

A *rem(e)* is included in σ_l' only if there is no concurrent $add(e)$. The specification also eliminates multiple operations on the same item e and only takes the last operation. If *add(e)* is followed by *rem(e)*, *add(e)* is eliminated and vice-versa.

First, we define two predicates.

$$\begin{aligned}Q_1 \triangleq &\forall e \in \mathit{itemsadded} : \exists node : node.item = e \\ &\wedge \{f, l, r\} \in node.vinfo \wedge l = newvid \wedge r = 0 \\ Q_2 \triangleq &\forall e \in \mathit{itemsremoved} : \exists node : node.item = e \\ &\wedge \{f, l, r\} \in node.vinfo \wedge r \leq newvid \wedge \\ &\nexists \{f', l', r'\} \in node.vinfo : l < f'\end{aligned}$$

We define two abstract variables *itemsadded*, *itemsremoved* to keep track of the operations processed during the merge. The predicate Q_1 says that for each *add(item)* that is merged, there is a *node* for the *item* with vinfo $\{f, newvid, 0\}$ to reflect the fact the last *add(item)* so far is at the version *newvid*.

After processing each *item* in *setl.added*, we add the *item* to *itemsadded*. At the end of the for loop, we can see that *itemsadded* contains all the items in *setl.added* and the predicate Q_1 is true. By $\textsc{Inv}_l(\sigma_l[t], setl[t])$ and the specification of orset, *itemsadded* includes all items added in σ_l'.

The predicate Q_2 says that for each item removed, the item is marked as removed at version *newvid*, by modifying the vinfo. At the end of the for loop, Q_2 is true. By $\textsc{Inv}_l(\sigma_l[t], setl[t]$ and the specification of σ_l', *itemsremoved* consists of all items removed in σ_l'.

We can see that $Q_1 \wedge Q_2 \implies \textsc{Inv}_5(\sigma_g \cdot \sigma_l', setg)$. At line 55, when we update *setg.vid* to *newvid*, we also execute the merge in the abstract execution. Thus $\textsc{Inv}_g$ holds on the updated global states.

Algorithm 3.10 Invariants in merge operation of AW-set.

```
function MERGE(AWSet set)
  setg := set.setg;
  setl := getLocalObject(set.setl);
  lock setg
  newvid := setg.vid+1;
  /* Merge adds */
  $itemsadded = \emptyset \wedge Q_1$
  for ⟨item,parent⟩ in setl.added do
    if setg.root = null then
      n := NEWNODE(item)
      LISTADD(n.vinfo, ⟨ newvid, newvid, 0 ⟩)
      setg.root := n
      $itemsadded := itemsadded \cup \{item\} \wedge Q_1$
    else
      if parent = null then
        parent := setg.root
      end if
      n := FINDNODE(item, parent)
      if n $\neq$ $null$ then
        if LASTNODE(n.vinfo).removed > 0 then
          LISTADD(n.vinfo, ⟨newvid, newvid, 0⟩)
          $itemsadded := itemsadded \cup \{item\} \wedge Q_1$
        else
          LASTNODE(n.vinfo).last := newvid
          $itemsadded := itemsadded \cup \{item\} \wedge Q_1$
        end if
      else
        n := NEWNODE(item)
        LISTADD(n.vinfo, ⟨newvid, newvid, 0⟩)
        INSERTNODE(n, parent)
        $itemsadded := itemsadded \cup \{item\} \wedge Q_1$
      end if
    end if
  end for
  $itemsadded = setl.added \wedge Q_1$
  $itemsadded = \{e | add(e) \in \sigma_l', \sigma_g \cdot \sigma_l' = merge_{\texttt{orset}}(\sigma_g, (\sigma_s[t], \sigma_l[t]))\}$

  /* Continued in Algorithm 3.11 */
```

Algorithm 3.11 Algorithm 3.10 continued.

```
    /* Merge removes */
    $itemsremoved = \emptyset \wedge Q_2$
    for ⟨n,obvid⟩ in setl.removed do
       v := TAIL(n.vinfo)
       if v.removed ≠ 0 then
          $itemsremoved = itemsremoved \cup n.item \wedge Q_2$
       else if v.last > obvid then
          $\exists m \in \sigma_g : m = add(n.item) \wedge v(m) > setl.vid$
       else
          v.removed := newvid
          $itemsremoved = itemsremoved \cup n.item \wedge Q_2$
       end if
    end for
    $itemsremoved = \{e | rem(e) \in \sigma_l', \sigma_g \cdot \sigma_l' = merge_{\texttt{orset}}(\sigma_g, (\sigma_s[t], \sigma_l[t]))\}$
    $Q_2$
    $\text{INV}_5(\sigma_g \cdot \sigma_l', setg)$
    setg.vid := newvid  merge set
    $\text{INV}_g$
    unlock setg
end function
```

Correctness of add and remove Since add and remove operations do not modify the shared tree, they do not affect the invariant on the snapshots or the global view. It is easy to show that $\text{INV}_l(\sigma_l, setl)$ is maintained at the end of add and remove methods.

3.7 Discussion

In concurrent programs with shared-memory synchronization, data structures are in general mutable and updates are executed in-place. The correctness condition that is traditionally applied to these objects is linearizability [55]. Linearizability induces restrictions on possible parallelism and imposes high cost due to coordination and synchronization. In contrast, replicated data types for distributed systems [79, 23, 22] guarantee convergence while avoiding coordination and facilitating asynchronous replication.

Akin to conflict-free replicated data types (CRDTs) in distributed systems, we, therefore, proposed *Multi-view Data Types* which is designed based on the global-local view model. In this model, a thread updates its private view of an object and later merges its changes to the shared global view.

The CRDTs designs used for geo-replicated distributed systems tend to be too inefficient for our purpose because of their relatively expensive merge operation. In geo-replicated systems, the cost of merge operation is negligible compared to the high synchronization cost. Hence CRDTs are not designed to keep the merge efficient. However, in shared memory concurrent programs, since the merge is executed synchronously, it is essential to reduce the cost of it. We identified important properties of multi-view objects that can be employed for efficient implementations in shared memory systems. We discussed the implementation of several MDTs that guarantee these properties thus providing scalable implementations.

Chapter 4

Mergeable Transactional Memory

Destructible updates on shared objects require careful handling of concurrent accesses in multi-threaded programs. Paradigms such as Transactional Memory (TM) [82] support the programmer to correctly synchronize access to mutable shared data by serializing the transactional reads and writes. All operations within a transaction are executed atomically and isolated from concurrent threads, thus providing a consistent view of the state. When transactions concurrently operate on the same memory locations, with at least one thread updating the variable, the operations conflict. Transactions thus fail their serializability certification check and have to re-execute [78].

Semantically, serializability is unnecessarily strict for a multitude of applications. For example, Algorithm 4.1 shows the code snippet for K-means clustering from the STAMP benchmark suite [70]. The K-means algorithm partitions n data points (given as x,y coordinates) into k clusters such that the total distance for the data points to their respective cluster center is minimized. In each iteration, a thread picks up a data point p, finds its nearest cluster center and updates the cluster center. Classical TMs serialize all transactions that access the same cluster center. However, the only requirement for the correctness of the algorithm is that, after all the points have been processed, `cluster.points` and `cluster.count` must contain the sum of all points and the number of points that belong to that cluster, respectively. Even with relaxed transactions [75, 74, 26], conflicts and hence aborts can still arise when updates cannot be serialized.

In this chapter, we discuss Mergeable Transactional Memory (MTM) which implements a consistency semantics that allows for more scalability even under contention. Instead of aborting and re-executing, object versions from conflicting updates are merged using data-type specific semantics.

Algorithm 4.1 Pseudocode for parallel K-means clustering algorithm using STM.

```
function KMEANS(k, points, numIterations)
   chunks = divide points into N chunks
   for i in 0..k do
      clusters[i].center = a random point from points
   end for
   for loop in 1..numIterations do
      In Parallel
         for n in 1..N do
            thread[n].CLUSTER(clusters, chunks[n])
         end for
      for i do in 0..k
         clusters[i].center = cluster.points/cluster.count;
      end for
   end for
   return clusters
end function

function CLUSTER(clusters, points)
   for p in points do
      cluster = findNearestCluster(clusters, p)
      beginTxn
         cluster.count++;
         cluster.points += p; //add x,y coordinates
      endTxn
   end for
end function
```

Overview

- We introduce MTM in Section 4.1, and discuss the semantics of MTM informally.
- We define a call by need core calculus for MTM and discuss the operational semantics of MTM in Section 4.2.
- In Section 4.3, we describe an algorithm that implements MTM semantics.
- In Section 4.4, we discuss how to use multi-view objects in MTM.

4.1 Mergeable transactions

MTM allows to compose operations on shared mergeable objects. Akin to Software Transactional Memory (STM), MTM guarantees atomicity, isolation and (weak) consistency for dynamic transactions operating on mergeable objects. A mergeable object is an instance of a data type that defines a merge operation. Two versions of a mergeable object can be merged to get a new version. In Section 4.4, we discuss the relation between Multi-view Data Types and the mergeable objects used in MTM.

Similar to snapshot isolation [75], MTM transactions read from a consistent snapshot and operate concurrently on shared objects. Instead of aborting and re-executing in case of conflicts, transactions commit their changes by merging states of concurrently updated objects. All updates from a transaction become visible together. An efficient merge operation enables MTM to execute multiple updates in parallel to other threads and execute the merge inside the critical section.

Semantics of Execution The shared objects are available in a global memory. At any (logical) time, the global state is a consistent snapshot that includes all updates from transactions that have committed. A snapshot is created when a transaction commits.

When a transaction starts, it sees a consistent snapshot of the global state which was created before the start of the transaction. The transaction then creates a local copy of this consistent snapshots. A transaction always executes the reads and writes on its local snapshot. Thus, no updates that were committed after the snapshot was created are visible to the transaction. For each update operations, the transaction modifies its local snapshot. At the commit, it merges its local snapshot to the global state to create a new snapshot. The merge incorporates all the local updates into the global state using the type-specific merge operation of the shared objects.

The snapshots evolve linearly, similar to snapshot isolation [75] and serializability [78]. Figure 4.1 gives an example. The states (numbered 0-5) are snapshots. Snapshot s_0 is the initial state. Transaction t_1 reads from s_0. It executes updates on its local copy of the snapshot and buffers the modified state in s_{t_1}. Concurrently, another transaction t_2 reads from s_0 and buffers its updates in s_{t_2}. During the commit, t_1 merges its changes to the latest snapshot s_0 to create a new snapshot s_1. Then, t_2 merges its changes with the latest snapshot, which is s_1, to get s_2. While t_2 is committing, t_3 reads snapshot s_1. Any number of concurrent transactions may execute on a particular snapshot. However, the commits are serialized so that the snapshots are totally ordered.

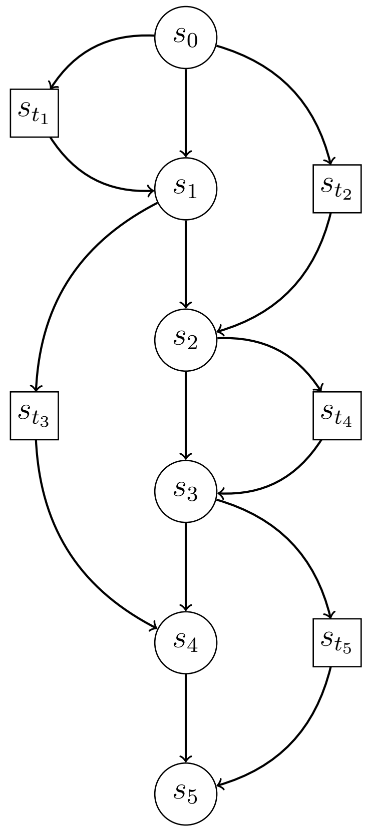


Figure 4.1: Evolution of snapshots in MTM. s_i: snapshot i. s_{t_i}: state of transaction t_i.

4.2 Operational semantics

To specify the consistency semantics of MTM transactions, we introduce a call-by-need core calculus, Λ_{MTM}, with an operational semantics based on transition rules. Figure 4.2 shows the syntax of Λ_{MTM}. It relies on disjoint sets of variables (Var) and references (Ref). A value is either a reference r, a mergeable value m, a function, a monadic return, an integer i, or unit $()$.

Expressions are given as values, variables, function application, monadic bind, thread fork, MTM transactions, operations on objects, and arithmetic expressions. The expressions marked in gray do not appear in source programs but represent dynamically generated locations and intermediate system states arising during commits.

$$\begin{array}{llll} x \in & \text{Var}, & r \in & \text{Ref} \\ v \in & \text{Val} & ::= & \boxed{\texttt{r}} \mid m \mid \lambda x.e \mid \texttt{return}\ e \mid i \mid () \\ e \in & \text{Exp} & ::= & v \mid x \mid e\ e \mid e \gg\!= e \mid \texttt{forkIO}\ e \mid \\ & & & \texttt{eventually}\ e \mid \boxed{\texttt{commit}\ \Theta\ e} \mid \texttt{new}\ e \mid \\ & & & \texttt{read}\ e \mid \texttt{write}\ e \mid e + e \mid e * e \mid ... \end{array}$$

Figure 4.2: Syntax of Λ_{MTM}.

$$\begin{array}{lll} t & \in \text{ThreadId} & \\ \Theta & \in \text{Heap} & = \text{Ref} \rightharpoonup \text{Exp} \\ P & \in \text{ThreadPool} & = \text{ThreadId} \rightharpoonup \text{Exp} \end{array}$$

Figure 4.3: Λ_{MTM}: State-related definitions.

A program state $P;\Theta$ is a pair consisting of a thread pool P (partial mapping of thread identifiers to expressions) and a heap Θ (Figure 4.3). A reference l corresponds to an object allocated on the heap Θ. Dereferencing $\Theta(l)$ yields the associated object, while a heap update $\Theta[l \mapsto e]$ returns a heap that is identical to Θ, but maps l to e. Similarly, we denote updates in the thread pool P by $P\{t \mapsto e\}$.

The evaluation of a program starts in an initial state $\{t_0 \mapsto e\};\emptyset$ with an empty heap and the main thread t_0. The evaluation stops when the program reaches a final state of the form $\{t_0 \mapsto v_0, \ldots, t_n \mapsto v_n\};\Theta$. The reduction rules in Figure 4.4 define the semantics of the language constructs. Each global reduction step $\rightarrowtail$ nondeterministically selects a thread from P, thus modeling an arbitrary thread scheduling.

The IO Monad is the top-level evaluation context. Rule IO-Monad enables the execution of reductions within the current context. Spawning a thread (rule Spawn) adds a new entry with a fresh thread identifier to the thread pool and returns unit to the parent thread. A transactional expression is evaluated against a copy of the current heap (rule Txn), possibly using multiple transactional transitions denoted by $\Rrightarrow$.

Within a transaction, reading an object returns the value referenced in the heap (rule Read). Similarly, after applying the updates, the resulting value is written back to the heap (rule Write), replacing the previous value. When allocating a new object, rule New ensures that the heap is extended using a fresh reference (i.e. one that has not been used in the heap or in concurrently running threads). The initial value of the object is then added to the transaction-local heap instance under the new reference.

Finally, an evaluated transaction is represented as a commit record con-

Evaluation contexts:

$$\mathbb{E} ::= [] \; e \mid [] \gg\!= \; e \mid [] + e \mid v + [] \mid [] * e \mid ...$$

Expression evaluation $\rightarrow$:

$$(\lambda x.e) \; e' \rightarrow e[e'/x] \qquad \frac{e \rightarrow e'}{\mathbb{E}[e] \rightarrow \mathbb{E}[e']} \qquad i + j \rightarrow i \oplus j \qquad i * j \rightarrow i \otimes j$$

$$\texttt{return} \; e' \gg\!= \; e \rightarrow e \; e'$$

Thread evaluation $\rightarrowtail$:

$$\frac{t' \text{ fresh}}{P\{t \mapsto \mathbb{E}[\texttt{forkIO} \; m]\}; \Theta \rightarrowtail P\{t \mapsto \mathbb{E}[\texttt{return} \; ()], t' \mapsto m\}; \Theta} \qquad \textsc{Spawn}$$

$$\frac{e; \Theta \Rightarrow \texttt{return} \; e'; \Theta'}{P\{t \mapsto \mathbb{E}[\texttt{eventually} \; e]\}; \Theta \rightarrowtail P\{t \mapsto \mathbb{E}[\texttt{commit} \; \Theta' \; e']\}; \Theta} \qquad \textsc{Txn}$$

$$P\{t \mapsto \mathbb{E}[\texttt{commit} \; \Theta' \; e]\}; \Theta \rightarrowtail P\{t \mapsto \mathbb{E}[\texttt{return} \; e]\}; \Theta \Cup \Theta' \qquad \textsc{Commit}$$

$$\frac{e \rightarrow e'}{P\{t \mapsto \mathbb{E}[e]\}; \Theta \rightarrowtail P\{t \mapsto \mathbb{E}[e']\}; \Theta} \qquad \text{IO-}\textsc{Monad}$$

Evaluation steps in transaction $\Rightarrow$:

$$\frac{\Theta(r) = m}{\mathbb{E}[\texttt{read} \; r]; \Theta \Rightarrow \mathbb{E}[\texttt{return} \; m]; \Theta} \qquad \textsc{Read}$$

$$\frac{\Theta(r) = m}{\mathbb{E}[\texttt{write} \; r]; \Theta \Rightarrow \mathbb{E}[\texttt{return} \; ()]; \Theta[r \mapsto m]} \qquad \textsc{Write}$$

$$\frac{r \text{ fresh}}{\mathbb{E}[\texttt{new} \; m]; \Theta \Rightarrow \mathbb{E}[\texttt{return} \; r]; \Theta[r \mapsto m]} \qquad \textsc{New}$$

$$\frac{e \rightarrow e'}{\mathbb{E}[e]; \Theta \Rightarrow \mathbb{E}[e']; \Theta} \qquad \text{MTM-}\textsc{Monad}$$

Figure 4.4: Operational Semantics for Λ_{MTM}.

sisting of the local heap copy, containing possible modifications, and the expression to be returned. Rule COMMIT then applies the heap modifications atomically to the globally shared heap and returns. The changes from the local heap copy Θ' are propagated to the current globally shared heap Θ by merging the individual entries with the thread-local ones. The function $\uplus$ :: Heap $\times$ Heap $\rightarrow$ Heap defines the heap merge:

$$(\Theta \uplus \Theta')(r) = \begin{cases} \text{merge } m\ n & \text{if } \Theta(r) = m, \Theta'(r) = n \\ m & \text{if } r \notin \mathit{dom}(\Theta), \Theta'(r) = m \\ n & \text{if } r \notin \mathit{dom}(\Theta'), \Theta(r) = n \end{cases}$$

4.2.1 Properties of MTM

Based on the operational semantics for Λ_{MTM}, we can now further characterize MTM transactions.

MTM allows non-serializable transactions. By rule TXN, the heap-modifying side-effects of a transaction `eventually` e are not immediately applied to the shared global state but deferred to another reduction step under rule COMMIT. Depending on the scheduling, other transactions may also execute without committing their changes yet. If there are read-write dependencies between the transactions, it is not possible to construct a reduction sequence yielding the same final state.

All updates are eventually applied to the shared state. The type-specific merge during the commit ensures that concurrent updates are merged deterministically into a consistent state of the object.

All updates performed by a transaction are made visible atomically. By rule COMMIT, all updates from a transaction are merged to the globally shared heap in one step, which guarantees atomicity.

All reads performed by a transaction appear to be executed at a single point in time. In addition to publishing the updates atomically, transactions are executed on a consistent snapshot; i.e., a snapshot in which either all updates from some transaction that committed before the snapshot time are visible or none. All read operations within a transaction are guaranteed to see the state of objects from a consistent snapshot taken at the time when the transaction started. Rule TXN shows that all operations inside a transaction are executed against the same state Θ. Although there could be concurrently executing transactions, their updates are not globally visible.

4.3 Algorithm

An algorithm for implementing the semantics of MTM transactions is given in algorithm 4.2. To guarantee that a transaction never tries to read an object that has been modified by another transaction while executing (leading to a read-write conflict), we apply a multi-versioning scheme for mergeable objects. As previous studies have shown [75, 74, 26], multi-versioning of objects can be efficiently employed to achieve permissive transactions.

A shared mutable reference to a mergeable object which can be accessed in a MTM transaction is represented by `var`. A `var` references a list of versions. Each version contains a value of the object and its version identifier.

A transaction `txn` maintains a snapshot id `sid` in addition to a read set and a write set which are represented as maps. When the transaction starts, its `sid` is assigned to be the current value of a *globalclock*. The *globalclock* is expected to generate unique monotonic numbers so that versions are totally ordered. The operations of the transaction are executed on the snapshot identified by this `sid`, which includes updates from all transactions committed before this time.

A `var` is accessed using the `READ` and `WRITE` methods. When reading, if the write set or read set contains a local copy of `var`, it is returned. Otherwise, the version corresponding to the transaction's snapshot-id (`sid`) is obtained and inserted in the read set. A new value of the object is written back to `var` using method `WRITE`, which inserts the value in the write set. Reading an object does not necessarily pass over the entire object. Depending on the actual representation of the object, a read might only be reading a reference.

When committing, the transaction acquires a lock on all objects in its write set. This ensures atomicity when two transactions try to commit to the same object. To prevent deadlocks, locks are obtained in a predefined order. Next, a new version id is generated from the current global clock value. The objects updated in the transaction are then merged with the latest version available, using the objects' merge method, thereby creating new versions.

Algorithm 4.3 shows the versioned read and write functions. The function `WRITENEWVERSION` adds the new value with its *vid* to the head of list of versions. Since *globalclock* is incremented during commit, the `sid` of a transaction always denotes the version id of a committed transaction or a concurrently committing transaction. When reading from the list of versions of a `var`, if the required version is not available, a concurrent transaction might be committing that version. Hence, it waits for the lock to be released before retrieving a version with an id equal to or smaller than its `sid`. If

Algorithm 4.2 MTM Algorithm.

```
Transaction : {VersionId sid, Map writeset, Map readset}
Var : {Versions versions, Lock lock}
Versions : [{Value val, VersionId vid}]
VersionId : int
function BEGINTRANSACTION(Transaction txn)
  txn.sid ← globalclock
  txn.writeset ← ∅
  txn.readset ← ∅
end function

function READ(Var var,Transaction txn)
  if txn.writeset.contains(var) then
    val ← txn.writeset.lookup(var)      ▷ read your own writes
  else if txn.readset.contains(var) then
    val ← txn.readset.lookup(var)
  else
    val ← READVERSION(var, txn.sid)
    txn.readset.add(var,val)
  end if
  return val
end function

function WRITE(Var var, Value val, Transaction txn)
  txn.writeset.insert(var,val)
end function

function COMMIT(Transaction txn)
  lockAll(txn.writeset)
  versionid ← ++globalclock
  for all (var,val) ∈ txn.writeset do
    v' ← READLATESTVERSION(var)
    newval ← merge(v',val)
    WRITENEWVERSION(var, newval, versionid)
  end for
  unlockAll(txn.writeset)
end function
```

Algorithm 4.3 Versioned read and write operations in MTM.

```
function READVERSION(Var var, VersionId versionid)
   v ← var.versions
   if v.head.versionid ≥ vid then
      // Required version is available
      vr ← v.head
   else
      //Wait for a concurrent committer to write required version
      waituntil (not locked(var))
      vr ← var.versions.head
   end if
   while vr.versionid > versionid do
      vr ← vr.next
   end while
   return vr.val
end function

function READLATESTVERSION(Var var)
   return var.versions.head.val
end function

function WRITENEWVERSION(Var var, Value val, VersionId versionid)
   v ← newVersion(val, versionid)
   var.versions.addHead(v)
end function
```

the lock is released, it means that there is no other transaction which could potentially commit a version required by this transaction. This guarantees that a transaction always reads from a consistent snapshot identified by its `sid`.

4.3.1 MTM in Haskell

We implemented a prototype of MTM in Haskell. Harris et al. [46] have highlighted the benefits of Haskell's monadic type system for composing STM actions and restricting access to transactional variables to the STM monad. MTM is implemented analogously to the STM monad, though with different semantics.

For the MTM programming model, we provide an `MTM` monad (Fig-

```haskell
data MTM a  = ...
data CVar a = ...

-- MTM Functions
eventually :: MTM a -> IO a
newCVar    :: Mergeable a => a -> MTM (CVar a)
readCVar   :: Mergeable a => CVar a -> MTM a
modifyCVar :: Mergeable a => CVar a -> (a -> a) -> MTM a

-- Mergeable Objects
class Mergeable a where
   merge :: a -> a -> a
```

Figure 4.5: Interface for MTM in Haskell.

ure 4.5). The shared mergeable objects used in MTM transactions are of type `CVar`; `CVar`[1] stands for convergent variables indicating that concurrent versions converge into a consistent state. Every operation executed on a `CVar` must be an MTM action. These actions can be sequentially combined using monadic bind. The function

```haskell
eventually :: MTM a -> IO a
```

takes an MTM action, executes it, and returns the result. Using function `modifyCVar` to update a `CVar` guarantees that the mergeable values does not escape a transaction's scope.

The type specification ensures that mergeable objects are accessed only inside a MTM transaction. These objects must be of `class Mergeable` and define a `merge` function. Figure 4.6 shows the implementation of two mergeable objects. The `Counter` contains two integers, one representing the global value and the other the thread-local increments. The `merge` adds the local increments to the global value `g` and resets the local increments to 0. The `LWWRegister` implements a last-writer-wins register, where the last merge overwrites the previous value.

Example The following example shows how to program with `CVars` and the MTM monad in Haskell.

```haskell
addToBag :: Int -> CVar (Bag Int) -> CVar (Counter) -> MTM [
   Int]
addToBag e bag size = do {
```

[1]The name `MVar` for mergeable variables is already used in Haskell.

```haskell
-- Mergeable Counter
data Counter = Counter Int Int
instance Mergeable Counter where
 merge (Counter g _) (Counter _ i) = Counter (g+i) 0

newCounter::Counter
newCounter = Counter 0 0
value :: Counter -> Int
value Counter g l = g+l
incrBy :: Int -> Counter -> Counter
incrBy i (Counter g l) =
 Counter g (l+i)

-- LWWRegister
type LWWReg = Int
instance Mergeable LWWReg where
    merge g l = l

-- Mergeable Bag
data Bag a = Bag [[a]] [a]
instance CRDT (Bag a) where
 merge (Bag g _) (Bag _ i) =
  Bag (i:g) []
newIntBag :: Bag Int
newIntBag = Bag [[]] []
add :: a -> Bag a -> Bag a
add e (Bag g l) = Bag g (e:l)
```

Figure 4.6: Mergeable objects in Haskell.

```
  ; b <- modifyCVar bag  (add e)
  ; s <- modifyCVar size (incrBy 1)
  ; return (toList b)
}
```

The function `addToBag` inserts an element to some bag and increments a counter representing the size of the bag. It then returns the elements from the bag in a list, including the added element `e`, but excluding elements that have been concurrently added. When calling the function using

```
eventually addToBag x b s
```

with a bag `b` and size counter `s`, the library guarantees that both shared objects are atomically updated and have consistent values.

4.4 Relation to MDTs

Multi-view Data Types define a merge operation and provide an efficient implementation that can handle multiple versions (views) and the merge operation. A question that naturally follows is whether we can use multi-view objects (pure mergeable) in MTM.

To support multi-versioned snapshots, multi-view objects must provide multiple versions of the global view. However, the global-local view model defines only one global view. There are two ways to use multi-view objects in MTM.

1. MTM transaction handle the version management.
   MTM algorithm described in Section 4.3 does version management within the transaction. The object versions are treated as immutable values that are cloned to create new versions. We can use multi-view objects similarly in MTM. However, treating them as black boxes would not let us exploit their efficient designs to handle multiple versions.

2. MDT does versioning of its global view.
   Although it was not a property of the global-local view model, most of the MDT implementations (Section 4.4.2) are suitable for supporting a multi-versioned global view. Hence, we argue that extending the MDT model to support multiple versions of the global view and make the multi-view objects do their version management would enable us to exploit the performance benefit of MTM fully.

Figure 4.7 shows a multi-view object extended to support MTM. Compared to Figure 3.1, the global view in Figure 4.7 is multi-versioned. Each

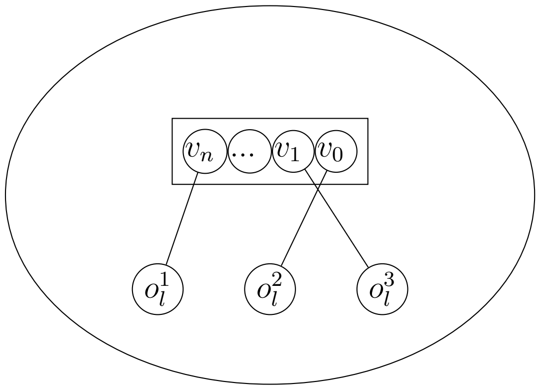


Figure 4.7: A multi-view object with multi-versioned global view. v_i: version i of the global view. o_l^i: thread-local view of thread i.

merge creates a new version of the global view. An MTM transaction reads from a specific version from the list of global versions. Thus the local view of each transaction(thread) is derived from a specific version of the global view.

4.4.1 Extensions to Global-Local View Model

We extend the global-local view model to support the use of MDTs in MTM. The modifications of the abstract model from Figure 3.13 is shown in Figure 4.8. The global view is now a map of version id to a version ($[vid \mapsto \sigma]$). The version ids are totally ordered. The initial version of the object consists of version 0 mapping to an empty sequence of operations (Rule ALLOC). The pull operation provides a version id to get a specific version (Rule PULL). The merge operation executes the type-specific merge on the latest version of the global view (Rule MERGE). The weak read and weak update operations execute on the local view and hence not affected by these extensions.

MTM assumes that updates to all shared mutable objects are *weak* updates that can be merged at the time of commit. *Strong* operations can result in conflicts which cannot be merged or serialized at the time of commit. Hence we consider only purely mergeable MDTs and do not discuss MDTs with *strong* operations.

4.4.2 Impact on the implementation

Although not intended for the design of multi-view objects, most of the implementations discussed in Section 3.5.2 can support a multi-versioned global

$$\frac{\texttt{newaddr}(\Gamma) = r}{\langle \texttt{new}\ \tau, (\Lambda, \Gamma)\rangle \hookrightarrow \langle r, (\Lambda, \Gamma[r \mapsto [0 \mapsto \emptyset]])\rangle}\ \textsc{Alloc}$$

$$\frac{\mathit{vid} \mapsto \sigma_g \in \Gamma(r)}{\langle \mathbb{E}[\texttt{pull}\ \mathit{vid}\ r], (\Theta, \Lambda, \Gamma)\rangle \hookrightarrow \langle \mathbb{E}[\texttt{skip}], (\Lambda[r \mapsto \langle \sigma_g, \emptyset\rangle], \Gamma)\rangle}\ \textsc{Pull}$$

$$\frac{\begin{array}{c}\mathit{lastvid} \mapsto \sigma_s \cdot \sigma_c \in \Gamma(r)\\ \Lambda(r) = \langle \sigma_s, \sigma_l\rangle \qquad \mathit{type}(r) = \tau \qquad \mathit{merge}_\tau(\sigma_s \cdot \sigma_c, \langle \sigma_s, \sigma_l\rangle) = \sigma_g\end{array}}{\langle \texttt{merge}\ r, (\Lambda, \Gamma)\rangle \hookrightarrow \langle (), (\Lambda[r \mapsto \langle \sigma_g, \emptyset\rangle], \Gamma[r[\mathit{lastvid} + 1 \mapsto \sigma_g]])\rangle}\ \textsc{Merge}$$

Figure 4.8: Operations in GL model extended to support multi-versioned global view.

view. We discuss how the implementations of the counter, the bag and the add-wins set can be used in MTM.

Counter The global view of the counter is represented by an integer. To support multiple versions of the global view, the counter replaces the integer by a map of version ids to integers. Thus the state of the new MTM-enabled counter is defined as follows:

```
type Counter = {
    Map<VersionId, int> gcntrmap,
    ThreadLocal CounterLocal lcntr
  }
```

The operations that access the global view also must be adapted to operate on the required version.

Bag As we discussed in Section 3.5, copying the entire object for each version is only suitable for objects that are small in size, such as the counter. Fortunately, the add only bag implementation (Section 3.5.2) already provides a multi-versioning scheme in the form of a multi-headed list. To keep multiple versions of the global view, the bag can store a map of version ids to the heads of the list resulting from each merge. Figure 4.9 shows a map of version ids to the lists resulting from the merge operation of two threads as shown in Figure 3.14.

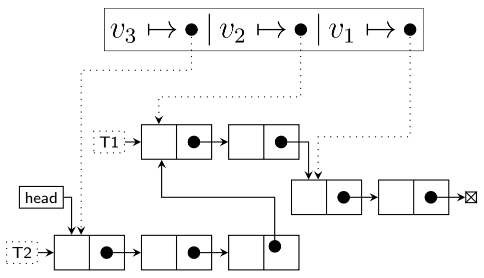

$v_3 \mapsto \bullet$ | $v_2 \mapsto \bullet$ | $v_1 \mapsto \bullet$
T1
head
T2

Figure 4.9: Multiple global versions in Add only bag. Each version v_i points to the head of the list resulting from a merge operation. The dotted boxes represents the the thread-local heads before the merge.

AW-set The implementation of AW-set has a version management system internally. Each node stores the metadata regarding the versions in which the item was added or removed. It keeps the entire history of the versions; thus all the global versions are immediately available without any modification to the design. This information is already used in the lookup operation that looks up an item on a particular version.

4.5 Discussion

MTM provides a novel semantics as an alternative to the often too strict semantics of serializable transactions. The relaxed semantics together with mergeable semantics embedded in the data types prevents unnecessary aborts and thus improves the overall performance.

Despite the scalable semantics, MTM comes with its overhead. MTM needs additional memory for storing multiple versions of each object as well as the metadata related to each version. A long-running transaction must traverse the entire list of versions to find an old version that it requires.

As in any other multi-versioning algorithm, MTM implementations must also remove old versions to limit the memory consumption. The garbage collection mechanism must ensure when it is safe to remove a version so that versions required by long running transactions are still reachable. The processing needed for garbage collection would thus be an additional overhead compared to traditional STMs. Nevertheless, it will be an exciting future work to find an efficient garbage collection algorithm tailored for MTM.

Current MTM semantics allows only *weak* operations within the transaction. To allow other multi-view objects with strong operations or other non-mergeable traditional data types, we must extend the semantics of MTM.

This extension will broaden the applicability of MTM to more programs that may exploit the mergeability of some data types while keeping the traditional serializable semantics for others.

Chapter 5
Evaluation

To evaluate the performance and scalability of Multi-view Data Types (MDT) and MTM, we provide several microbenchmarks and application benchmarks.

In Section 5.1.1, we describe the implementation of the MDTs and the reference implementation of the corresponding linearizable data type. We implemented and evaluated counter, queue, and AW-set. The microbenchmarks in Section 5.1.2 shows that MDT implementations perform and scale better compared to the linearizable implementations. We also discuss some of the application benchmarks that use MDTs. We evaluated a breadth-first traversal on a graph using the MDT queue, and work stealing queue implemented using the global-local view model. These application benchmarks show the usability of MDTs in common algorithms and evaluate the scalability compared to using linearizable data types.

In Section 5.2.1, we discuss the implementation of MTM in Haskell. We evaluated MTM using several microbenchmarks and an application, as presented in Section 5.2.2.

5.1 Multi-view Data Types

5.1.1 Implementation

We implemented the MDTs in Java. The implementation followed the generic pattern described in Section 3.5.2. Each MDT is represented by a class. The local view is defined as a nested class and instantiated in the MDT as a `ThreadLocal` object. An object with `ThreadLocal` descriptor has a thread-local copy of the object independently initialized.[1] Thus, it allows each thread to maintain its own independent thread-local view.

[1]Many programming languages support some form of thread-local storage (TLS). Java has a `ThreadLocal` class that provides thread-local variables. In C++, one can have thread local variables with the help of `thread_specific_ptr` from the boost library. A mergeable data type can also implement its own thread local storage by mapping thread ids to different instances of the object.

Counter The global view of the counter is an `AtomicLong` that provides atomic update methods. The local view is a class that consists of two integers: one for the snapshot and one for the local increments. The merge uses the atomic `addAndGet` method of the `AtomicLong`. The access to the `AtomicLong` object is the only point where any synchronization is needed. The strong operations such as *strongInc* and *strongDec* also use the atomic methods of the `AtomicLong` global view.

To compare the performance of the Mergeable Counter, we also implemented a linearizable counter. The linearizable counter is an `AtomicLong` object and uses its atomic methods to increment and decrement.

We also implemented the counter in C++ using the boost library. The global view is an `atomic` integer and the snapshot and the local increments in the local view, similar to java implementation are thread-local variables defined using `thread_specific_ptr` of the boost library.

Queue To evaluate the scalability of the hybrid mergeable queue (referred to as mergeable queue), we implemented four different queues in Java. All four designs implement the queue as a single-linked list with a *head* pointing to the first node and a *tail* pointer pointing to the last node. The enqueues add items to the end of the list and dequeues removes items from the head. The four implementations differ in their semantics (linearizable vs. mergeable) and the concurrency control mechanism (lock-based vs. lock-free).

1. Lock-based linearizable (LL) queue based on Michael and Scott's 2-lock queue [69]: This implementation relies on two locks: a *head lock* and a *tail lock*. A thread acquires the *tail lock* when it enqueues an item. The *head lock* is acquired during a dequeue. The *head* always points to a dummy node, so that an enqueue and a dequeue can be executed concurrently without taking each other's locks.

2. Lock-based mergeable (ML) queue: Similar to the LL-queue, it has a *head lock* and *tail lock*, and the global queue's *head* points to a dummy node. The local view consists of a linked list that accumulates all locally enqueued items. The merge acquires the *tail lock* and appends the local linked list to the tail of the global list. The process is similar to the enqueue in LL-queue except that the merge adds more than one item at once instead of one item during the enqueue. The dequeue is similar to the LL-queue where it acquires the *tail lock*.

3. Lock-free linearizable (LLF) queue based on the Michael and Scott's lock-free queue algorithm [69]: This implementation, similar to the LL-

```
private transient volatile Node<T> head =
                              new Node<T>(null, null);
private transient volatile Node<T> tail = head;
ThreadLocal<LocalQ> localView;

public void merge(){
  Node<T> localhead = localView.get().head;
  if(localhead == null){
    //nothing to merge;
    return;
  }
  Node<T> localtail = localView.get().tail;
  Node<T> curtail;
  while(true){
    curtail = tail;
    Node<T> next = curtail.getNext();
    if(curtail == tail){
      if(next == null){
        if(curtail.casNext(next, localhead)){
          casTail(curtail, localtail);
          break;
        }
      }
      else {
        casTail(curtail,next);
      }
    }
  }
  //Remove all merged items from local view.
  localView.get().reset(null);
}
```

Figure 5.1: Code for lock-free merge in MLF-queue.

queue, has a *head* that always points to a dummy node. It uses *compare and swap* to handle concurrent accesses to head and tail pointers. This is a widely used algorithm to implement a concurrent linearizable queue and is used to implement the `ConcurrentLinkedQueue` of the Java Concurrent Collections in OpenJDK (jdk7)[2].

4. Lock-free mergeable queue (MLF): This variant uses a similar algorithm as the LLF-queue. The merge adapts the algorithm of enqueue in the LLF-queue to incorporate insertion of multiple nodes at once. Figure 5.1 shows the lock-free merge using compare and set adapted from the linearizable queue.

Add-Wins Set We compared the performance of Add-Wins Set (AW-set) with two versions of linearizable sets. All three implementations use a binary tree as the underlying data structure. The first linearizable implementation acquires exclusive access to the tree for each operation using the `synchronized` primitive provided in Java. The second linearizable implementation is a non-blocking concurrent binary search tree presented in [32]. In this implementation, modifications to different parts of the tree do not interfere with one another, resulting in less contention. The implementation of AW-set follows the algorithm presented in Section 3.5.2.

Applications

We also implemented and evaluated two applications that can leverage the global-local view model.

Breadth First Search on a graph Breadth First Search (BFS) is an algorithm for traversing a graph data structure. It is used to solve many problems in graph theory such as finding the shortest path or connected components. The sequential BFS algorithm starts from a vertex and adds it to a traversal queue which is initially empty. At each step, the algorithm dequeues a vertex from the queue, processes it, marks it as processed and adds it neighbors to the queue. The traversal finishes when the queue is empty.

The parallel BFS algorithm uses two queues: one for the current level and one for the next level. At each step, the threads dequeue a vertex from the current level to process and add its neighbors to the next-level queue to be processed later. When the current-level queue is empty, the threads execute a

[2]http://openjdk.java.net/

barrier and wait until all threads have finished on the current level. The next iteration uses the next-level queue as the current-level queue.

Since in each iteration parallel threads are accessing both queues, we need to use thread-safe concurrent queues. Typically, a linearizable queue will be used. However, the algorithm does not require linearizable enqueues. At the same time, if an item is dequeued twice, it will lead to redundant work in the best case (if the processing computation is idempotent), or it will lead to incorrect results in the worst case. Hence, we can use our hybrid mergeable queue to speed up the parallel BFS algorithm. In this version, the threads merge the next-level queue at the end of each level so that the local enqueues are visible to the other threads.

We implemented a sequential BFS algorithm to calculate the sum of the numerical label of the vertices in the graph. We implemented the parallel versions using linearizable queues and hybrid mergeable queues. We used JGraphT[3], a free Java graph library, to implement graph objects. The sequential BFS algorithm uses the `BreadthFirstIterator` provided by the library. The library does not provide parallel algorithms. Hence, we implemented a `ParallelBFSIterator` that uses two-level queues. Figure 5.2 shows the code of ParallelBFSIterator using mergeable lock-free queues. The threads use the iterator's `next` function to traverse the graph. The implementation using linearizable queue is similar except that the `levelDone` method do not execute a merge. In the real implementation, ParallelBFSIterator is an abstract class that defines the common methods, and we have subclasses for the iterator with linearizable and mergeable queues.

Work-stealing Queue We further implemented the work-stealing queue designed using the global-local view model. The implementation uses a double-linked list for the global queue and the local queue. Similar to the hybrid mergeable queue, it uses two locks, a *head lock* and a *tail local* to protect the global queue during concurrent dequeues and merges. The access to the local queue does not require any concurrency control primitive as it is only accessed by the owner thread.

To compare its scalability, we also implemented the work-stealing queue of the Cilk runtime, that has a queue per thread. This version also uses a linked list for its queues. Accesses to the local queues are protected by locks to handle concurrent access by the owner thread and the thieves.

[3]http://jgrapht.org/

```java
public class ParallelBFSIterator<V, E> {
  protected Graph<V, E> graph;
  protected V startVertex;
  private ConcurrentHashMap<V,V> seen;
  private MQueue<V> currentLevel;
  private MQueue<V> nextLevel;

  public V next(){
    V nextVertex = currentLevel.dequeue();;
    if(nextVertex == null){
      levelDone();
    }
    else {
      addNeighboursOf(nextVertex);
    }
    return nextVertex;
  }
  private void addNeighboursOf(V vertex){
    for(V n: Graphs.neighborListOf(graph, vertex)){
      V vseen = seen.putIfAbsent(n, vertex);
      if( vseen == null){
        nextLevel.enqueue(n);
      }
    }
  }
  protected void levelDone(){
    nextLevel.merge();
  }
  public boolean hasNext(){
    return !(currentLevel.isEmpty() && nextLevel.isEmpty());
  }
  public void nextLevel(){
    MQueue<V> q = nextLevel;
    nextLevel = currentLevel;
    currentLevel = q;
  }
}
```

Figure 5.2: Code for ParallelBFSIterator using mergeable queue. Constructor and other less significant private methods are not shown.

5.1.2 Evaluation

The evaluations have been performed on a 12 core 2.40GHz Intel(R) Xeon(R) CPU E5-2620 processor (2 NUMA nodes) with 2-way hyper-threading, under linux 4.4.0-62 Ubuntu x86_64 and openjdk version 1.8.0_121, clang version 4.0.0-svn297204-1, boost 1.58.0.1ubuntu1.

Counter We provide and compare two variants of a mergeable counter with a linearizable counter. 1) a mergeable counter with only weak operations and merge, 2) a hybrid mergeable counter where application switches to strong operations when needed. This experiment shows the result of the counter implemented in C++.

In the first version, threads increment a shared mergeable counter and periodically merge with the global view. In the experiment, we allow threads to increment the shared mergeable counter until a *target* value is reached. Since the threads might not know about non-merged increments from other threads, they typically end up overshooting the target. For this experiment, the *target* is set to 5×10^6 increments.

We evaluated several merge intervals and measured their throughput and the overshoot from the target. A merge interval of m means that a thread executes m weak increments between two merges. Figure 5.3 shows that the throughput scales with the number of threads and with the merge interval. At the same time, the overshoot increases. However, the percentage of the overshoot is small. (Notice that the overshoot is upper bound by the number of threads multiplied by the merge interval, as this reflects at any given time the number of increments not yet accounted for.) Points in the lines are labeled with the number of threads employed. As expected, the system does not scale beyond the point where the number of threads exceeds the number of cores (i.e., at 24 threads). Also, note that for a single thread, overshoot is zero and thus the value is outside the logarithmic scale.

Figure 5.4 shows the throughput of the mergeable counter compared to a linearizable counter. The linearizable counter never overshoots the *target*, but since the threads are always competing on the increment, performance is inadequate, and no speedup is obtained from multi-threading. In contrast, the mergeable counter can scale linearly up to a good fraction of the available concurrency, in particular with merge interval from 4096.

While some applications could tolerate an overshoot, in general, applications will require strong target enforcement. To address this, we provide a variant of the mergeable counter that makes a hybrid use of initial weak local

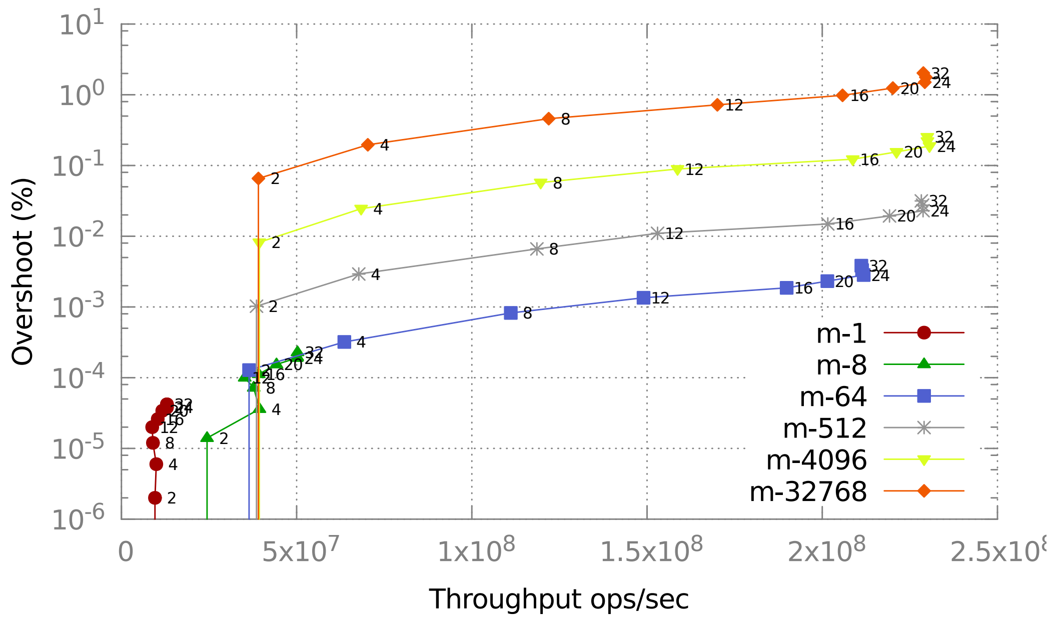


Figure 5.3: Throughput vs Overshoot of mergeable counter with different merge interval.

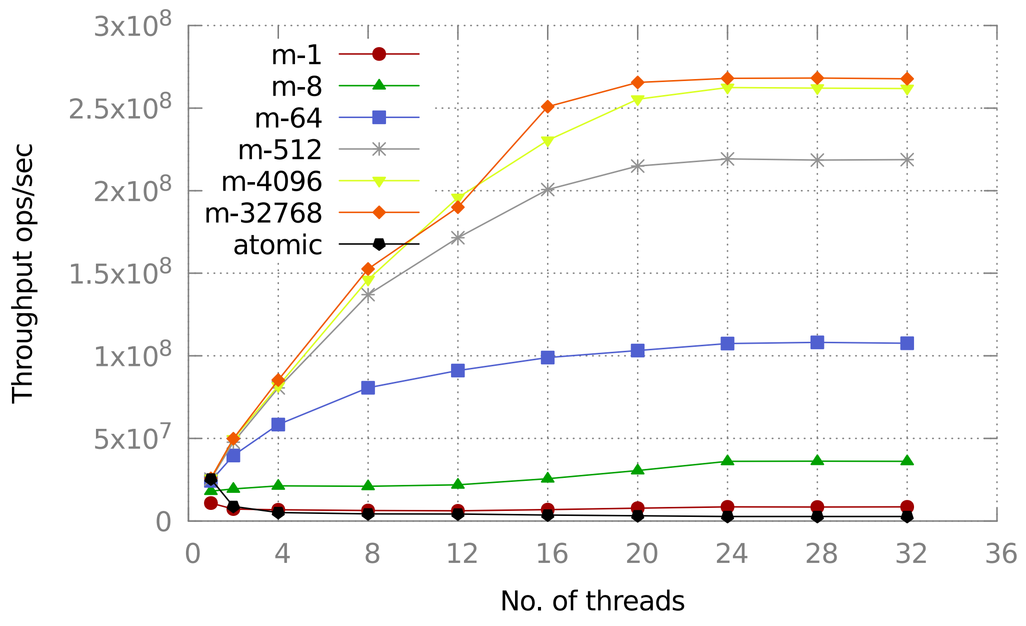


Figure 5.4: Throughput of mergeable counter vs linearizable counter. atomic: linearizable counter. m-i: mergeable counter with merge interval i.

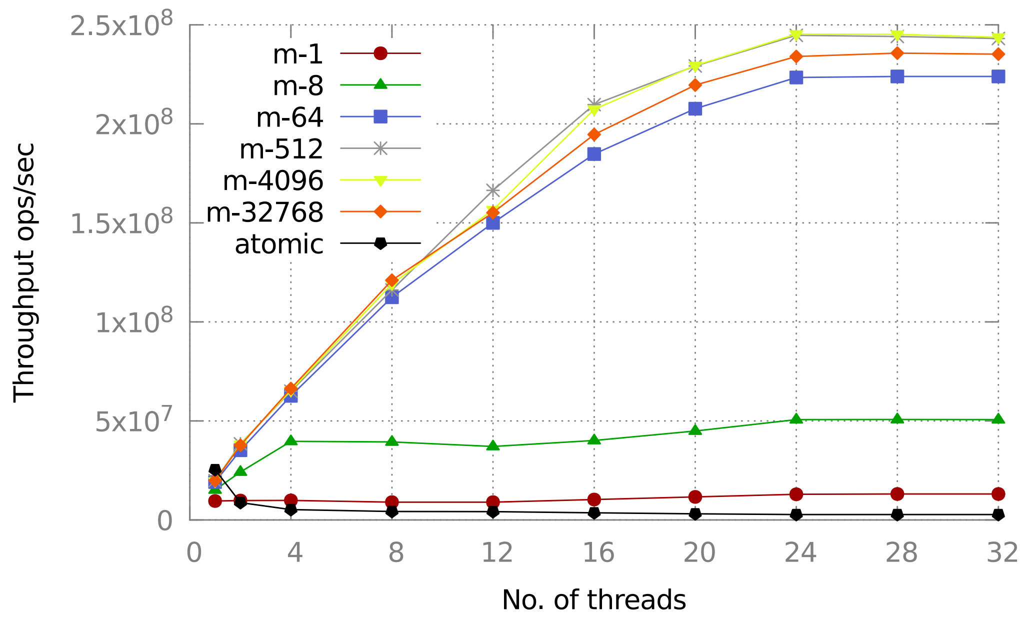


Figure 5.5: Throughput of hybrid mergeable counter (overshoot free) vs linearizable counter. atomic: linearizable counter. m-i: hybrid counter with merge interval i.

increments and later switches to strong increments when approaching the target. The first thread that, upon the periodic merges, detects that it is close to the target, initiates a barrier synchronization to ensure that all threads have switched to strong operations. (We found that all threads must switch in tandem upon this phase transition.)

Figure 5.5 shows that under this approach, overshoot is eliminated while the performance is almost identical to the mergeable counter. In general, the hybrid approach is efficient as long as the target is much larger than the merge interval since this limits the proportion of the execution done under synchronized execution. Note that the strategies to decide when to use weak or strong operations is application dependent and not part of the global-local view model.

Comparison to CRDT Counter. In this experiment, we demonstrate that CRDT designs have significant overhead in performance when used in a shared memory program. This experiment uses the counters implemented in Java. We implemented a CRDT counter on the global-local view model, where each local view and global view are a CRDT replica. We implemented the G-counter [79] using 1) a HashMap that maps thread-id to an integer, 2) an array where the array index corresponds to a thread id. Figure 5.6 shows

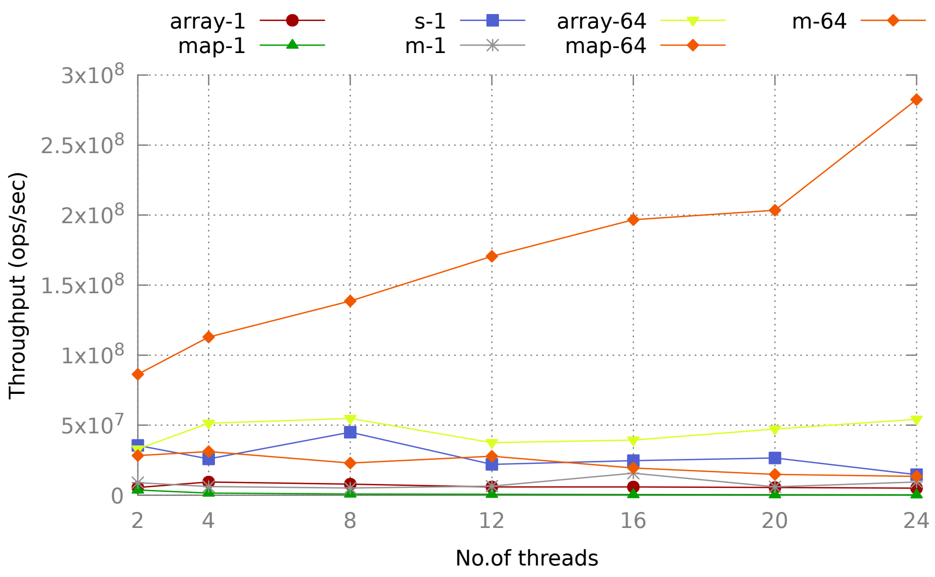


Figure 5.6: Comparison to CRDT Counter using array and map; m - mergeable counter with merge-interval 1,16. sync - atomic counter.

the throughput for the two versions, compared to an atomic counter and a mergeable counter. It shows that the array scales better when the merge-interval is large. However, the size of the array must be fixed to the number of threads. The map implementation does not scale well because 1) there is an overhead in accessing the map entries, 2) merge requires an iteration over the entire map resulting in longer critical section.Thus, the cost of merge operation is negating the benefit achieved by the asynchronous local increment.

Queue We evaluated the four implementations of the queue: lock-based linearizable, lock-free linearizable, lock-based mergeable and lock-free mergeable. We measured the time to perform a total of 5×10^6 enqueues and dequeues by a group of concurrent threads. In the experiment, we divided the group of threads into two halves: one group of producers which only enqueue items to the queues and the second group of consumers that consume items by dequeuing from the queue. We forced half of the producers (and consumers) to run on one NUMA node and the other half on the second NUMA node.

Figure 5.7 shows the total time needed to perform 5×10^6 enqueues and dequeues by the linearizable queues and the mergeable queues with different merge interval m (a thread performs a merge after m enqueues). For both lock-based and lock-free versions, the mergeable queue is faster than its linearizable counterpart. Since this is a high-contention workload, the lock-based version performs better than the lock-free version. Unlike the mergeable counter,

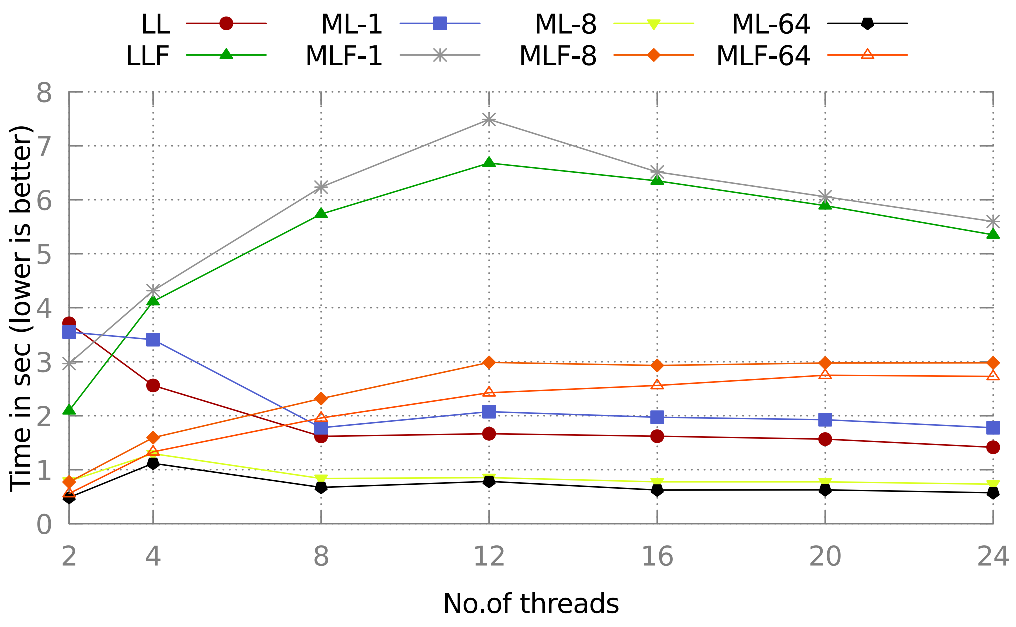


Figure 5.7: Evaluation of Queue. LL: linearizable lock-based, LLF: linearizable lock-free, ML: mergeable lock-based, MLF: mergeable lock-free. 1,8,64 - merge interval for mergeable queues.

increasing merge interval from 8 to 64 does not improve the performance significantly. This is because *dequeue* is always executed in a serial way, which shadows the performance gain from asynchronous *enqueues*

In the previous results, half of the threads were running on one core, and the other half were running on the second core. We ran the same experiment with all of the threads running on the same core and compared them with the previous result. Figure 5.8 shows the time taken for the linearizable queues when threads are running on the same core and different cores. It is evident that there is a significant performance overhead when the threads have to synchronize across different cores.

Figure 5.9 shows the speed up of the mergeable queue compared to the linearizable queues when using same core and different cores. In this experiment, we used the mergeable queue with merge interval 64. The speed-up when all the threads running on the same core is maximum 40% for lock-based implementations and 50% for lock-free implementations. On the other hand, the mergeable queues yield around 57% and 67% speedup, respectively, when the threads are running on different cores. With different access latencies in NUMA architectures, the cost of synchronization with distant cores is higher. Thus the global-local view model becomes more relevant in such many-core architectures where reducing the number of synchronization operations im-

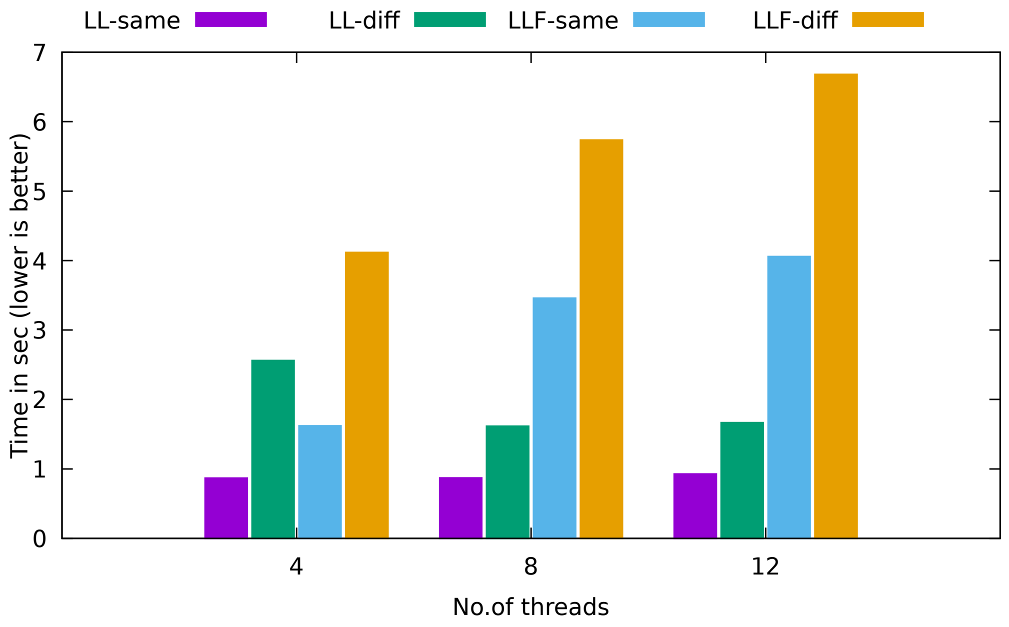


Figure 5.8: Performance of linearizable queue when running on same core vs different core.

proves the performance significantly.

Add wins set We evaluated the mergeable AW-set and two versions of linearizable set implemented over binary trees. The first linearizable version (sync) acquires exclusive access to the tree using locks for each operation. The second linearizable version (lf) is a lock-free implementation. In the experiment, each thread adds and remove items from a set of 10^7 item chosen using a uniform distribution.

Figure 5.10 shows the throughput of the two linearizable implementations and the AW-set at different merge intervals. The throughput of the synchronized tree (sync) is very low and does not scale. The mutual exclusion for each operation limits their concurrency, resulting in such low performance. On the other hand, the performance of AW-set is higher compared to the synchronized version. AW-set with merge interval 8 performs better than the synchronized version, but does not scale well with the number of threads. AW-set with merge interval 64 scales better than the lower merge interval. However, the lock-free concurrent binary search tree (lf) scales better compared to the other implementations. The lock-free implementation does not block concurrent operations that modify different parts of the tree. This results in less contention and better scalability. The AW-set requires exclusive access to the tree during the merge, which prevents concurrent merges to

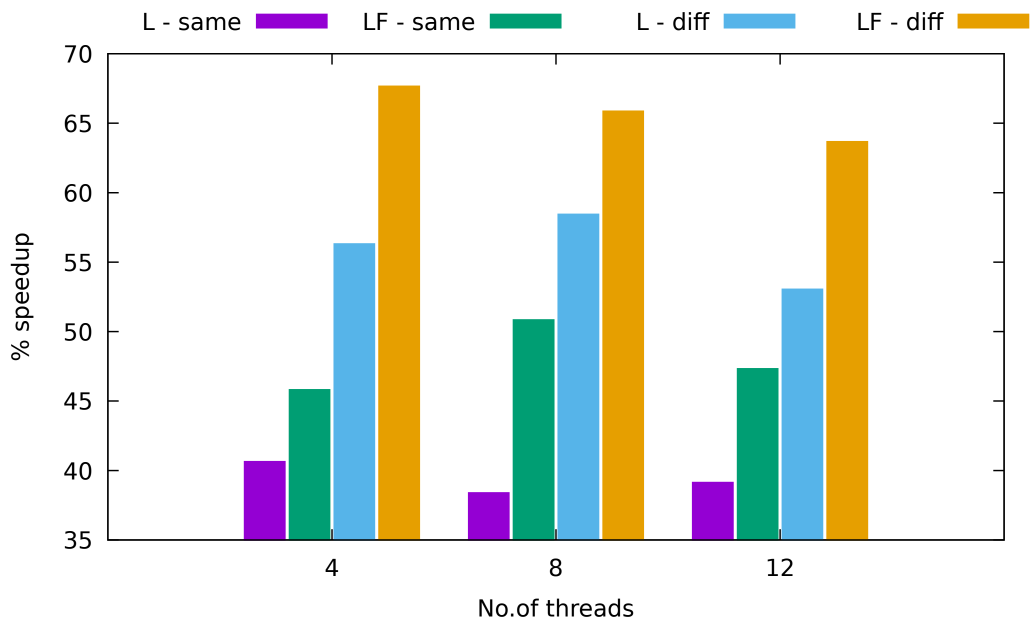


Figure 5.9: Percentage speedup of mergeable queue with merge interval 64 compared to linearizable queue.

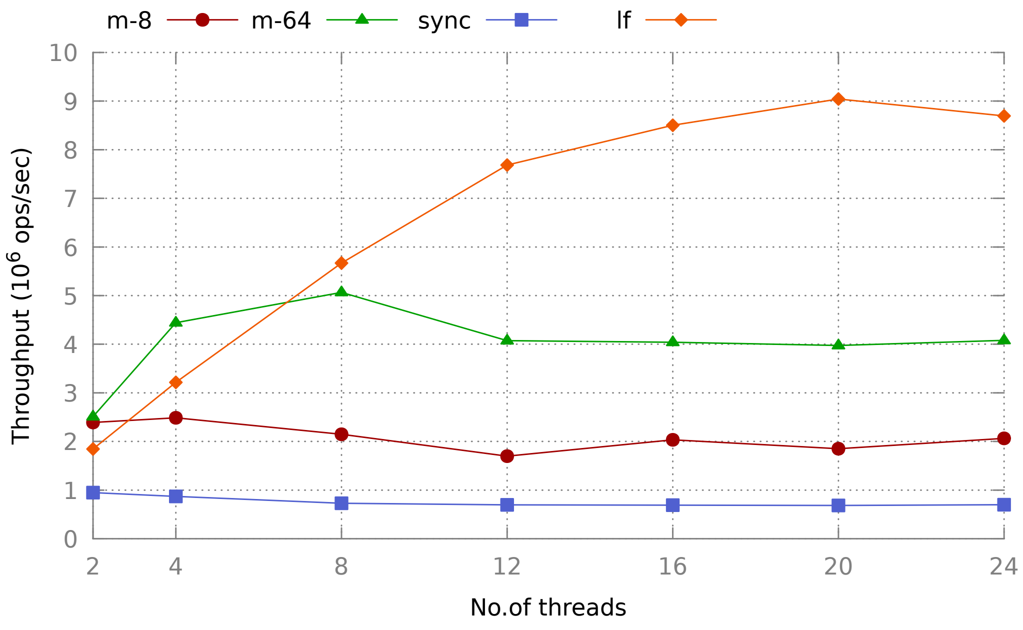


Figure 5.10: Throughput of AW-set compared to linearizable sets. lf - lock-free concurrent BST. sync - synchronized tree. m-i - AW-set with merge interval i.

progress in parallel. On the other hand, the atomic merge operation and access to atomic snapshots in AW-set makes it a good candidate to use in software transactions. A transaction cannot exploit the scalability of lock-free BST.

Breadth-First Traversal We evaluated four versions of the parallel BFS implementation using the four different queue implementations. In this evaluation, each implementation traversed a randomly generated graph having 2 $\times 10^6$ vertices and 2×10^7 edges. Unlike the micro-benchmark for the queue, there is no fixed merge frequency. Instead, the threads merge their local queue at the end of processing each level.

Figure 5.11 shows the speedup of each version compared to a single-threaded implementation. Mergeable queues scale better than their linearizable counterparts. The speedup of the lock-free mergeable queue is significantly higher than that of the others and scales almost linearly until 16 threads. Beyond 16 threads, the number of vertices processed by each thread at each level is reduced, as they are divided among the threads, leading to smaller merge interval. We believe the abrupt drop in the speedup of lock-based queues after 12 threads is due to the additional cost in synchronization to the second NUMA core. Compared to the high-contention micro-benchmark from Figure 5.7, this is a low-contention workload because a significant amount of time is spent in processing the nodes rather than updating the queue.

Work-stealing Queue We evaluated two different versions of work-stealing queues: one implemented on the global-local view model (mv-queue), and another one implemented based on the Cilk runtime's work stealing queue (ll-queue) (see Section 3.2). In this evaluation, each thread executed 10^7 operations. There is no fixed merge interval; instead, the mv-queue executes a merge operation when it observes that the number of items in the global view is below a threshold (In our experiments, we kept the threshold to 2).

We ran the benchmarks with different workloads and a varying number of threads. In the first workload, each thread generates enough tasks for itself such that no thread has to steal tasks generated by others. This is the ideal scenario for a work-stealing queue, in which the task-generation and consumption are uniformly distributed among the threads. However, in real programs, this is hardly the case. In the second workload, half of the threads are generating the tasks and the other half is consuming them. This is the worst-case scenario, where half of the threads are always pushing tasks to

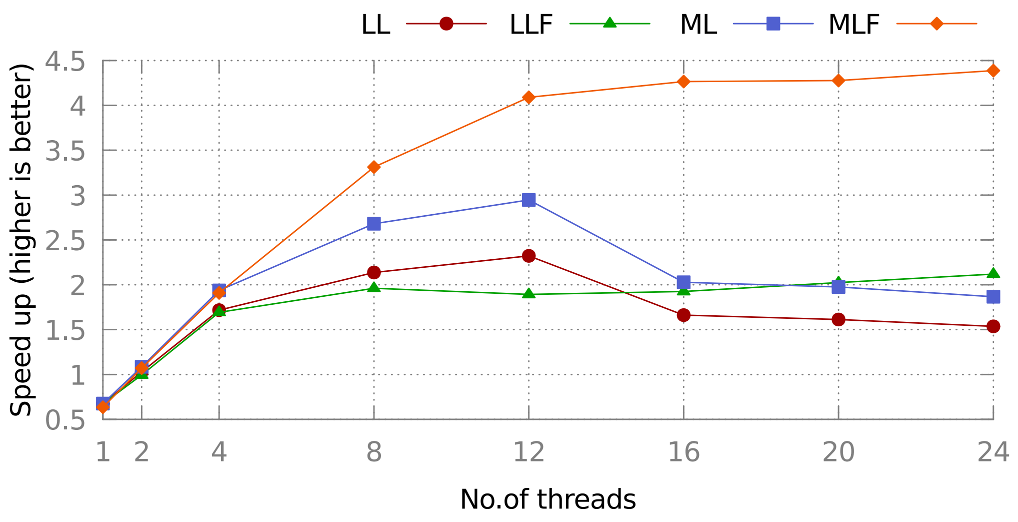


Figure 5.11: Speed up of parallel breadth-first traversal on a graph using different queue implementations compared to a sequential implementation. LL: linearizable lock-based, LLF: linearizable lock-free, ML: mergeable lock-based, MLF: mergeable lock-free.

the queue while the other half is always stealing. It is an extreme case and rarely happens in real applications. The third workload simulates real-world scenarios, where all threads push and get tasks and sometimes steal from others.

Figure 5.12 shows the throughput for the queues for different workloads. Workload 1 has better throughput for both queues because there is no contention in accessing the queue. The mv-queue performs better than ll-queue and scales better because it never has to execute any synchronization operation. On the other hand, the ll-queue has to acquire the locks when updating its local queue for protecting against concurrent access from the thieves even though no threads have to steal in this workload.

Workload 2 has the worst throughput for both queues. Half of the threads are continually trying to steal from other threads resulting in high contention. Hence the throughput does not scale with the number of threads. Even though the thieves are distributed among the threads in the ll-queue, the throughput of multi-view queue is slightly better. This is due to the smaller number of synchronization operation needed for mv-queue.

In workload 3, we observed that around 5% of the task retrieval operations resulted in stealing. In this case, similar to workload 1, the throughput of both queues is scaling. As expected, mv-queue scales better because of the smaller number of synchronization operations.

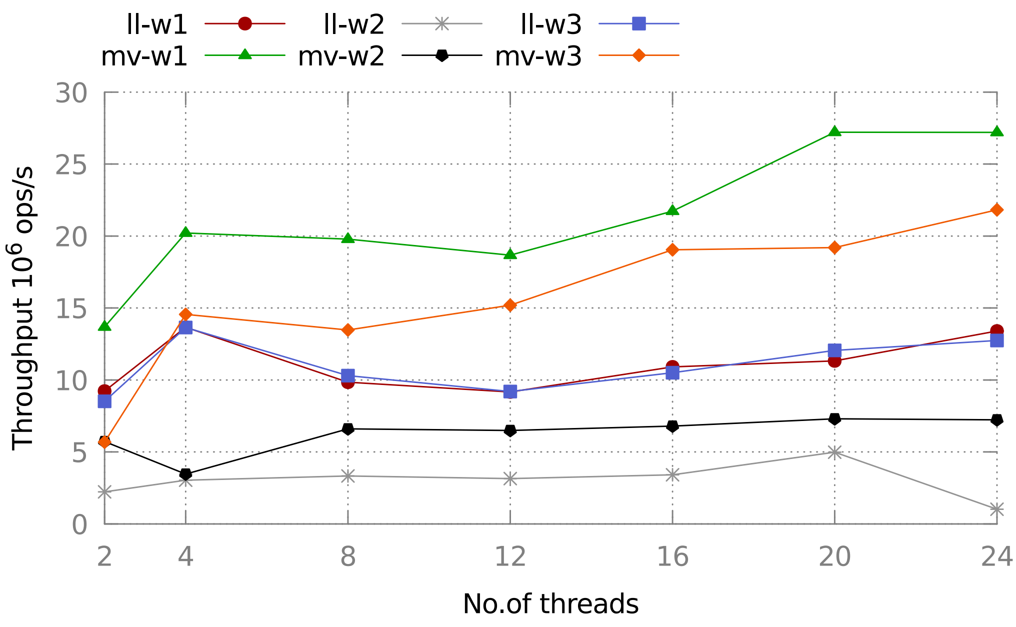


Figure 5.12: Throughput of work-stealing queue.

5.2 Mergeable Transactional Memory

5.2.1 Implementation

We implemented MTM as a library in Haskell. To compare it with traditional serializable transactions, we use the GHC's STM. GHC supports serializable transactions where multiple memory locations can be read and updated with in a `atomically` combinator. GHC's STM is tightly integrated with the runtime system and employs a number of optimization techniques with respect to GC interaction and scheduling. To approximate the runtime overhead incurred by implementing MTM as a library, we implemented an STM algorithm based on 2-phase-commit (2PC) similar to TL2 [28]. We use the 2PC implementation as another point of comparison.

Both MTM and 2PC implementations use spin locks for locking the resources during the commit phase. 2PC aborts and restarts the transactions, if the spin lock fails to guarantee serializability, while MTM retries until it acquires the locks and merges its changes. MTM and 2PC use a global counter that is used to generate monotonic unique ids for the versions.

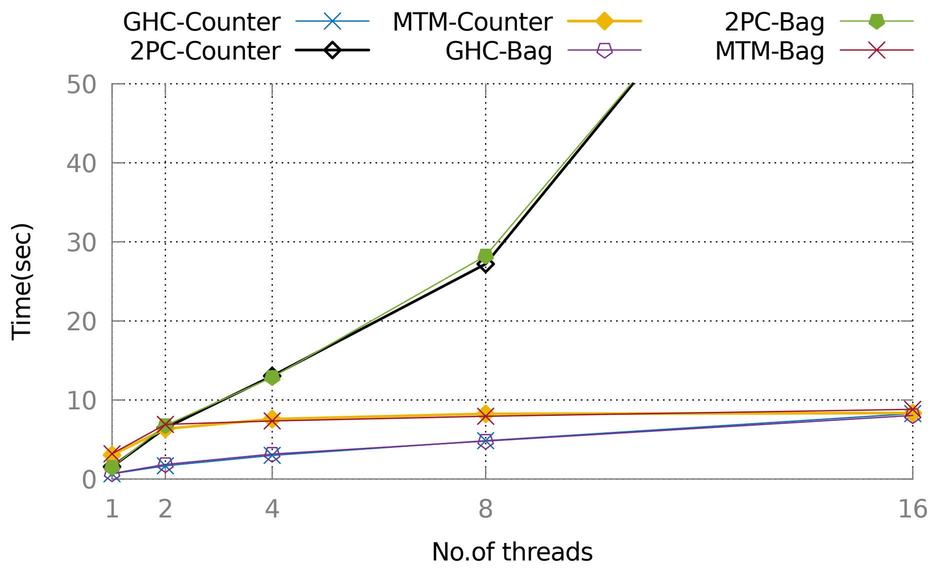


Figure 5.13: MTM micorbenchmark 1: Every thread updates once the same shared object in a transaction.

5.2.2 Evaluation

To evaluate the applicability of MTM we ran microbenchmarks, comparing our MTM implementation as Haskell library with a 2PC library implementation and GHC's STM implementation. All experiments were run on a Quad-core 2.4GHz Intel Xeon processor with two-way hyperthreading, under Linux 2.6.32-64-server Ubuntu x86_64 and GHC version 7.8.3. The results given are the averages taken over 10 runs for each benchmark.

Microbenchmarks: Counter and Bag In a first experiment, we compared the performance of a shared counter and bag under high contention. The STM variants implement the counter as a `TVar Int` and `TVar [Int]`, while MTM relies on a mergeable counter and bag, as introduced in Section 4.1. For the experiment, each thread repeatedly increments the same shared counter. In total, there were 2×10^6 increments distributed over the available number of threads.

As Figure 5.13 shows, the performance of the library version of STM degrades quickly while both MTM and GHC's STM handle the contention more gracefully.

To evaluate the throughput, we chose a workload where each transaction updates m randomly selected objects from a pool of n objects: the larger the pool (n), the lower the probability of contention; the larger the transaction size

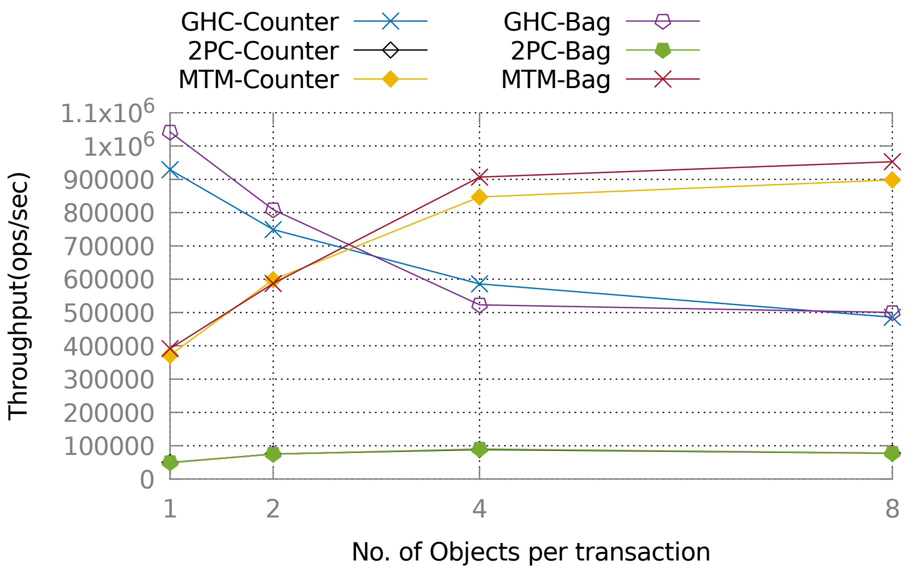


Figure 5.14: MTM microbenchmark 2 : Every thread updates M objects in a transaction

(m), the higher the probability of conflicts as it is more likely that transaction executions overlap. For $n = 8$ and various transaction size, MTM yields better performance than the STM implementations, even under low contention (Figure 5.14).

Application: K-means: To see how actual applications benefit from the MTM programming model, we reimplemented the K-means benchmark from the STAMP benchmark suite [70] in Haskell described in Algorithm 4.1.

For the version running GHC's STM and MTM a cluster centre is updated inside a transaction after processing every data point (here: 10^6 points). We also derived an alternative implementation to exploit the semantics of MTM, MTM-Opt, where all points assigned to some thread are processed together, and cluster centers are updated atomically. This version runs longer transactions, but has less frequent updates to cluster centers.

Both under high contention (Figure 5.15) and low contention (Figure 5.16), MTM-Opt outperforms GHC and MTM. In particular, MTM-Opt is scalable even under high contention in contrast to the other versions. The reason is that GHC's STM and MTM are blocking during commit, which prohibits scalability when the number of concurrent transactions is high. In the optimized version, commits are less frequent and transactions can run in parallel without the need for serializing the updates to shared memory.

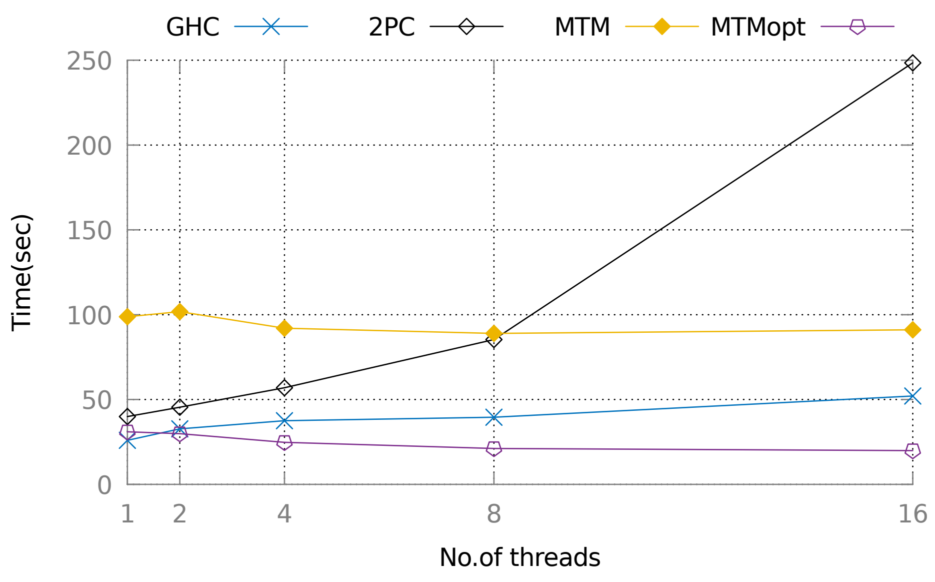


Figure 5.15: K-means: High contention.

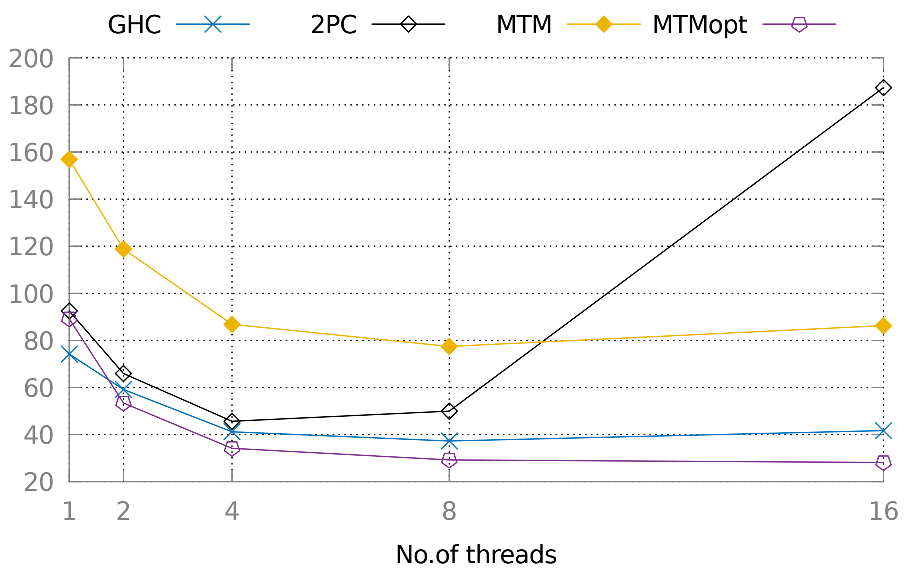


Figure 5.16: K-means: Low Contention.

5.3 Discussion

Linearizable implementations of data types like counters and queues are not scalable because all operations content on the same memory location and hence must be executed serially. Multi-view data types, on the other hand, allow to extract more parallelism by reducing the need to synchronize on each operation; instead they allow to operate on the local copy asynchronously. As the experiments demonstrate, the less need for synchronization has a massive impact on the scalability of multi-threaded programs. On the other hand, the global-local view model does not improve the scalability of data types compared to their linearizable implementations where concurrent operations do not contend. The operations on a set data type, even in their linearizable implementations, result in less contention and can be executed in parallel because in most cases they modify different memory locations.

CHAPTER 6

Related Work

Improving the scalability of shared memory synchronization is a widely studied topic in theory and practice. The literature spans across different areas from finding scalable synchronization mechanisms to relaxing the semantics of the data types. In this section, we summarize the related work in the areas of scalable concurrent data structures, software transactional memory, and relaxed consistency semantics.

6.1 Scalable concurrent data structures

Scalable designs of concurrent linearizable data structures have been an active area of research.

Michael et al. [69] proposes a non-blocking lock-free algorithm using a *compare and swap* synchronization primitive, and an algorithm that uses two locks for implementing practical queues. The lock-free queue algorithm is widely used including the concurrent queue in jdk[1]. Similary, there are numerous literature that explores lock-free and scalable algorithms for various data structures including linked-list [38], skip-list [53], priority queue [29], binary search tree [20] and so on. Some of these techniques can be adapted to implement lock-free *merge* and *strong* operations in MDT. We incorporated techniques of the lock-free enqueue operation from the Michael-Scott queue to implement lock-free merge operation in MDT queue.

Herlihy et al. [51] introduced a generic method for constructing wait-free data structures by applying universal constructions. The wait-free mechanisms are further explored in [86, 61]. Other generic synchronization mechanisms such as Flat combining and RCU to implement concurrent data structures are described below.

Flat combining (FC) [49] is a synchronization paradigm for implementing linearizable data structures using coarse locking instead of traditionally applied fine-grained locking technique. In flat combining, a data structure has

[1]java.util.concurrent.ConcurrentLinkedQueue

an associated global lock and a publication list. A thread publishes its read-/update requests to its record in the publication list and checks if the global lock is free. A thread that acquires the lock becomes a combiner and processes all pending requests in the publication list, and writes the responses in the corresponding thread-local record. The threads that could not acquire the lock spin on their publication record until a combiner writes the response to the record. FC thus improves the performance of a concurrent data structure access by reducing the contention on the lock and improving cache locality. A thread that acquires a lock processes all pending requests similar to the helper mechanisms in wait-free algorithms. FC is an effective approach for many, but not all, linearizable data structures. MDT, in contrast, allows executions of a large number of mergeable operations in parallel, thus effectively reducing the contention on the shared data structure. On the other hand, the strong operations and the merge of MDTs can employ FC mechanisms to mitigate the performance degradation due to the synchronization needed to access the global view. The techniques from FC-based queues [50] can be easily applied to our multi-view queue.

Read-copy-update (RCU) [68, 42] is a synchronization mechanism to allow processes to read a shared object while a concurrent modification is in progress. Similar to our model, multiple versions of the object are maintained so that readers observe a consistent state while a modification is in progress. However, RCU is suited only for a single writer-multiple readers scenario. Read-log-update (RLU) [67] is an improvement over RCU that allows concurrent writers. Unlike our model, concurrent writes are serializable which is achieved by serializing the writes or by fine-grained locking.

Distributed Queues (DQ) [44], on the other hand, relax the sequential specification of the data structure to implement a relaxed queue semantics. DQ consists of multiple FIFO queues and uses a load balancer or a Least Recently Used mechanism to distribute enqueue/deqeue operations among the partial queues to improve the overall performance.

The idea of using a per-core copy of a counter is exploited in sloppy counters [19]. Distributed Shared Objects [14] emphasize the need for non-shared local objects per processor to avoid expensive communication and contention, and propose locality-aware objects to improve cache locality.

Scalable NonZero Indicators (SNZI) [33] is a scalable linearizable implementation of counters with weak semantics. Instead of returning the exact value of the counter, an SNZI counter returns a boolean indicating whether the value is non-zero. They are more scalable than traditional counters and useful for specific use cases such as reference-counting garbage collectors where the exact value is not necessary.

6.2 Relaxed consistency models

Many models attempt to relax the strict semantics of linearizability [55] to achieve better performance. Quasi-linearizability [9] allows each operation to be linearized at a different point at some bounded distance from its strict linearization point. For example, a queue that dequeues in a random order, but never returns empty if the queue is not empty, is a quasi-linearizable queue. Quasi-linearizability allows more parallelism by allowing flexible implementations.

k-linearizability [59, 10] allows upto k operations to be re-ordered. Similar to the linearizability, k-linearizability requires that the history is equivalent to a k-serializable history S, where the responses to a call in S can be delayed up to the next k operations. For example, a dequeue operation on a k-linearizable queue returns one of the k oldest items.

Other models, such as quiescent consistency [54, 27] also define the correctness based on some sequential history, possible reordered, of the operations.

Weak and medium future linearizability [60] apply to the data types implemented using *futures* [58]. A call to an operation returns a future which can be later evaluated. For example, the following code depicts the invocation of two enqueue operations by a thread.

```
Future fx = queue.enq(x);
Future fy = queue.enq(x);
fx.eval();
fy.eval();
```

In weak future linearizability (weak-FL), the operation takes effect at some instant between the future creation and the return from the future's evaluation. Weak-FL thus allows re-ordering of operations from the same thread violating the program-order guarantees offered by many other relaxed consistency models including the guarantees provided by the global-local view model. In the above example, the two enqueues may be seen in different order by other threads. Medium future linearizability (medium-FL), in addition to the guarantees provided by the weak-FL, requires that the operations issued by the same thread to the same object take effect in the same order as their future creation. Similar to MDT queues, the implementations of both weak-FL and medium-FL queues allow optimizations by batching operations.

Local linearizability [43] requires that each thread-induced history (a subset of each thread operations) is linearizable. The implementation of a local linearizable dequeue resembles that of the work-stealing queue described in Section 3.2. Local linearizability applies only to container-type data structures such as pools, queues, and stacks.

Our work is complementary to these models, allowing a flexible combination of strong and weak updates to achieve different consistency semantics. Moreover, the notions like k-linearizability are not straightforward to extend to transactions.

6.3 Software transactional memory

Relaxing strong guarantees such as serializability has been considered by different STMs. Multi-versioned STMs [26] and Snapshot Isolation in STMs [75] allow read-only transactions to proceed without any conflicts. However, there may be aborts in case of write-write conflicts. Different approaches have been proposed to avoid abort or restarting of whole transactions by delaying some computations [77] to commit time and re-executing parts of transactions [26]. Twilight STM [17] allows transaction-specific conflict handling when inconsistencies are detected in commit phase.

Elastic Transactions [34] avoid aborts due to conflicts that do not affect the correctness. Elastic Transactions are useful in large data structures like linked lists where insertion/deletion on two different locations that induce an abort in traditional serializable transactions. Similar to Elastic Transactions, composable partitioned transactions [88] avoid unnecessary aborts by a model in which programmers can highlight the data the transaction needs by splitting a transaction into a *planing phase* and an *update phase*. SemanticTM [15] is yet another mechanism to avoid unnecessary aborts due to conflicts between readers and writers.

Unlike the above mechanisms, MTM focuses on introducing the conflict handling mechanisms at the object level.

Composable Memory Transactions [47] provide primitives for making serializable transactions composable in Haskell. The authors describe the benefits of Haskell's type system and monads to achieve safety and composability of transactions. We have adopted these techniques to implement the MTM monad. However, as MTM transactions never abort, we restrain from providing additional operations that support composability such as `retry` and `orElse`.

6.4 Transactional data structures

In most cases, Transactional Memory (TM) cannot exploit the design of existing concurrent linearizable data structures. Research in Transactional Data Structures aims for scalable data structures to use within a TM.

Transactional Boosting [52] is a method which allows operations on highly concurrent linearizable objects to execute using concurrent transactions, without the need for acquiring an exclusive lock on the object. A method's abstract lock issues a conflict only if two concurrent method invocations are non-commutative; therefore, concurrent commutative operation on an object can execute without aborting the transaction. Transactional boosting is a pessimistic approach by eagerly acquiring locks on the objects. Optimistic Transactional Boosting [48] is yet another methodology for transforming concurrent data structures to transactional objects. Both approaches take commutativity of operations as the base for detecting conflicts and thus achieving serializability. In contrast, MTM relies on object-specific conflict resolution which may allow non-commutative operations to occur in parallel.

Software Transactional Objects (STO) [56] implement parts of the transactional commit protocol within the object, that enables the data types to use semantic conflict detection rather than conflicting access to low-level untyped memory. STO thus prevents aborts due to false conflicts improving the overall performance. Transactional Data Structure Libraries [84] introduce transactional awareness to concurrent data structures to achieve scalable transactions.

Using Consistency Oblivious Programming (COP) [8], operations on a concurrent data structure can be designed to work efficiently with STM. Similar to the partitioned transactions, the idea is to split the code into multiple parts, where one part is safe to execute without any consistency check and a second part that validates the result from the first part before committing any updates.

6.5 Monotonic and mergeable data structures

The idea of concurrent updates to the replicas of an object and merging them to a convergent state was formalized by Conflict-free Replicated Data Types (CRDTs) [79, 80], which are now widely used in distributed replicated data systems. The properties of CRDTs, such as commutative operations and/or a semi-lattice structure, guarantee that concurrent updates can be safely executed on different replicas and later merged to get a consistent state on all replicas. A state-based CRDT takes its values from a semi-lattice. Two states of the same objects are merged by taking their least upper bound in the semi-lattice. Op-based CRDTs, on the other hand, exploit commutativity of updates to converge the states of two replicas deterministically. The high network latency and possible reordering of messages in distributed systems resulted in properties of CRDTs much different from what is required in a shared memory system. In this thesis, we show implementations of mergeable data types that are tailored for shared memory concurrent programs.

LVars [62, 63] are lattice-based data structures used for deterministic parallel programming in Haskell. The put operation changes an LVar's state in such a way that it monotonically increases in the lattice structure. Updates from concurrent threads on an LVar result in the same state, irrespective of which order they occur, thus guaranteeing determinism. The merge function always computes the least upper bound according to the lattice. LVars focus on deterministic and efficient execution for parallel programming models to support producer/consumer-like application.

We believe that lattice-based data structures such as LVars and CRDTs are beneficial for deterministic merging and verifying the correctness of applications. However, it is not trivial how to construct efficient merge operations in order to be useful in an optimistic transactional model to improve performance. In this thesis, we have discussed mergeable data structures which are not lattice structures.

Confluent persistent data structures [31, 36] allow operations on multiple versions of a data structure. These operations (e.g., concatenation, union) are constructed in a way such that previous versions are still accessible. Confluent persistent data structures are designed to perform these operations efficiently, in space and time. The applicability of these techniques in mergeable objects is an interesting topic for future work.

Even though no consolidated theory on mergeable data types exists in the shared memory ecosystem, there have been systems that use such types with restricted properties. Doppel [72] is a multi-core database that uses a mechanism called phase reconciliation to parallelize conflicting transactions. When a high contention workload is detected, Doppel switches to a split phase where the transaction updates the per-core copy of the objects of the contended data in parallel. After the split phase, the per-core copies are merged, and the transactions proceed to execute using classical concurrency control techniques. Whether transactions can be executed in the split phase, is decided based on the commutativity of operations, thus preserving sequential consistency.

6.6 Programming models

Maintaining per-thread replicas and performing updates on them has been considered by different programming models in the literature.

Burckhardt et al. [21, 65] propose a programming model for concurrent programs using revisions and isolation types. Each revision is considered a unit of concurrency. It executes operations on its local copy of the shared data concurrently to other threads. The modified data is visible to the main thread only after the revision is explicitly joined. The conflicts occurring due to concurrent updates are resolved using custom merge operation for cumulative types and a joinee-wins strategy for versioned types. Though MTM and the revisions model share similar semantics in executing operations on consistent snapshots and merging conflicting updates, they target different settings. The revisions programming model is a fork-join model and is suitable for short-running threads that operate mostly in isolation. MTM targets long-running threads which need to share data with other threads using transactional semantics periodically.

Global Sequence Protocol (GSP) [25] is a model for replicated and distributed data systems. Similar to our model, GSP has a global state which is represented as a sequence of operations. Each client stores a prefix of this global sequence. The updates by a client are first appended to the local sequence of pending operations and then broadcast to other replicas using a reliable total order broadcast protocol which enforces a single order on the global sequence. Since GSP addresses a distributed system's system model, with no bounds on message delays, there is much less control on replica divergence and liveness of the global sequence evolution. In contrast, we address a shared-memory concurrent architecture that allows to reason about bounds on divergence and stronger progress guarantees on the evolution of the shared state.

Distinguishing thread-local memory and shared memory within hardware memory models is explored in Acoherent Shared Memory (ASM) [57]. Instead of providing strong cache-coherence, the ASM model provides an abstraction based on software revision control, where threads can *checkout* data to their private memory and *checkin* to publish their updates to the shared memory. By allowing programs to control when to synchronize their cache, ASM reduces unnecessary communication between processors resulting in low energy and high-performance systems. However, ASM cannot handle type-specific merge that incorporates semantic conflict resolution. It would be interesting to see further applicability of the global-local view model at the hardware level.

6.7 Distributed systems

Weak consistency models, such as eventual consistency and causal consistency, are being widely researched and used in distributed systems. Many modern datastores (Cassandra [3], DynamoDB [83] etc.) provides eventual consistency with last-writer wins conflict resolution strategy. Other databases such as Riak [5] and Antidote [2] uses CRDTs to merge conflicting updates. The protocol, Cure [12], implemented in Antidote supports mergeable transactions with Transactional causal+ consistency semantics. SwiftCloud [89] is another system that supports client-side replication and uses CRDTs to deterministically merge conflicting updates, while supporting Transactional causal+ consistency. The semantics provided by mergeable transactions in these systems are different from MTM. In a distributed setting, providing a single global snapshot similar to MTM is expensive in terms of synchronization. The systems must consider the faults, network delays, and re-ordered message delivery to implement meaningful semantics correctly. While inspired by the development of weakly consistent distributed systems, the global-local view and MTM provides a solution that is suitable for a shared memory system.

Burckhardt et al. [24] present the idea of eventually consistent transactions and an implementation technique which provides this semantics. The Global Sequence Protocol [25] provides a programming model for replicated data stores and a weak consistency model relying on a global total order of updates. Though many recent works have studied eventual consistency in distributed database systems, few have addressed its applicability in multi-core programs.

Chapter 7

Conclusion

An ever-increasing number of cores in combination with heterogeneous access latencies at different cache levels have advanced the spectrum of attainable performance from multi-thread programming. At the same time, this breaks the transparency concerning data locality. As processor components become more numerous and spatially distributed, the cost of synchronization and communication among distant components will keep increasing in comparison to ones that are more closely located. When building internet-scale distributed systems, similar concerns lead to the design of scalable systems that limit global synchronization and operate locally when possible [66, 16].

Incorporating more information about the respective datatype semantics is crucial for datatype designs that are more parsimonious regarding synchronization. CRDTs succeed in capturing datatypes with clear concurrency semantics and are now standard components in internet-scale systems. However, they do not migrate trivially to shared-memory architectures due to high computational costs from merge functions, which becomes apparent once network communication is removed.

Inspired by CRDT, in this thesis we studied if relaxing strong consistency requirement helps to achieve scalability by avoiding synchronization. As a result, we presented the global-local view model as a base for a framework that allows capturing the semantics of multi-view datatypes. The global-local view distinguishes between local fast state and distant shared state where operations need to be synchronized. This distinction allows the Multi-view data types to explore the trade-offs in the design when using *weak* or *strong* operations. Multi-view Data Types in shared memory concurrent programs can be considered analogously to CRDTs in replicated distributed systems. However, it is important to notice that the implementations of these two data types must be adapted to the specific system at hand. We identified two important properties of MDTs that determine the actual efficiency of an implementation: persistence and mergeability. Our approach enables speedups in order of magnitudes while preserving the datatypes target behavior.

The mergeable semantics of multi-view data types enables the development of software transactions with relaxed semantics. Mergeable Transactional Memory provides an alternative to the often too strict semantics of traditional serializable transactions. Instead of aborting and re-executing the transactions, MTM merges its changes using the type-specific merge operation. Thus it extends the relaxed, but the scalable semantics of MDTs into composable transactions.

Future Work

Usability Our approach helps in scaling the applications, albeit at the loss of linearizability for individual data types and serializability for software transactions. However, it is not trivial for application developers to decide if the new semantics is correct for their applications.

Tools that help developers to analyze their programs to verify the correctness when using these relaxed semantics would be exciting to study in the future. There has been much research in verifying if geo-replicated applications meet their specification when using relaxed consistency together with CRDTs [90].

Data types We believe that the examples shown here are just the tip of the iceberg regarding applicable datatypes. It is essential to develop more data types with better implementations that benefits a wide range of applications.

It is entirely possible that further increments of the number of components involved will lead to a multi-tier model with more levels than the current binary, local vs. global, scheme.

Bibliography

[1] 40 years of microprocessor trend data. https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/. Accessed: 2018-05-25. (Cited on page 1.)

[2] Antidote DB. http://antidotedb.org/. Accessed: 2018-06-20. (Cited on pages 12 and 110.)

[3] Apache Cassandra. http://cassandra.apache.org/. Accessed: 2018-06-20. (Cited on pages 12 and 110.)

[4] The free lunch is over. http://www.gotw.ca/publications/concurrency-ddj.htm. Accessed: 2018-08-08. (Cited on page 1.)

[5] Riak KV. http://basho.com/products/riak-kv/. Accessed: 2018-06-20. (Cited on pages 12 and 110.)

[6] ZeroMQ. http://zeromq.org/. Accessed: 2018-06-20. (Cited on page 21.)

[7] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012. (Cited on page 12.)

[8] Yehuda Afek, Hillel Avni, and Nir Shavit. Towards consistency oblivious programming. In *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, pages 65–79, 2011. (Cited on page 107.)

[9] Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, pages 395–410, 2010. (Cited on page 105.)

[10] Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. On the availability of non-strict quorum systems. In *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, pages 48–62, 2005. (Cited on page 105.)

[11] Deepthi Devaki Akkoorath and Annette Bieniusa. Transactions on mergeable objects. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, pages 427–444, 2015. (Cited on page 22.)

[12] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*, pages 405–414, 2016. (Cited on page 110.)

[13] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, pages 483–485, 1967. (Cited on page 1.)

[14] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc A. Auslander, Michal Ostrowski, Bryan S. Rosenburg, Amos Waterland, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, and Livio Soares. Experience distributing objects in an SMMP OS. *ACM Trans. Comput. Syst.*, 25(3):6, 2007. (Cited on page 104.)

[15] Hillel Avni, Shlomi Dolev, and Eleftherios Kosmas. Proactive contention avoidance. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications - COST Action Euro-TM IC1001*, pages 228–241. 2015. (Cited on page 106.)

[16] Valter Balegas, Cheng Li, Mahsa Najafzadeh, Daniel Porto, Allen Clement, Sérgio Duarte, Carla Ferreira, Johannes Gehrke, João Leitão, Nuno M. Preguiça, Rodrigo Rodrigues, Marc Shapiro, and Viktor Vafeiadis. Geo-replication: Fast if possible, consistent if necessary. *IEEE Data Eng. Bull.*, 39(1):81–92, 2016. (Cited on page 111.)

[17] Annette Bieniusa, Arie Middelkoop, and Peter Thiemann. Brief announcement: actions in the twilight - concurrent irrevocable transactions and inconsistency repair. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 71–72, 2010. (Cited on page 106.)

[18] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient

multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996. (Cited on page 20.)

[19] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Tappan Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 1–16, 2010. (Cited on pages 2, 16, 38 and 104.)

[20] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 257–268, 2010. (Cited on page 103.)

[21] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 691–707, 2010. (Cited on page 109.)

[22] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, pages 283–307, 2012. (Cited on page 62.)

[23] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 271–284, 2014. (Cited on page 62.)

[24] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 67–86, 2012. (Cited on page 110.)

[25] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *29th European Conference on Object-Oriented*

Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic, pages 568–590, 2015. (Cited on pages 109 and 110.)

[26] João P. Cachopo and António Rito Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006. (Cited on pages 12, 65, 72 and 106.)

[27] John Derrick, Brijesh Dongol, Gerhard Schellhorn, Bogdan Tofan, Oleg Travkin, and Heike Wehrheim. Quiescent consistency: Defining and verifying relaxed linearizability. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 200–214, 2014. (Cited on page 105.)

[28] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, pages 194–208, 2006. (Cited on page 98.)

[29] Kristijan Dragicevic and Daniel Bauer. A survey of concurrent priority queue algorithms. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*, pages 1–6, 2008. (Cited on page 103.)

[30] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 109–121, 1986. (Cited on page 38.)

[31] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989. (Cited on page 108.)

[32] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 131–140, 2010. (Cited on page 86.)

[33] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. SNZI: scalable nonzero indicators. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 13–22, 2007. (Cited on page 104.)

[34] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In *Distributed Computing, 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings*, pages 93–107, 2009. (Cited on page 106.)

[35] Amos Fiat and Haim Kaplan. Making data structures confluently persistent. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.*, pages 537–546, 2001. (Cited on page 35.)

[36] Amos Fiat and Haim Kaplan. Making data structures confluently persistent. *J. Algorithms*, 48(1):16–58, 2003. (Cited on page 108.)

[37] Faith Ellen Fich, Danny Hendler, and Nir Shavit. Linear lower bounds on real-world implementations of concurrent objects. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*, pages 165–173, 2005. (Cited on page 2.)

[38] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 50–59, 2004. (Cited on page 103.)

[39] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 212–223, 1998. (Cited on page 20.)

[40] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. (Cited on page 12.)

[41] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java language specification, Java SE 8 edition. 2015. (Cited on pages 9 and 10.)

[42] Dinakar Guniguntala, Paul E. McKenney, Josh Triplett, and Jonathan Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with linux. *IBM Systems Journal*, 47(2):221–236, 2008. (Cited on page 104.)

[43] Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. Local linearizability for concurrent container-type data structures. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, pages 6:1–6:15, 2016. (Cited on page 105.)

[44] Andreas Haas, Michael Lippautz, Thomas A. Henzinger, Hannes Payer, Ana Sokolova, Christoph M. Kirsch, and Ali Sezgin. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*, pages 17:1–17:9, 2013. (Cited on page 104.)

[45] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010. (Cited on pages 3, 7 and 10.)

[46] Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, 2008. (Cited on page 74.)

[47] Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA*, pages 48–60, 2005. (Cited on page 106.)

[48] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Optimistic transactional boosting. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 387–388, 2014. (Cited on page 107.)

[49] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, pages 355–364, 2010. (Cited on page 103.)

[50] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Scalable flat-combining based synchronous queues. In *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*, pages 79–93, 2010. (Cited on page 104.)

[51] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. (Cited on page 103.)

[52] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pages 207–216, 2008. (Cited on page 107.)

[53] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer, 2006. (Cited on page 103.)

[54] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008. (Cited on page 105.)

[55] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. (Cited on pages 1, 15, 18, 62 and 105.)

[56] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 31:1–31:16, 2016. (Cited on page 107.)

[57] Derek R Hower. *Acoherent shared memory.* PhD thesis, The University of Wisconsin-Madison, 2012. (Cited on page 109.)

[58] Robert H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985. (Cited on page 105.)

[59] C Kirsch, Hannes Payer, and Harald Röck. Scal: Non-linearizable computing breaks the scalability barrier. Technical report, Technical Report 2010-07, Department of Computer Sciences, University of Salzburg, 2010. (Cited on page 105.)

[60] Alex Kogan and Maurice Herlihy. The future(s) of shared data structures. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 30–39, 2014. (Cited on page 105.)

[61] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN Symposium*

on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012, pages 141–150, 2012. (Cited on page 103.)

[62] Lindsey Kuper and Ryan R. Newton. Lvars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing, Boston, MA, USA, FHPC@ICFP 2013, September 25-27, 2013*, pages 71–84, 2013. (Cited on page 108.)

[63] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze after writing: quasi-deterministic parallel programming with lvars. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 257–270, 2014. (Cited on page 108.)

[64] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979. (Cited on page 8.)

[65] Daan Leijen, Manuel Fähndrich, and Sebastian Burckhardt. Prettier concurrency: purely functional concurrent revisions. In *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*, pages 83–94, 2011. (Cited on page 109.)

[66] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 265–278, 2012. (Cited on page 111.)

[67] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: a lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 168–183, 2015. (Cited on page 104.)

[68] Paul E. Mckenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998. (Cited on page 104.)

[69] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 267–275, 1996. (Cited on pages 84 and 103.)

[70] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: stanford transactional applications for multi-processing. In *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*, pages 35–46, 2008. (Cited on pages 65 and 100.)

[71] Mark Moir and Nir Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications.* 2004. (Cited on page 1.)

[72] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Tappan Morris. Phase reconciliation for contended in-memory transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 511–524, 2014. (Cited on pages 21 and 108.)

[73] Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999. (Cited on page 36.)

[74] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in STM. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 16–25, 2010. (Cited on pages 65 and 72.)

[75] Torvald Riegel. Snapshot isolation for software transactional memory. In *In Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT06*, 2006. (Cited on pages 11, 65, 67, 72 and 106.)

[76] David P. Rodgers. Improvements in multiprocessor system design. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, pages 225–231, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press. (Cited on page 1.)

[77] Wenjia Ruan, Yujie Liu, and Michael F. Spear. Transactional read-modify-write without aborts. *TACO*, 11(4):63:1–63:24, 2014. (Cited on page 106.)

[78] Michael L. Scott. Sequential specification of transactional memory semantics. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Jun 2006. (Cited on pages 3, 11, 65 and 67.)

[79] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, January 2011. (Cited on pages 3, 36, 62, 91 and 107.)

[80] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, pages 386–400, 2011. (Cited on pages 3, 13, 35, 36 and 107.)

[81] Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011. (Cited on pages 2 and 15.)

[82] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, pages 204–213, 1995. (Cited on page 65.)

[83] Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 729–730, 2012. (Cited on page 110.)

[84] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 682–696, 2016. (Cited on page 107.)

[85] Martin Sstrik. ZeroMQ. In *The Architecture of open source applications, Volume 2*, 2012. (Cited on pages 2, 15 and 22.)

[86] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 357–368, 2014. (Cited on page 103.)

[87] Werner Vogels. Eventually Consistent. *ACM Queue*, 6(6):14–19, 2008. (Cited on page 12.)

[88] Lingxiang Xiang and Michael L. Scott. Software partitioning of hardware transactions. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 76–86, 2015. (Cited on page 106.)

[89] Marek Zawirski, Nuno M. Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, December 07 - 11, 2015*, pages 75–87, 2015. (Cited on page 110.)

[90] Peter Zeller. Testing properties of weakly consistent programs with repliss. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2017, Belgrade, Serbia, April 23 - 26, 2017*, pages 3:1–3:5, 2017. (Cited on page 112.)

Deepthi Devaki **Akkoorath**

dd.thekkedam.org | deepthidevaki | deepthidevaki

Education

Technical Univerisity of Kaiserslautern *Kaiserslautern, Germany*
PhD. in Computer Science *Feb. 2014 - Oct. 2018*

University of Amsterdam *Amsterdam, The Netherlands*
M.Sc. in Computer Science *Sep. 2010 - Aug. 2012*

Amrita University *Kollam, India*
B.Tech. in Computer Science and Engineering *Aug. 2005 - Jul. 2009*

Work Experience

Camunda Services Gmbh *Berlin, Germany*
Software Engineer *Dec. 2018-*

Multicoreware. Inc *Chennai, India*
Software Engineer *Feb. 2013 - Dec. 2013*

Publications

Peer Reviewed

Global-Local View: Scalable Consistency for Concurrent Data types. *Aug. 2018*
24th International European Conference on Parallel and Distributed Computing (EuroPar).

FMKe: A Real-World Benchmark for Key-Value Data Stores. *Apr. 2017*
Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC).

Observing the Consistency of Distributed Systems. *Sep. 2016*
Proceedings of the 15th International Workshop on Erlang.

Cure: Strong semantics meets high availability and low latency. *Aug. 2016*
36th IEEE International Conference on Distributed Computing Systems (ICDCS).

Highly-scalable Concurrent Objects. *Apr. 2016*
Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC).

Transactions on Mergeable Objects. *Dec. 2015*
13th Asian Symposium on Programming Languages and Systems (APLAS).

White Paper

Antidote: the highly-available geo-replicated database with strongest guarantees. *Aug. 2016*
SyncFree Project.