Consistency, Isolation, and Irrevocability in Software Transactional Memory

Annette Bieniusa

Dissertation zur Erlangung des Doktorgrades der
Technischen Fakultät der
Albert-Ludwigs-Universität Freiburg im Breisgau

2011

Dekan:	Prof. Dr. Bernd Becker
Erstgutachter:	Prof. Dr. Peter Thiemann, Universität Freiburg
Zweitgutachter:	Prof. Dr. Satnam Singh, University of Birmingham
Tag der Disputation:	28. September 2011

Abstract

Software transactional memory (STM) is a promising paradigm for the development of concurrent software. It coordinates the potentially conflicting effects of concurrent threads on shared memory by running their critical regions isolated from each other in transactions in an "all-or-nothing" manner. When encountering a conflicting access to shared memory, a conflict resolution strategy decides which transaction has to revert its changes and is restarted.

However, this automatic coordination is sometimes too restrictive: non-revertible operations such as I/O operations are disallowed in a transaction and some transactions fail for minor conflicts that could easily be resolved. In addition, most STM schemes focus on shared-memory architectures where costly memory updates impede scalability.

This thesis tackles these limitations by exploring extensions to the standard STM schemes. It discusses two novel STM algorithms which broaden the scope for applicability of STM and provide insights into the strength and limitations of transactional programming.

Twilight STM proposes to extend STM with non-revertible actions and inconsistency repair. It separates a transaction into a functional transactional phase, and an imperative irrevocable phase, which supports a safe embedding of I/O operations as well as a repair facility to resolve minor conflicts. In contrast to other implementations of irrevocable transactions, twilight code may run concurrently with other transactions including their twilight code without data races. Twilight STM further allows the implementation of application-specific conflict resolution strategies. To analyze their influence on the semantics of the transactions, a formal framework for investigating consistency and isolation properties is developed and applied to different repair operations.

Decent STM transfers the STM paradigm to a distributed setting. It is a fully decentralized object-based STM algorithm with versioning. It relies on mostly immutable data structures, which are well-suited for replication and migration. A randomized consensus protocol guarantees consistency of shared memory. Transactions may proceed tentatively before consensus has been reached. Object versioning ensures that transactions read from a consistent memory snapshot, and the consensus protocol determines which transactions can merge in their effects and which transactions need to restart. Hence, delayed communication, e.g. caused by retransmissions in the transport layer, can only affect performance, not consistency.

Both STM algorithms have been implemented in functional, object-oriented, and imperative languages, on multi-core and distributed architectures. This comprehensive study points out the need for an enhanced STM interface for more flexibility and higher programming convenience. Benchmarks featuring various workloads show the scalability and competitiveness with state-of-the-art systems.

Zusammenfassung

Software transactional memory (STM) ist ein Programmiermodell zur Entwicklung nebenläufiger Programme. In diesem Modell werden potentiell konfligierende Datenzugriffe von Threads verhindert, indem kritischen Programmabschnitte voneinander isoliert in Form von Transaktionen ausgeführt werden. Tritt ein Zugriffskonflikt auf, wird er zur Laufzeit aufgelöst, indem Transaktionen ihre Änderungen rückgängig machen und die Berechnung erneut ausgeführt werden. Das Laufzeitsystem garantiert außerdem, dass die Modifikationen der transaktional verwalteten Daten konsistent und aus der Sicht der anderen Transaktionen atomar durchgeführt werden.

Eine solche transparente Konfliktstrategie ist bisweilen zu restriktiv: Transaktionen oft auf Grund von Konflikten abgebrochen, die einfach aufgelöst und kompensiert werden könnten. Sie erfordert außerdem, dass unumkehrbare Operationen, wie I/O, nicht innerhalb von Transaktionen ausgeführt werden. Außerdem ist das klassische STM-Modell auf Shared-Memory-Architekturen beschränkt, bei denen die Konsistenz der Daten in den lokalen Caches der Prozessoren durch Speicherbarrieren erzwungen werden.

Die vorliegende Dissertation beschreibt zwei neuartige Algorithmen, die zum einen das klassische STM um adaptive Konfliktstrategien und die Einbettung von I/O erweitert, zum anderen STM auch in verteilten System zur Anwendung bringt.

Twilight STM erlaubt die Einbettung unumkehrbarer Operationen sowie flexibler Konfliktbehandlungsstrategien für Transaktionen. Es teilt eine Transaktion in eine funktionale, transaktionale Phase, sowie eine imperative, irreversible Phase auf. Im Gegensatz zu anderen STM-Erweiterungen für irreversible Transaktionen können Twilight Transaktionen vollständig parallel ohne konfligierende Datenzugriffe ablaufen. Desweiteren erlaubt Twilight STM die Implementierung von anwendungsspezifischen Konfliktstrategien. Um deren Semantik zu analysieren, stellt die Dissertation ein formales System zur Verfügung und wendet es auf verschiedene Strategien an.

Decent STM transferiert STM in eine verteilte Ausführungsumgebung. Es ist ein dezentraler, objekt-orientierter STM-Algorithmus, der auf Datenversionierung aufbaut. Da Versionen nach ihrem Transfer in das dezentrale Speichersystem unveränderlich sind, können sie einfach repliziert und migriert werden. Ein randomisiertes Konsensusprotokoll garantiert die Konsistenz bei transaktionalem Datenzugriff. Darüberhinaus können Transaktionen optimistisch bereits vor Erreichen eines Konsensus ihre Berechnungen starten. Verzögerungen im Datentransfer beeinträchtigen daher lediglich die Geschwindigkeit des Programms, nicht jedoch die Konsistenz der Daten.

Die STM-Algorithmen sind in funktionalen, objekt-orientierten und imperativen Programmiersprachen, für Multi-Core-Architekturen und verteilte Systeme implementiert. Die vorliegende Dissertation bietet einen Überblick über die praktische Anwendbarkeit von STM und unterstreicht den Bedarf einer Erweiterung des Basismodells. Verschiedene Testprogramme belegen die Skalierbarkeit der Systeme und zeigen die Wettbewerbsfähigkeit in Bezug auf andere STM-Systeme.

Acknowledgements

I am heartily thankful to my supervisor, Peter Thiemann. Without his supervision and support, this thesis would have not been possible. He introduced me to the world of research, the art of (functional) programming, and the pitfalls of academia. Thank you, Peter, for your never-ending patience and encouragement, and for running a whole night through Tokyo in order to find this DanceRevolution machine!

Furthermore, I would like to thank Satnam Singh for being the co-reviewer of this thesis, without hesitation, despite having a tight schedule.

I also owe many thanks to Thomas Fuhrmann. He sparked my interest for distributed systems and was always willing to discuss even the most quirky ideas. Thank you, Thomas, for your support and mentoring, and for three unforgettable summer schools!

The DFG graduate school Embedded Microsystems and the German Federal Ministry of Education and Research supported me gratefully under the JCell project.

Thank you: Arie Middelkoop, for late-night hacking and pushing me towards my goals; Stefan Wehr, Markus Degen, Phillip Heidegger, and Konrad Anton, for innumerable coffee breaks and being great colleagues; Markus Degen, for always having an open ear; Konrad Anton, for emergency cookies; Felix Atmannspacher, and Paresh Paradkar, for implementing prototypes and finding the bugs, and their enthusiasm for STM; all members of the grad school on Embedded Microsystems, for introducing me to the fascinating world of very small systems and a great time in Japan; Tim Nonner and Christian Gunia, for sharing the grad school experiences; Berit Brauer, Manuela Kniss, and Martin Preen, for their administrative support; Fabian, Florian and Andrea Fuhrmann, for offering the visitor bed in their playroom during project meetings; Uta Thiemann, for barbeques and books; Ulrike Pado and Rotraud Miessner, for wine, chocolate, and girls talk. And thanks to all friends from all over the world for being what they are!

Zum Schluss möchte ich meinen Eltern danken, für ihre Unterstützung und Liebe in allen Lebenslagen.

Contents

List of Figures

List of Listings

List of Algorithms

Chapter 1.

Software Transactional Memory

1.1. Concurrent Programming

Concurrent programming [7, 27] is the art of controlling (pseudo-) simultaneous execution of multiple interacting computations. The primary objective for using this programming technique is to increase the application throughput and to use available hardware resources efficiently. Furthermore, there are problem instances for which concurrent programming is a natural paradigm (e.g., client-server architectures and event-based architectures).

Programming in a concurrent style is difficult:

- The partitioning of a program into several *threads*,[1] each executing a part of the whole program's task, is in general challenging because of data and control flow dependencies between the threads.
- The coordination of multiple threads is a complex task which requires communication among the threads. This communication overhead can impede the scalability of the system.

Synchronization[73] refers to the coordination of simultaneously running threads and the maintenance of a coherent view of data shared between threads. Synchronization requires communication. Communication between threads comes in two major flavors, via shared memory or via message passing. Access to shared memory can be coordinated in many different ways, among them locking and transactional memory. The following subsections give a brief overview on two classic paradigms, indirect communication using mutual exclusion, and direct communication via message passing. We then sketch a third modern paradigm, transaction-based synchronization, that takes a different approach to synchronization.

1.1.1. Lock-Based Synchronization

Shared memory imposes only little overhead on data synchronization. The classical approach to synchronization in this setting grants code fragments (*critical sections*)

[1] This thesis considers "thread" and "process" as synonyms and uses the word "thread" throughout. The usual reading is that threads run in a shared address space whereas processes may run in separate address spaces.

only *mutually exclusive access* to the shared memory. This *locking* of resources guarantees that a thread obtains exclusive access for some time to complete its memory operations undisturbed. Unfortunately, excessive locking can reduce parallelism. Even worse, *deadlocks* can arise when threads that have already obtained some locks are blocked, mutually waiting for further locks to be released. Similarly, threads can end up in a *livelock* where their state constantly changes but no progress is made. Explicit synchronization via locking is commonly thought to be error-prone due to its delicate semantics.

A *monitor* [40] mediates all accesses to and modifications of some portion of shared memory. It guarantees that the procedures associated with the monitor obtain mutually exclusive access to the resources guarded by it. The standard implementation is via locks (generated by the compiler or runtime environment).

A programmer aiming for ultimate performance in an application usually applies locked-based algorithms and hence has to tackle the aforementioned obstacles. In particular, he must rely on explicit reasoning with the synchronization primitives to construct a correctness proof. This thesis investigates an alternative approach which provides stronger safety guarantees, and also possibly better performance for specific classes of problems.

1.1.2. Message-Based Synchronization

In a parallel architecture with *distributed memory*, threads do not share a common address space. Thus, it is more appropriate to manage data sharing via *message passing* (MP) than to grant remote memory access. In this setting pairs of corresponding send and receive operations transport data between threads. The message passing interface (MPI [69]) and its successor MPI2 [29] define standard APIs that many programming languages implement. Message passing is also the basis of the Erlang programming language [3, 4]. Communication operations can be classified with respect to the following categories:

Point-to-point vs. global A thread can send a message either to one other thread or to all other threads (*broadcast*). It is also possible to group threads for communication purposes.

Synchronous vs. asynchronous In synchronous mode, the sender blocks until the receiving thread has started its receive operation. In asynchronous mode, the send operation does not block. Instead, the run-time system buffers the message until the receiver requests it.

Accumulation vs. non-accumulation A special receive operation can accumulate messages from multiple threads with a specified reduction operation. The receiver sees only the final result. Alternatively, all messages are sent directly to the receiver.

1.1.3. Transaction-Based Synchronization

Software Transactional Memory (STM) [67] is seen by some as a more user-friendly approach to synchronization in a shared memory setting. It offers a high-level mechanism and shifts the implementation of mutual exclusion as well as some data management tasks to the runtime environment.

Transactional memory evolved from ideas of transaction processing in database systems and borrows much of its terminology. Central to STM is the notion of an *atomic block* which is used to encapsulate the accesses to the shared memory in a safe manner. A *transaction* starts when entering an atomic block and ends when leaving it. The STM system guarantees that the computations inside a transaction either execute as a whole or not at all. Moreover, other concurrently running threads cannot observe intermediate states of the computation inside an atomic block. These intrinsic features are referred to as *atomicity* and *isolation*.

The implementation of STM has to cope with concurrently running transactions accessing and modifying the same memory locations. Transactions are in *conflict* if they access the same memory location and at least one transaction is modifying the content. To ensure the absence of conflicts, a conflict detection mechanism checks the system's consistency and eventually arbitrates between the conflicting parties. In general, this arbitration may lead to the abort of a transaction and a retry later in time. A transaction that finishes its computation without encountering a conflict *commits* when leaving the atomic block. Otherwise the transaction performs a *rollback*. In this case, all its effects of the atomic block must be undone and the former state is restored. The runtime has to provide the means to perform such a rollback, for example by logging of values. To sustain the isolation property, many implementations forbid irreversible operations, such as I/O, inside of atomic blocks. Often a type system restricts the mutation of shared data to atomic blocks. Software Transactional Memory (STM) gives high-level guarantees about the interaction of concurrent threads. In the STM paradigm, read or write accesses to shared memory are only permitted inside a transaction, where a thread is guaranteed an isolation property which roughly states that it never is confronted with an inconsistent memory snapshot.

The descriptions of lock-, transaction-, and message-based synchronization suggests that these concepts are fundamentally distinct. Nevertheless, there are many hybrid forms. For example, STM systems can be implemented with a 2-phase commit protocol using some kind of locking when checking and writing data at the end of a transaction [20].

Most modern programming languages support these paradigms, either by linguistic means or through libraries.[2]

[2] There are also pure hardware and hybrid platforms that implement thread communication. This thesis nevertheless concentrates on the software implementations.

1.2. Designing an STM system

The design space for STM systems extends not only to the semantics of the atomic blocks themselves, but also to the behavior of the system when interacting with non-transactional computations. The following overview introduces a classification of STM systems with respect to the most important design choices:

Atomicity *Strong atomicity* ensures that an atomic block executes in isolation with respect to all other computations. *Weak atomicity* guarantees isolation only with respect to other atomic blocks.

Conflict detection *Pessimistic conflict detection* checks the validity of read and written data progressively, so conflicts are detected early and transactions which are bound to fail are aborted quickly. *Optimistic conflict detection* postpones data validation until the end of an atomic block.

Granularity of conflicts Conflicts may occur at the *object level*, the *cache-line level*, or the *word level*. Whereas object conflict detection is a sensible choice in object-oriented programming languages, the other options are useful in less structured settings.

Data versioning *Eager versioning* performs an in-place memory update during a transaction. It saves overwritten values in an undo-log structure for reconstruction on a potential rollback. With *lazy versioning* each atomic block maintains its own local write buffer whose values are later on committed to the shared memory.

Nesting A transaction [56] is nested when an atomic block is enclosed in another atomic block. One approach to define the semantics of nested transaction is to *flatten* the enclosed transactions with the outermost one into a single transaction. *Closed nesting* performs a commit check and possible rollback of a nested transaction, but ensures that the nested transaction's effects only become globally visible when the outermost transaction commits. *Open nesting* allows nested transactions to commit their result globally and irreversibly before the outer transaction has finished its execution.

System architecture On a *shared-memory architecture*, synchronization relies on the specification for memory access as provided by the chip vendor and operating system. Normally, the architecture ensures cache coherence and data consistency across the processing nodes with some form of hardware support. In *distributed architectures* (also called NUMA architectures), the participating nodes have to communicate explicitly to make remote data available to the other processors. This requires the usage of some network layer which connects the node.

The design of an STM system is usually strongly influenced by the host language, in particular its memory model. The memory model specifies when updates to

shared data are propagated to the threads having access to this data. Often the synchronization has to be explicitly triggered by memory barriers. For example, the Java memory model (JMM) applies memory barriers upon access to volatile fields or when entering `synchronized` blocks. When integrating an STM into an existing programming language, the semantics of its present synchronization operations has to be taken into account.

For a detailed overview on the design space in Transactional Memory, state-or-the-art implementation, and TM's history, we refer to the extensive survey by Harris, Larus, and Rajwar [47].

1.2.1. Semantics of Transactions

The semantics of the access to globally shared memory is just one, but yet the most important aspect in the design of an STM system. It has great influence on the system's behavior because it specifies the actual semantics of programs that are executed within the system.

As transactional memory has its origin in concurrency control for databases, the synchronization mechanisms for shared memory access in STM are strongly related to their counterpart in database transactions. Research on semantics for transactions therefore applies to both areas equally though the actual systems that are implemented differ in their requirements for size, speed, or operational reliability.

Berenson and co-workers [8] have defined several isolation levels by characterizing the phenomena that the semantics of an isolation level admits[3].

Dirty reads Transaction T_1 modifies a data item. Another transaction T_2 reads that data item before T_1 performs a commit or rollback. If T_1 then performs a rollback, T_2 has read a data item that has never been committed and so never existed in the global view of the shared state.

Non-repeatable reads Transaction T_1 reads a data item. Another transaction T_2 then modifies the data item and commits. If T_1 then attempts to reread the data item from shared state, it receives a modified value which differs from its first read access.

Dirty writes Transaction T_1 modifies a data item. Another transaction T_2 then further modifies this data item before T_1 finishes. If T_1 or T_2 then perform a rollback, it is unclear what the restored data value should be.

Lost update Transaction T_1 reads a data item. Another transaction T_2 updates the data item, then T_1 (based on its earlier read value) updates the data item and commits.

[3] We omit the anomalies that are defined with respect to database predicates as there is no direct equivalent in STM.

Figure 1.1. Example: Violation of invariants under write skew.

`Invariant: x + y < 5, initially: x = y = 0`

Thread 1	Thread 2

`atomic {`
`  if (x + y < 2)`
`  x = 3;`
`}`
	`atomic {`
	`  if (x + y < 2)`
	`  y = 3;`
	`}`

Read Skew Transaction T_1 reads a data item x. Then, another transaction T_2 updates data items x and y, and commits. If now T_1 reads y, it may see an inconsistent state.

Write Skew Transaction T_1 reads data items x and y consistently. Then, T_2 reads x and y, updates x and commits. Finally, T_1 writes y. If there was a constraint between x and y, it might be violated.

The problems resulting from computations that exhibit any of these anomalies in their execution are well known.

Consider for example the code snippet in Figure 1.1. It shows two threads that access concurrently memory locations `x` and `y` in a transactional way. In a system allowing write skews, the concurrent write access to these variables does not give rise to a conflict. Thus, `x = y = 3` is a possible final state, though it breaks the invariant `x + y < 5` which is respected by each atomic block individually.

In databases, the admissibility of anomalies corresponds to different ANSI isolation levels. The more of these anomalies a transactional system prohibits, the higher are the costs for synchronizing the access to shared data, and the higher is the likelihood of aborted transactions and rollbacks.

In transactional memory systems, the main research focus is currently on systems with an isolation level of *opacity*. In contrast to database transactions, an opaque transactional system also requires not only committing, but also aborting transactions to operate on a consistent memory snapshot. STM systems that do not implement opacity for their atomic code blocks have to provide safety mechanisms like sand boxing to prevent *zombie transactions* that can neither abort nor commit.

For an example, consider the code in Figure 1.2. Thread 1 increments both `x` and `y` to ensure the invariant that `x = y`. In a system which admits dirty reads or read skews, the updates of thread 1 might become partially visible to concurrently running threads. Thread 2 might then observe the increment of `x`, but not of `y`. In this case, the transaction in thread 2 would get stuck in the non-terminating loop and become a zombie transaction.

Figure 1.2. Example: Non-termination under dirty reads or read skew.

```
Invariant: x = y, initially x = y = 0
```

Thread 1	Thread 2

`atomic {`
`  x++;`
`  y++;`
`}`
	`atomic {`
	`  if (x != y)`
	`  while (true){}`
	`}`

1.3. Contributions of this thesis

This thesis explores several extensions to the standard STM schemes. To this end, it introduces two novel STM algorithms which broaden the scope for applicability of STM and provide insights into the strengths and limitations of transactional programming.

The thesis makes the following contributions:

1. Twilight STM is a transactional memory system designed for shared-memory architectures. It augments transactions with non-reversible operations and allows introspection and modification of a transaction's state. The extended API provides a wide test-bed for exploring different transactional semantics.
   - The thesis introduces the features of Twilight STM and highlights the underlying design principles.
   - It defines an algorithm for Twilight STM and shows its correctness.
   - It formalizes a monadic core calculus for STM with opacity and snapshot-isolation semantics and proves its correctness. It extends the core calculus to Twilight STM and verifies the semantic equivalence of specific Twilight transactions with the other calculi.
   - It gives an account of implementations of Twilight STM in different programming languages (Haskell, C, Java) and discusses the results of evaluating several benchmarks.
2. Decent STM is a fully decentralized STM system for distributed systems based on a messaging scheme. A multi-versioning scheme allows the construction of consistent memory snapshots for the transactions. It provides a range of commit protocols which give different fairness and progress guarantees for concurrently running transactions.
   - The thesis discusses the challenges in constructing an STM for a distributed environment and how Decent STM meets these challenges.

- It describes the architecture of Decent STM and the interaction between the components.
- It presents a new randomized consensus protocol and a scheme for constructing consistent memory snapshots from multi-versioned data in a decentralized way. Based on these results, it defines an algorithm for Decent STM and shows its correctness.
- It reports on implementations of Decent STM as Java libraries for a distributed environment with explicit network communication and for multi-cores with implicit shared-memory synchronization. In addition to the actual runtime environments, the implementation also provides a tool for transforming code with transactional annotations to alleviate the development of STM applications.
- It summarizes the results of running several benchmarks on the implementations and discusses their performance with regard to the underlying system architecture, the workload, and variations in the transactional semantics.

The thesis is structured into two main parts. Part I is dedicated to Twilight STM. Part II is committed to Decent STM. The thesis finishes with a discussion of related work and the conclusion.

Some of the material presented in this thesis is based on the following publications by the author of the thesis and others:

- A brief announcement published at PODC 2010 [10] (joint work with Arie Middelkoop and Peter Thiemann) outlines the main ideas behind Twilight-STM.
- A paper published at ESOP 2011 [11] (joint work with Peter Thiemann) establishes the correctness proofs for opaque TM algorithms based on traces.
- A paper published at IPDPS 2010 [9] (joint work with Thomas Fuhrmann) introduces the basics of DecentSTM and evaluates the algorithm on a theoretical and practical basis.

Under the supervision of the author of this thesis, Felix Atmanspacher implemented the Twilight STM for C and Paresh Paradkar implemented the prototype of the distributed version of Decent STM.

Part I.

Twilight STM

Chapter 2.

Introduction

Software Transactional Memory (STM) is a promising paradigm for the development of concurrent software. It provides fine-grained deadlock-free mutual exclusion in a scalable and composable way. The underlying concept is simple: computations on shared memory are grouped into blocks that are executed atomically and in isolation on a consistent memory snapshot by transactions. Conflicting accesses to shared memory are detected at run time and resolved transparently. The prevailing transaction continues and commits its write operations while the other transactions are aborted and restarted.

The advantages of STM come at a price. Non-reversible operations like I/O do not mingle well with the transparent rollback mechanism of transactions. Therefore, most implementations of STM rely on programming conventions to exclude non-reversible operations from transactions. However, the need for this functionality has been demonstrated by several extensions of STM with *irrevocable transactions* [78, 70] that may contain non-reversible operations. As further elaborated in Section 12, each of these proposals severely constrains concurrency by serializing the execution of irrevocable transactions.

Another challenge when designing an STM algorithm is the choice of an appropriate strategy for contention management. High contention on heap locations can lead to poor performance and poor scalability if a transaction is repeatedly restarted because of conflicts. While the problem with contention is common among all concurrency paradigms, the transparent nature of TM hinders the detection of highly contented locations and often it is impossible to come up with an application-specific solution as the contention strategy is hard-coded into the system.

This part of the thesis presents Twilight STM, an STM system which safely augments transactions with non-reversible operations and allows introspection and modification of a transaction's state.

Twilight STM splits the code of a transaction into a (functional) atomic phase and an (imperative) twilight phase. The atomic phase implements the standard semantics of atomicity, consistency and isolation. Code in the twilight phase executes before the decision about a transaction's fate (restart or commit) is made and can affect its outcome based on the actual state of the execution environment. To this end, the Twilight API has operations to detect and repair read inconsistencies as well as operations to overwrite previously written values. For compatibility with other STM systems, a transaction can only finish successfully if the twilight code

resolved all inconsistencies. Otherwise it restarts the transaction.

Twilight STM also permits the safe embedding of I/O operations with the guarantee that each I/O operation is executed only once. In contrast to other implementations of irrevocable transactions, twilight code may run concurrently with other transactions including their twilight code in a safe way. However, the programmer is obliged to prevent deadlocks and race conditions when integrating I/O operations that participate in locking schemes.

Outline

- Chapter 3 introduces the basic concepts of Twilight STM from a programmer's point of view. Section 3.1 highlights the differences between standard STM transactions and twilight transactions. The API of Twilight STM laid out in Section 3.2, followed in Section 3.3 by examples which show how to apply Twilight features in practice. The chapter closes with a discussion on the limitations of the Twilight approach in Section 3.4.

- Chapter 4 presents the fine-grained locking algorithm underlying Twilight STM. After introducing the global meta data such as the global timer in Section 4.1, the semantics of the STM operations, repair operations, and methods for marking is explained (Section 4.2). Section 4.3 concludes with a discussion of the algorithm's deadlock-freedom and progress, and consistency of transactions.

- Chapter 5 formalizes the static and dynamic semantics of Twilight STM. The concept of execution traces is informally introduced with some examples in Section 5.1. Section 5.2 gives a formalization of an STM with lazy updates as a monadic lambda calculus. The formal system is then shown in Section 5.3 to implement the isolation level of opacity. Further, a slightly modified formal system is proved to implement snapshot isolation in Section 5.4. Finally, the formal system is extended with Twilight operations and it is shown that twilight transactions implement different isolation levels, depending on the operations in their twilight zones (Section 5.5).

- Chapter 6 describes implementations of Twilight STM in different kind of languages. Section 6.1 reports on a word-based implementation in C which is evaluated on the STAMP benchmark suite and compared against a state-of-the-art STM. Section 6.2 comments on an object-based implementation in Java which offers open nesting for twilight transaction. Lastly, the implementation in Haskell which is evaluated on the Haskell STM benchmark and compared against GHC's STM system (Section 6.3).

Chapter 3.

A Tour of Twilight STM

In this section, we first give an informal account of Twilight STM. After motivating the design, we introduce the extensions to the standard STM API for Twilight STM and explain them from a programmer's point of view. We then show several use cases for Twilight operations in practical settings:

- for printing debug output for transactions in a reliable way,
- for implementing fine-grained conflict detection for linked data structures with commuting operations, and
- for including mutex via locks and network communication into transactions.

3.1. Twilight transactions

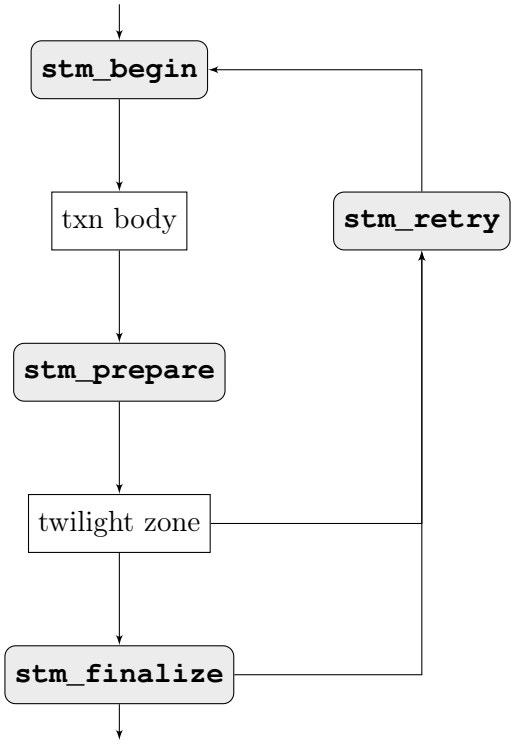
When implementing programs in the STM paradigm, the scope of a transaction is defined by the atomic block that is executed by the transaction. Depending on the programming language and possible syntax extensions, an atomic block is sometimes explicitly specified with a special syntactical construct:

```
atomic {
   x = 3;
   if (y != 45) {
    ...
   }
}
```

An atomic block is typically de-sugared by the compiler or a preprocessor into calls to native functions interfacing the runtime environment or designated methods in the host language (see also Section 10.3):

```
stm_begin();
stm_write(x,3);
if (stm_read(y) != 45) {
    ...
}
stm_end();
```

Figure 3.1. Workflow of a Twilight transaction.



In this section, we exercise the latter version to make a clear distinction between access to transactionally managed memory and other types of memory. This convention is also employed by the implementation of Twilight in C (see Section 6.1).

3.1.1. Workflow of twilight transactions

A standard transaction follows a workflow as follows: A call to **`stm_begin`** initializes the transaction's state for handling transactional read and write accesses and detecting of conflicts. When the execution reaches **`stm_end`**, the commit protocol is launched which makes the updates that are issued by the transaction visible to other transactions. Two-phase commit [23] is a common commit protocol that proceeds in several steps. The system first acquires exclusive access to the memory locations that are being updated, then checks for intermediate updates that might violate the transaction's consistency, and only updates the memory locations if there are no consistency violations. If the consistency check fails, the transaction is aborted, its state is reverted, and the evaluation of the atomic block is restarted.

Figure 3.1 displays the workflow of a twilight transaction. As with standard STM systems, it splits a transaction's commit phase into two phases.

The first phase is similar to the transactional execution of the atomic block.

Reads and writes of transactional variables are passed to the memory management system where the STM algorithm imparts a consistent view of the shared state to the transaction.

After executing the transaction's body, the transaction obtains exclusive access to the memory locations that it wants to update in **`stm_prepare`**, and checks for the status of the variables that it has read. This approach is akin to that of two-phase commit.

Diverging from standard protocols, the transaction does not restart in the presence of conflicts. Instead, it continues with executing the *twilight zone*. In this additional phase, a transaction can resolve inconsistencies that have been detected in **`stm_prepare`**. By reloading the current values of the updated memory locations and re-computing values that are going to be updated, repair actions can fix the inconsistent state of the transaction. The programmer has to specify these patches by means of dedicated operations from the Twilight API. Alternatively, if the modifications that have been issued by other transactions are not semantically invalidating the transaction, the programmer can state that the transaction's updates are still to be issued and that the inconsistencies are to be ignored.

The irrevocable nature of the twilight state offers the programmer the chance to issue any non-revertible actions. Based on possible input from the irrevocable operations, the values to be committed can still change and may be adapted.

Eventually, the transaction finishes with a call to **`stm_finalize`**. It then commits its changes to shared state and releases its exclusive access to the updated memory locations.

3.1.2. Properties of Twilight STM

Twilight STM's semantics can be characterized by the following properties:

Atomicity. Aside from explicitly observed intermediate updates, all read and write operations on shared memory within a transaction appear to take place at a single point in time.

Consistency. Transactions always read from a consistent memory snapshot.

Progress. All Twilight API operations are deadlock-free if the execution of twilight code does not involve waiting for other transactions due to non-transactional synchronization.

Irrevocability. A transaction executing its twilight code can always finish successfully.

Independence. Transactions executing their twilight code concurrently have disjoint write sets.

The first two properties specify the transactional behavior. The transparent conflict detection and resolution within a transaction hides the complexity of concur-

rently running and interacting threads and provides the illusion of an atomic execution of transactions to the programmer. The consistency property guarantees that all memory locations are read in a transaction as coming from a single point in time. In case of conflicts, out-dated values are consistently re-read. Results from computations on them can only be out-dated, but not wrong as the algorithm prevents inconsistent memory snapshots. Furthermore, the transaction can also observe the current values at commit time and has a chance to adjust its results.

The last three properties are related to the twilight code. The progress condition states that if a thread attempts to execute an atomic block successfully, some thread will succeed in doing so. This means that although a transaction might be forced to restart, the overall system is making progress because another transaction finishes successfully. Irrevocability ensures that twilight code can perform I/O and other irrevocable actions safely, and independence enables it to adjust the outcome of the transaction without further re-validation.

3.2. The API of Twilight STM

Listing 3.1 gives an overview of the Twilight API for C. In contrast to the object-based versions for Java and Haskell, the Twilight API for C operates on word-sized items of data where the actual size depends on the underlying hardware architecture. The actual implementation supports all word-sized data types generically. Other data types such a long integers or doubles are processed with the help of customized handlers.

We explain the operations now in greater detail. To this end, we distinguish between general system operations, the standard transactional operations, and the extensions for Twilight STM.

3.2.1. STM system operations

The transactional memory system is initialized with **`stm_start`**. It sets up the infrastructure for the conflict detection and resolution. With **`stm_shutdown`**, the system is closed down and the acquired resources are released again.

3.2.2. Basic transactional operations

A transaction is initialized and started with **`stm_begin`**. In the body of the transaction, operations on shared data have to be conveyed via **`stm_read`** and **`stm_write`**.

Twilight STM performs write operations lazily. It records the new values in a transaction-local write set. These write operations get flushed to the shared memory only on completion of the transaction.

The read operation yields the value of a memory location which originates from the same snapshot as the transaction's read accesses so far. Upon the first access, the current value of the variable is recorded in the read set of the transaction if is

Listing 3.1 Twilight API for C.

```
/* STM system operations */
void stm_start()
void stm_shutdown()

/* Basic operations */
void stm_begin()
void stm_end()
void *stm_read(word *var)
void stm_write(word *var, word val)
void *stm_alloc(int size)
void stm_free(word *var)

/* Regions */
tag stm_new_tag()
void stm_mark(tag t, word *var)

/* Transactional workflow */
boolean stm_prepare()
void stm_finalize()
void stm_retry()

/* Twilight operations */
void stm_reload()
void stm_ignore_updates()
boolean stm_inconsistent(tag t)
boolean stm_only_inconsistent(tag t)
```

consistent with the transaction's state. If it inconsistent, the transaction is aborted and restarted. If the memory location has been modified by the transaction, the method returns the corresponding entry in the write set.

Allocating and releasing memory is buffered in the same way as the write operations. They have to be issued via special operations, **stm_alloc** and **stm_free**, to allow easy reverse in case of a rollback.

3.2.3. Demarcating groups of variables

The read set of the transaction is unstructured in the sense that simply maps memory addresses to local copies. As conflict detection and resolution based on addresses can be rather tiresome, Twilight STM introduces tags to mark read set entries and group them according to their semantic meaning. For example, in linked data structures, a tag can be used for the node pointers that build the structure and another

one for the data entries stored in the nodes.

Tags are valid for the extent of one transaction. A new tag can be created with **stm_new_tag**. During the transaction's execution, locations are added to a tag with **stm_mark**.

In the twilight code, the programmer can use the tags to determine which kind of memory locations was marked as inconsistent and compensate for it as applicable.

3.2.4. Transactional workflow

Reminiscent of a two-phase commit in the database world, the Twilight STM splits the commit operation into operations **stm_prepare** and **stm_finalize** as indicated in Figure 3.1. First, **stm_prepare** obtains exclusive access to the variables in its transaction's write set. Then it attempts to validate the transaction by checking whether variables in the read set have been written to by other transactions. The return value of **stm_prepare** indicates if the transaction's read set is inconsistent because of other transactions' commits since the transaction's start. However, the decision on the outcome of the transaction is left to the twilight code that follows. Exclusive access to the variables in the write set is maintained to keep the transaction finalizable.

The twilight code can observe and modify the state of the transaction with *twilight operations*. Subsequently, the code can either restart the transaction (**stm_retry**) or it can fix the inconsistencies and finalize the transaction (**stm_finalize**). In the latter case, the values in the write set are published to shared memory. In either case, exclusive access to the variables in the write set is released.

3.2.5. Twilight operations

The read and write operations behave slightly different within the twilight zone:

- **stm_read** (`var`): Returns the value of variable `var` as it is recorded in the read set. It results in an error if the variable is still inconsistent or if the variable is not in the read set.
- **stm_write**(`var,val`): Updates the recorded value of variable `var` with value `val`. It results in an error if the variable is not in the write set.

Depending on the implementation of the Twilight API, these errors can either lead to a program failure or the programmer can deal with them by installing some exception handling in the application.

The following operations are unique to Twilight. They must not be used outside twilight code.

- **stm_reload**(): Atomically reloads a consistent snapshot of the read set. Afterwards, the transaction is ready to commit. Depending on the new values in the read set, the programmer may either want to update the write set or to restart the transaction.

- **stm_ignore_updates**`()`: Causes the transaction to ignore all updates performed by other transactions on variables that the transaction has read. It can be used to achieve snapshot isolation semantics (see Section 5.5.5).

  A call to either **stm_reload** or **stm_ignore_updates** is required for a successful commit if the twilight code is entered with an inconsistent read set. Otherwise, the transaction fails.

- **stm_inconsistent**`(t)`: Indicates whether the set of locations tagged with `t` contains at least one inconsistent variable.

- **stm_only_inconsistent**`(t)`: Indicates if the set of locations tagged with `t` contains at least one inconsistent variable, but no other tagged sets contain inconsistencies.

The operation **stm_only_inconsistent**`(t)` is usually used in combination with **stm_inconsistent**`(t)` to trigger repair code for regions which contain inconsistent variables.

3.3. Twilight STM in Action

Applications can benefit from twilight operations in many ways. We highlight its strong points with some examples of its use in a practical setting. A first concise example features transactional variables with high contention, avoids rollback via recalculation, and makes use of safe I/O actions in an atomic block to provide debugging output. We then show how the conflict detection may be employed to implement coarse grained transactions on collections whose operations semantically commute, but exhibit conflicts in the memory access patterns. Finally, we sketch how classical locks and network communication can be integrated in a safe way into Twilight transactions without causing the system to deadlock.

3.3.1. Debug traces

Listing 3.2 shows the code for a worker thread which executes concurrently with other threads on shared memory using STM for synchronization and information interchange. Suppose a programmer wants to trace the order in which transactions commit successfully for debugging purposes or for monitoring the application's progress. To this end, he introduces a global variable `counter` for assigning a unique identifier to each worker thread. To prevent the compiler from optimizations on this shared variable, he specifies it to be `volatile`. As every transaction reads and writes the counter, the counter is heavily contended and causes read inconsistencies in (almost) all concurrently running worker threads. As each such inconsistency aborts the respective transaction and restarts it, the transactional calculations are repeated fruitlessly and the possibility to execute the transactional parts of the program in parallel is lost.

Listing 3.2 Example: Debugging with `printf`.

```
volatile int counter;

void threadWorker() {
  stm_begin();                  // transaction body begins

/** complex computation with STM operations omitted   **/

  tag t = stm_new_tag();                // tag for counter
  int pos = stm_read(counter) + 1;      // update counter
  stm_write(counter, pos);
  stm_mark(t, counter);

  boolean succeeded = stm_prepare();    // twi zone starts

  if (!succeeded) {
    if (stm_only_inconsistent(t)) {
       stm_reload();                    // update read set
       pos = stm_read(counter) + 1;     // update counter
       stm_write(counter, pos);
    } else {
       stm_retry();
    }
  }
  printf("Txn %i finished at %i.\n", tid, pos);
  stm_finalize();                   // twilight code ends
}
```

Instead of using a transactionally managed variable for the counter, the programmer might contemplate on using other synchronization such as mutexes or placing the counter outside the transaction. However, primitives like mutexes do not mingle well with transactions, because the programmer is responsible for their correct use to prevent deadlocks and data races. This may not be possible because an STM implementation may perform transparent rollbacks anytime during a transaction.

The remaining alternative is to place the counter outside the atomic block such that it gets incremented just after the transaction has finished. However, the resulting information might be too imprecise due to the nondeterministic thread scheduling by the run time system.

In this situation, Twilight comes to the programmer's aid. Instead of aborting and restarting a transaction whose only inconsistency is caused by the counter variable, the programmer can contribute code to repair this inconsistency. To this end, it is possible to attach a tag to each variable that has been read in the atomic block. In the twilight code (i.e., the code after the **`stm_prepare`** operation), the query **`stm_only_inconsistent`**`(t)` determines if variables with tag `t` have been found to be inconsistent.

If the counter is the only cause for inconsistencies, then the code obtains a consistent view on the memory using **`stm_reload`**, recalculates the counter's value, and updates it before finally committing. To avoid deadlocks in the underlying implementation of transactions, the Twilight code can only read and write transactional variables through handles which are returned by read and write accesses in the body of the transaction. As the transaction is now known not to abort anymore, it is safe to output the logging message.

On the other hand, if the counter is found to be consistent, the twilight zone restarts the transaction in case there are inconsistencies involving the remaining variables. If the remaining variables are also consistent, then the transaction continues knowing that the transaction cannot fail anymore.

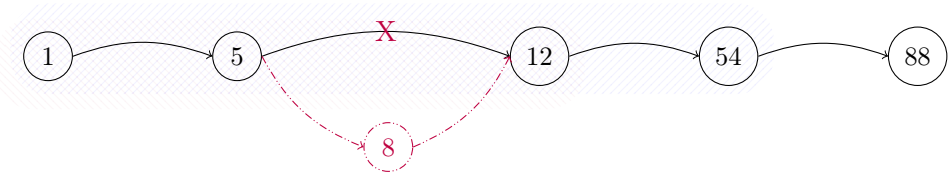
As the Twilight STM guarantees the atomic execution of the I/O actions in combination with the corresponding memory operations, it is particularly suited for debugging of STM applications. For example, we used it in our benchmark programs to determine the reasons for restart for memory locations with high contention.

3.3.2. Fine-grained conflict detection for data collections

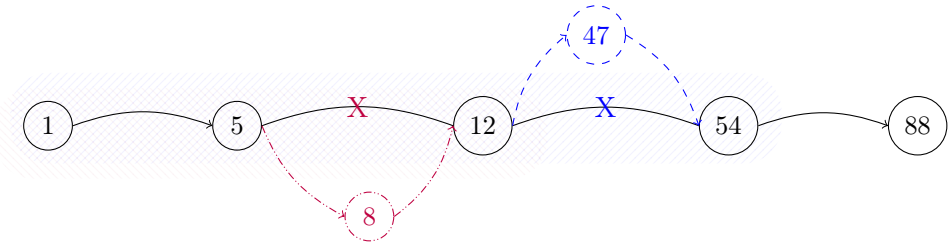
Typically, an STM algorithm implements a fixed isolation semantics such as opacity or snapshot isolation. This rigid choice limits a programmer's possibilities to implement application-specific conflict handling within the STM framework. Techniques like early release, weak atomicity, or open nesting [57] introduce some flexibility, but are difficult to handle correctly.

The algorithm underlying Twilight STM provides opacity [31] as the default semantics. In twilight code, however, the programmer can weaken the semantics by ignoring conflicts that are benign to the application. We illustrate the ideas with transactional operations on a singly-linked list data structure.

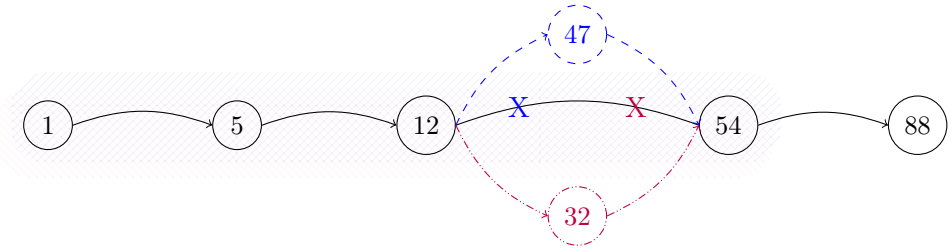
Figure 3.2. Example: Concurrent operations on lists.



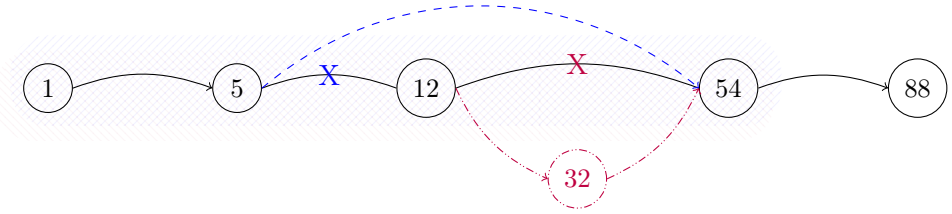
(a) Transaction 1 (red): Find 30. Transaction 2 (blue): Insert 8.



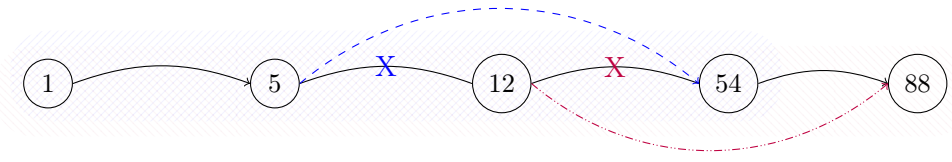
(b) Transaction 1 (red): Insert 8. Transaction 2 (blue): Insert 47.



(c) Transaction 1 (red): Insert 47. Transaction 2 (blue): Insert 32.



(d) Transaction 1 (red): Delete 12. Transaction 2 (blue): Insert 32.



(e) Transaction 1 (red): Delete 12. Transaction 2 (blue): Delete 54.

Figure 3.2 displays critical situations when two threads are concurrently operating on a singly-linked list in a transactional fashion.

In the first example in Figure 3.2(a), transaction 1 (highlighted in red) searches for a node with key 30 while transaction 2 (marked in blue) tries to insert a new node with key 8. If transaction 2 commits before transaction 1, it invalidates the other transaction's read set causing it to restart and again iterate through the list. Yet, the insertion operation is semantically not relevant for the lookup operation. It is actually safe for transaction 1 to ignore all updates that are performed during its own execution while having a linearizable implementation for singly-linked lists:

- If the new node with a smaller key is inserted, the re-execution of the lookup yields the same result. The same holds if the key is larger than the one in the lookup transaction. The execution order of the transactions is interchangeable.
- If the new node is inserted at the critical point (in the example, between node 12 and 54), and has the sought-after key, a re-execution gives a different result. Though, linearizing the lookup transaction before the insertion yields the same result as when ignoring the update.
- Similarly, when a concurrently running transaction is deleting a node, the lookup operation is either not affected or can be linearized before the deletion.

The linearization point of the read-only lookup transaction is given when it successfully tests the key of a node to be equal or greater than the key it tries to locate in the list. This point is independent of concurrently running update operations such that these operations' linearization point can be ordered both before and after.

For transactions that modify the list structure, the linearization point is given when they gain exclusive access to the elements in their write set. Though insertions or deletion at different points in the list are semantically non-conflicting (Figure 3.2(b)), it is unsafe for these transactions to ignore all updates. In Figure 3.2(c), two transactions try to insert an new node at the same position. If the transaction that obtains the exclusive write access to the `next` pointer of node 12 secondarily ignored the update to this pointer by its predecessor, the node that was inserted first would be removed from the list.

One possible way to prevent this situation while still maximizing the commit rate of the STM system is to revert to snapshot isolation semantics for transaction [8]. It specifies that if two transaction try to update the same memory location, only one can perform this write operation successfully while the other has to abort and restart.

Snapshot isolation allows concurrent updates to the list structures as long as they happen at different places in the list. Consider however the example depicted in Figure 3.2(d). The write sets of transaction 1 and 2 do not overlap as one is modifying the `next` pointer of node 5, the other the `next` pointer of node 12. Ignoring the removal of node 12 causes transaction 2 to actually insert node 32 at a place where it is not reachable when iterating through the list. To prevent this

situation, the removal of a node from the list needs to modify both the pointer to the node and the pointer of the node to its successor (in the example, the next pointer of node 12). The write set of transaction 1 then overlaps with the one of transaction 2, causing the transaction which tries to commit later to detect a conflict and restart.

Listings 3.3 and 3.4 show parts of the implementation of such a concurrent sorted singly-linked list.

A list is a pointer to the first node in the list. A node consists of a key which is needed to sort the entries and a pointer to its successor. The lookup procedure in Listing 3.3 iterates through the list inspecting the keys of the list elements until it finds the node whose key is greater or equal to the parameter `n`. After entering the twilight zone (l.8), it instructs the STM framework to disregard all conflicts. Then it finishes the transaction and returns the result.

As the resulting lookup procedure is read-only and can safely ignore intermediate updates, it might even be implemented without STM operations, but with standard memory accesses. However, this requires the programmer to ensure carefully that list traversal is possible even when concurrent operations restructure the list by inserting or deleting elements.

When inserting a new key in the list (Listing 3.4), both the pointer to the new node and the pointer to its predecessor are marked for the inconsistency check (l.15). If they are found to be inconsistent, the transaction has to restart. Otherwise, it finishes its execution successfully. The deletion procedure is done analogously.

The tagging mechanism can also be facilitated to distinguish between modifications on the list structure and updates of data stored in a node. It is further possible to augment the list data structure with a field containing the current size of the list. As for the commit counter in Listing 3.2, it is necessary to provide repair actions of increment and decrement in the insert and deletion method's twilight zone to avoid the serialization of these modification operations.

3.3.3. External locking protocols

Although contrary to the spirit of STM, it is sometimes advantageous to provide some means of reverting to classical mutual exclusion mechanisms. For example, an optimistic execution of I/O operations on files requires complex buffering protocols and the enclosing of the file handle into a transactionally managed wrapper. With Twilight STM it is possible to include lock-based protocols into the twilight zone. The progress of the transactional system then requires the correct handling of the mutexes as otherwise transactions might get stuck in their twilight zone. Further, when integrating these locking protocols into Twilight STM, the programmer must keep in mind that transactions that currently execute their twilight zone have disjoint write sets. The next example shows how to combine Twilight STM and file access in a dead-lock free manner.

Consider a banking application where transactions are running concurrently to transfer money from one client's account to some other accounts. A transaction may contain multiple transfers and multiple transactions may be running simultaneously

Listing 3.3 Example: Data structures and lookup method for singly-linked list.

```
struct node {volatile int key, volatile node *next};
struct list {volatile node *head};

node* newNode(int key, node* next) {
  node* n = stm_alloc(sizeof(node));
  n->key = key;
  n->next = next;
  return n;
}

boolean lookup(volatile list *l,int n) {
  stm_begin();
  node *head = stm_read(l->head);
  boolean result = _lookup(head,n);
  stm_prepare();
  stm_ignore_updates();
  stm_finalize();
  return result;
}

boolean _lookup(node *head, int n) {
  node* current = head;
  while (current != null) {
    int key = stm_read(current->key);
    if (key == n) {
      return true;
    } else {
      if (key > n) {
        return false;
      } else {
        current = stm_read(current->next);
      }
    }
  }
  return false;
}
```

Listing 3.4 Example: Insertion for singly-linked list.

```
void insert (volatile list* l, int n) {
  stm_begin();
  tag t = stm_new_tag();
  node *head = stm_read(l->head);
  _insert(&l->head,head,n,t);
  if (!stm_prepare()) {
    if (stm_inconsistent(t)) stm_retry();
    else stm_ignore_updates();
  }
  stm_finalize();
}
void _insert (node **tohead, node *head, int n, tag t) {
  if (head == null || stm_read(head->key) > n) {
    head = newNode(n,head);
    stm_write(l->head,head);
    stm_mark(t,l->head);
    return;
  }
  if (stm_read(head->key) < n) {
    node **toprev = tohead;
    node *prev = head;
    node *next = stm_read(prev->next);
    while (next != null) {
      int key = stm_read(next->key);
      if (key == n) return;
      if (key > n) {
        node *new = newNode(n,next);
        stm_write(prev->next,new);
        stm_mark(t,prev->next);
        stm_mark(t,*toprev);
        return;
      }
      toprev = &(prev->next);
      prev = next;
      next = stm_read(prev->next);
    }
    node *new = newNode(n,null);
    stm_write(prev->next,new);
    stm_mark(t,prev->next);
    stm_mark(t,*toprev);
  }
}
```

on behalf of a single client. For performance reasons, all transactions operate on a fast in-memory model. For durability, a summary of the outcome of each transaction is also logged to permanent storage using one dedicated file per client. After a crash, the system should be able to reconstruct its state from these files.

Additionally, the bank runs a background task that regularly compares the state on disk with the state in memory to detect data corruption caused by faulty hardware or programming errors. As the transfer information in memory contains the file pointer to its summary on disk, validation is possible in both directions. The background task concurrently picks a client `c`, reads `c`'s data from memory, scans `c`'s file, and checks for mutual consistency. The major challenge in this scenario is that reading the memory and the file must form one atomic action, while other transactions can still execute concurrently and without deadlocking.

This non-trivial problem has a fairly straightforward solution in Twilight STM. Listing 3.5 contains the code for the transaction that performs some money transfers `transactMoney` as well as the transaction with performs the background verification process in `transactVerify`.

After transferring the money by reading and writing to the memory, the transfer transaction writes a summary to the file, obtaining location information that is also written to the in-memory model. There is one subtlety involved in this code. Storing the location information must be prepared with a dummy location in the transaction body because the twilight code is only allowed to write to variables that are already in the write set when entering the twilight zone. This dummy value is never visible outside the transaction.

The code for the background verification task follows a similar pattern. The file locks taken in the twilight zone ensure that the entire transaction forms an atomic unit together with the file access. Hence, the client's data in memory matches the contents of the file. This code thus fulfills the application's requirement.

3.3.4. Applications in a distributed setting

Distributed systems, like multi-player online games, require a multitude of different concurrent tasks. To ensure scalability, every participating execution node is responsible for modeling a fragment of the world consisting of a number of entities (e.g., monsters) that act autonomously in the virtual world. Furthermore, every node drives a GUI, processes commands entered by the player, and it receives status update messages from neighboring nodes.

Sweeney [72] suggests that each of these tasks can be assigned to its own thread that modifies the internal state of the node, which is shared between the threads. However, each thread also sends information about the state of its monster to the neighboring nodes. A mutual exclusion protocol with semaphores guarantees safe multiplexing of different TCP connections. The communication should not happen outside the atomic region to prevent the transmission of inconsistent state information.

In this scenario it is essential that multiple threads can execute the communication

Listing 3.5 Example: External synchronization.

```
void transactMoney(client c, t_info[] transfers) {
   stm_begin();

   // change in-memory model
   for (t_info t: transfers) {
      transfer(c, t.recipient, t.amount);
      add_to_summary(t.recipient, t.amount);
   }
   store_location_info (c, 0); // dummy value
   if (stm_prepare()) {
      // obtain a lock on the client's file
      acquire(file_lck);
      int loc = write_summary_to_file(c);
      release(file_lck);

      // update in-memory model
      store_location_info(c, loc);
      // complete the transaction
      stm_finalize();}
   else
      stm_retry();
}

void transactVerify(client c) {
   stm_begin();
   read_client_data(c);

   boolean consistent =  stm_prepare();

   if (!consistent) {
      stm_retry();
   }

   acquire(file_lck);
   assert_consistent_with_file(c);
   release(file_lck);

   stm_finalize();
}
```

phase in the twilight code at the same time, as a game model may contain 10,000 active game-play objects and maintain connections to 10-20 neighbors. In the style of the lock-based protocol in Section 3.3.3, the Twilight STM may be employed to obtain a high degree of concurrency.

A similar case study has been performed by Zyulkyaro and co-workers [81]. They restructured a parallel version of the Quake game server to use transactions. This application features I/O operations and system call invocations, as well as data that is accessed transactionally and non-transactionally. In refactoring such complex applications to transactional style, features of Twilight STM can be very helpful.

3.4. Limitations of Twilight STM

Twilight STM extends the scope for STM by providing a rich interface which allows for transaction-specific conflict handling and integration of non-revertible actions. Expert programmers benefit from these advantages in particular as the strict enforcements of standard STM systems impede application-specific solutions.

While Twilight STM aims at weakening some of these restrictions, programmers still have to comply with some limitations.

3.4.1. Nesting of transactions

Twilight STM has some short-comings when it comes to composing transactions. Transactional bodies can be flatly nested while not requiring any changes to the semantics. However, it is not clear what the semantics of twilight code that is nested in an outer transaction should be.

One option might be to collect the twilight codes of all nested transactions and execute them when the top level transaction commits. This poses however the question in what order the twilight zones are executed and hence when their side-effects become visible. Commit hooks apply a similar strategy: While evaluating the atomic block, a transaction can define code to be executed after a successful commit or an abort. The order of execution is usually fixed by the order in which they are collected.

Twilight code cannot be easily mapped to abort and commit hooks as it features operations that can turn an aborting transaction into a successfully committing one and vice versa. If one sub-transaction decides that the update to some variable is benign to the semantics of the program, should the program still restart if one of the sibling transactions declares a different strategy? If a nested transaction defines some repair code for fixing an inconsistency, how do the parent and sibling transactions adapt to these changes?

Open nesting [57] is a more natural choice for nested twilight transactions. Conflict resolution is then again defined on a per-transaction base. Side-effecting operations are executed as they appear in the program order. Yet, open nesting is introducing complications, such as the inversion of order between parent and child transactions, or breaking the transactional isolation when a child transaction leaks

values before the parent transaction issues their publication. Further, it would violate the guaranteed irreversibility of twilight zones. It would no longer be possible to provide the inclusion of irrevocable I/O as the restart of an outer transactions triggers the twilight code of the nested transaction again, even if the child transaction committed successfully. The design space for conflict resolution and detection is therefore growing more complex.

To simplify the design and allow easy reasoning about interacting twilight zones, Twilight STM requires that twilight code is only specified for top level atomic blocks. In programming languages with a rich static type system, like Haskell, this requirement can be statically enforced (Section 6.3). In addition, the twilight code for an atomic block can be wrapped into a function which can then be called in the top level twilight zone. It is then the programmer's responsibility to amalgamate the different repair operations or irrevocable actions in a safe manner.

3.4.2. Reading and writing in the twilight zone

The commit protocol of Twilight STM is an adaptation of the two-phase commit protocol. While in the twilight zone, the transaction is required to have exclusive access to the memory locations that it wants to modify. If the programmer would be allowed to extend the transaction's write set in the twilight zone, issuing writes to variables that have not been written to in the transaction's body, the STM algorithm would need to acquire exclusive access to further memory locations. This leads quickly to deadlock situations in the system if acquisition of locks gives rise to a cycle dependency on transactions that try to obtain a lock.

In a similar fashion, extending the read set by read accessing supplementary memory locations leads to complications in obtaining a consistent memory snapshot. For example, a transaction that has passed the consistency check in **`stm_prepare`** can invalidate its snapshot with reading another variable in the twilight zone.

Twilight STM therefore restricts reads and writes in the twilight zone to variables that have already been accessed with a read or write operation, respectively, in the transactional body. As with nesting of transactions, it is possible to statically enforce this obligation with Haskell's type system. In other programming languages, an invalid access is detected at runtime and results in an exception.

An alternative design for extending the read and write set can be obtained by trying to perform the operation and indicating a possible failure in the return result. However, this option puts again the burden on the programmer to define some appropriate reaction as answer to a failure and complicates the overall program design.

3.4.3. (Trans)Actions in the twilight zone

Although the twilight zone admits the execution of I/O operations in a safe manner, the programmer should make sure that a transaction is not stuck indefinitely in the twilight zone waiting for some external resource becoming available. As outlined in

Section 3.3.3, the design of protocols for mutual exclusion has to take into account that transactions that are executing their twilight zones currently have disjoint write sets.

Additionally, Twilight STM precludes that a transaction is nested into the twilight zone of another transaction. As with extending the read or write set, the execution of such a nested transaction could give rise to deadlocks and inconsistencies.

As with other concurrency paradigms that provide exclusive access to resources during critical operations like updates, it is good practice to release the resources as quickly as possible. Applying this guideline to Twilight STM, programmers will improve their application's throughput if the operations in the Twilight zone are of short duration. Also, they need to carefully balance out the cost of restarting a transaction and of repairing some inconsistency.

Chapter 4.

Algorithm

The STM algorithm underlying the Twilight STM is a time-stamp based algorithm. It resembles the classic TL2 algorithm [20], but differs in some vital parts.

Figures 4.1-4.5 contain the basic TwilightSTM algorithm in pseudo code notation. In this notation, *var* denotes a transactional variable or transactionally shared memory location. lock(*var*) denotes the lock associated to the transactional variable. A lock can either be Free, Reserved, or Locked. The store_lck(*var,val*) changes the current lock value to *val*. CAS(*l, A, B*) denotes a compare-and-store operation, that is, the store of value B at l is only performed if the current value of l is A.

The operation $(val, t, lck) \leftarrow \text{load}(var)$ operation atomically loads the data at *var*, the associated time stamp, and the current state of the associated lock. Similarly, store(*var, val, t, lck*) atomically stores the data in the shared heap. The store and load operations modify shared memory locations and require memory barriers to make changes visible to other threads.

For more details on how to actually implement the algorithm in different languages, we refer to Section 6.

4.1. Globally shared state

Timer To provide consistent memory snapshots to each transaction, the STM algorithm relies on a global timer `T`. Each transactional variable (i.e. a heap location that is administered by the STM system) is associated with a version number denoting the time of its last modification.

The timer is read upon each start of a transaction, and when reloading its memory snapshot. It is incremented when a transaction successfully commits changes to the shared heap.

Locks Each transactional variable is associated with a flag which denotes the variable's current state. It is used as a lock or monitor and is in one of the following states:

- A *locked* variable is currently modified by a transaction and not accessible by any other transaction.
- A *reserved* variable is in the write set of a transaction currently in the twilight zone. It may still be read safely by other transactions.

- A *free* variable is neither reserved nor locked.

The flags can be combined with the timer to reduce the memory overhead depending on the memory architecture of the processor. To improve the cache locality, it can be helpful to store the timer/lock-combinations not with each heap location, but to administer them in one hash table. In this case, care must be taken that the meta data is shared only between few transactional variables, as otherwise transactions will have to abort due to false conflicts.

To prevent deadlocks in the algorithm, Twilight STM reverts to the well-known technique of obtaining locks in an ordered fashion. The lock order can be obtained in many different ways, e.g., by explicitly enumerating them, or using their memory address as total order.

4.2. Transaction local state and operations

A transaction's meta data is responsible for tracking the read and write accesses dynamically such that conflicts with other transactions can be discovered efficiently. To this end, each transaction administers the following structures:

- A time stamp t_{init} taken at the begin of the transaction is needed to create a consistent view of the memory.
- A read set stores for each transactional variable that is read by the transaction the value and its modification time stamp.
- A write set contains the modified value for each transactional variable that is modified by the transaction.
- A tag map stores for each tag the associated transactional variables.
- A state flag indicates whether the transaction is consistent with respect to the point in time when the transaction entered the twilight zone. A transaction state is either *notchecked*, *consistent*, or *inconsistent*.

When starting a transaction with StmBegin (cf. Algorithm 4.1), it is initialized with empty mappings for the read set, write set, and the tag map. Its state is initially set as *notchecked*. Further, the current value of the global timer is read to initialize the t_{init} time stamp.

4.2.1. Reading and writing transactional memory

The code for the read and write operations are given in Algorithm 4.1. When reading a transactional variable, first the thread-local data structures, namely the write and read set, are checked if they contain local copies of the variable. Otherwise, it must be the first transactional access of the variable in the transaction. The system then creates an entry in the read set comprising the variable, its value, and its version

Algorithm 4.1 Twilight STM: Initializing a transaction, reading and writing transactional variables.

```
method StmBegin()
    readset ← ∅
    writeset ← ∅
    tags ← ∅
    state ← notchecked
    t_init ← read T
end

method StmRead(var)
    if writeset.contains(var) then
        val ← writeset.lookup(var)
    else if readset.contains(var) then
        (val, t_var) ← readset.lookup(var)
    else
        (val, t_var, l) ← load(var)
        if l = Locked || t_var > t_init then
            StmRetry( )
        end if
        readset.add(var, (val, t_var))
        return val
    end if
end

method StmWrite(var, val)
    writeset.add(var, val)
end
```

Algorithm 4.2 Twilight STM: Entering and exiting the Twilight zone.

```
method StmPrepare()
    reserve(writeset)
    return validate(readset)
end

method StmFinalize()
    if state ≠ consistent then
        StmRetry( )
    end if
    lock(writeset)
    publishAndUnlock(writeset)
end

method StmRestart()
    if state ≠ notchecked then
        unlock(writeset)
    end if
end

method StmCommit()
    StmPrepare( )
    StmFinalize( )
end
```

number at the time of the read operation. To prevent zombie transactions that run on an inconsistent view of the global heap, the STM system restarts a transaction if it tries to read from a variable with a time stamp later than t_{init}. To guarantee the correctness of the global read, values have to be read atomically together with their time stamp and lock flag.

Write operations to shared variables are performed lazily. They are recorded locally in the transaction's write set.

4.2.2. Committing a transaction

When the transaction attempts to commit, it invokes the method StmPrepare (cf. Algorithm 4.2). To prevent dead locks when several transactions acquire their reservation concurrently, the access to the locking flags has to be performed in a globally consistent order. After acquiring the reservation for the variables in the write set, the transaction validates the read set by checking that the variables in the read set are currently not locked and their current version numbers are still less than or equal to t_{init}.

In the end, the code either restarts the transaction with StmRetry or tries to

Algorithm 4.3 Twilight STM: Internal operations.

```
method LOCK(writeset)
    sortedlist ← sorted list of all entries in write set
    for (var, val) ∈ sortedlist do
        store_lck(var, Locked)
    end for
end

method RESERVE(writeset)
    for (var, val) ∈ writeset do
        CAS(lock(var), Free, Reserved)
    end for
end

method VALIDATE(readset)
    state ← consistent
    for (var, t_var) ∈ readset do
        (val, t'_var, l) ← load(var)
        if l = Locked || t_var ≠ t'_var then
            state ← inconsistent
        end if
    end for
end

method PUBLISHANDUNLOCK(writeset)
    t_commit ← sample T
    for (var, val) ∈ writeset do
        store(var, val, t_commit, Free)
    end for
end

method UNLOCK(writeset)
    for (var, val) ∈ writeset do
        store_lck(var, Free)
    end for
end
```

commit by calling StmFinalize. The latter operation also restarts if the read set is inconsistent. Otherwise, it publishes the write set by writing the new values to the shared variables and setting their version numbers to the current time t_{commit}. In any case, the transaction releases the exclusive access to the shared variables.

For compatibility with the standard STM programming interface, the Twilight API also includes a commit operation StmCommit which calls StmPrepare and StmFinalize. In case of conflicts, StmFinalize issues a restart. Otherwise, the transaction can commit its changes to the global heap (cf. Section 5.5).

4.2.3. Repair operations

After calling StmPrepare, the transaction enters its twilight code. If the twilight code wants to correct inconsistent reads, it first has to obtain a consistent read set with StmReload. The operation StmReload updates the read set atomically if the current read set is inconsistent. The operations StmReread and StmUpdate are equivalent to their regular pendants, StmRead and StmWrite, but do not allow extending the read and write set with new entries, respectively.

4.2.4. Tracking down inconsistencies

Querying transactional variables individually for inconsistencies is cumbersome. To simplify the consistency test, the Twilight API offers a tagging facility to combine variables to groups. A thread-local counter is used to generate unique tag values in StmNewTag. With StmAddTag the programmer can mark variables to belong to one group. A variable may belong to several tagging groups.

In the twilight zone, the method StmIsInconsistent checks whether the variables that were marked with a tag are inconsistent with respect to the global heap at the time when entering the twilight zone.

4.3. Properties of the algorithm

The Twilight API operations rely on a locking protocol that maintains the following invariants:

Lemma 4.3.1 (Correctness of the Twilight locking protocol)**.** *The access control in the Twilight algorithm guarantees the following invariants:*

1. *A transaction only reads variables that are not currently modified by other transactions.*
2. *A transaction has exclusive write access to the variables in its write set from* StmPrepare *till* StmFinalize*.*

Proof of 4.3.1: When reading a transactional variable in StmRead, an atomic load of both the value and its time stamp and lock flag is performed. The read

Algorithm 4.4 Twilight STM: Repair operations.

```
method StmUpdate(var, val)
    if writeset.contains(var) then
        writeset.add(var, val)
    else
        throw error
    end if
end

method StmReread(var)
    if writeset.contains(var) then
        val ← writeset.lookup(var)
    else if readset.contains(var) then
        (val, t_var) ← readset.lookup(var)
    else
        throw error
    end if
    return val
end

method StmReload()
    if state = consistent then return
    else
        snap ← false
        while !snap do
            t_reload ← read T
            readset' ← ∅
            snap ← true
            for (var, (val, t_var)) ∈ readset do
                (val, t_curr, l) ← load(var)
                readset'.put(var, (val, t_curr))
                snap ← t_curr < t_reload && l ≠ Locked
                if !snap then
                    break
                end if
            end for
        end while
        state ← consistent
        readset ← readset'
    end if
end
```

Algorithm 4.5 Twilight STM: Handling of tags.

```
method StmNewTag()
    tag ← generate new tag
    return tag
end

method StmAddTag(tag,var)
    tags.add(tag, var)
end

method StmIsInconsistent(tag)
    tagged ← tags.get(tag)
    for var ∈ tagged do
        (val, t_var) ← readset.get(var)
        if (t_var ⩾ t_init) then
            return true
        end if
    end for
    return false
end
```

operation is successful only if lock flag is *Free* or *Reserved*, i.e. no other transaction is modifying the variable's global state.

Similarly, the load of the variable in StmReload checks the flag status to be different from *Locked*.

Assume there are two transactions that both have a variable v that in their write sets. When entering the Twilight code in StmPrepare, only one transaction can reserve v by a successful CAS on the associated lock from changing v's state from *Free* to *Reserved*. □

From the previous lemma, we can deduce the correctness of the system with respect to memory consistency.

Definition 4.3.1 (Consistent memory snapshot)**.** *A transaction operates on a consistent memory snapshot if there exists a point in time in the execution such that for each variables in the read set, the heap either contains the value and timestamp as registered in the read set or a transaction has exclusive access to variable and is about to set this value and timestamp.*

Depending on the execution of a transaction, the point in time is given after the time stamp of either t_{init} or t_{reload} has been set. The first case in Definition 4.3.1 holds when the transaction which updated the variable in the read set has finalized its commit. The second case is given when the transaction still processes its commit, i.e., after increasing the timer T but before updating the value and releasing the lock

on the variable. A transaction can have such a value in its read set if it obtained the time stamp of t_{init} or t_{reload} while the transaction incrementing the counter is still processing its commit, but performs the read access later, after the transaction has released the lock.

Lemma 4.3.2 (Consistency)**.** *A transaction always operates on a consistent memory snapshot.*

Proof of 4.3.2: During the evaluation of the transactional body, all read operations on the shared heap check that a variable's time stamp that is entered into the read set has not been changed since the transaction has started and obtained t_{init}. In a similar way, the StmReload method either keeps the read set as is, or it obtains a memory snapshot of all read set entries that is consistent with respect to t_{reload}.

The update of the global memory in StmFinalize is visible to the outside atomically. Only after locking all variables, the global timer is atomically incremented. Transactions that obtain their t_{init} time stamp before this increment reject the updated values in StmRead or StmReload as they either fail or restart the snapshot creation. All transactions that obtain the same or a later time stamp cannot read the previous values because either the lock flag of the variables is still set to *Locked*, or they have already been overwritten with the newly committed values. □

Lemma 4.3.3 (Progress)**.** *All Twilight API operations are obstruction-free if the execution of user-defined twilight code for any transaction always terminates.*

Proof of 4.3.3: The only blocking operation in the Twilight API is the reservation of the transactional variables in the write set when entering the twilight zone. The corresponding CAS operation can only fail if the lock is not in the *Free* state, which means that some other transaction is currently in the critical twilight section. Under the assumption that execution of the twilight code always terminates, this transaction does eventually leave the twilight zone.

When no transaction is in its Twilight zone, the CAS operation succeeds, and thus the waiting thread can make progress. □

The Twilight STM is not lock-free: Transactions which have distinct write sets but read variables that are in another transactions write set cause each other to restart because the validation of the read set fails when they try to commit at the same time. Achieving lock-freedom can be achieved with the help of a contention manager which arbitrates between conflicting transactions in a fair way.

Chapter 5.

Correctness

By avoiding observable inconsistencies, the semantics of transactional memory provides a comparatively simple model for concurrent programming. Instead of (implicitly) associating several memory locations with a lock and requiring that the lock needs to be obtained before accessing any of these memory locations and released thereafter, accesses are grouped together in a transaction that runs at a proclaimed level of isolation.

Prior work on the semantics of transactions [1, 54] focusses primarily on weak atomicity and studies the interaction of transactional and non-transactional memory accesses. This is an important aspect for applications combining different kinds of synchronization. For example, an application that builds on legacy code might contain locking as well as transactional code. However, these formalizations do not account for the phenomena that occur in an interleaved execution of transactions. For example, in state-of-the-art algorithms like TL2 [20], threads may get stuck even when a fair scheduling of threads is provided because they are repeatedly forced to abort by other transactions' successful commits.

To illustrate the mechanism underlying the aborts, this chapter pursues an approach that abstracts program execution by traces of memory accesses and transaction control operations. To this end, we define a monadic lambda calculus with threads and transactions, Λ_{STM}. Similar to schemes in research on isolation levels for databases, each memory access is modeled by an effect on the global heap. This abstraction allows for an easy comparison of different TM algorithms. Their semantics are reflected in the effect traces which they generate during program execution under some scheduling schemes. The structure of the traces is determined by the isolation levels of the respective STM algorithm.

This chapter is ordered as follows:

1. We present a formalization of a semantics for transactional memory that is suitable for proving properties of a TM implementation.

   A high-level semantics abstracts from so many details that properties of the implementation become trivially evident [35]. A low-level semantics provides so many details that formal proofs of its properties are no longer tractable. Our semantics keeps the middle ground. It explicitly models the non-deterministic interleaving of the operations in each thread including operations in aborting transactions. However, it does not model implementation details like the construction of memory snapshots or the implementation of locks.

2. We prove that our semantics for Λ_{STM} implements opacity [31], that is, all execution traces in our semantics are equivalent to serial execution traces, where the execution of critical regions, namely the transaction bodies, is non-interleaved.

3. We demonstrate that a small modification of the semantics (the TM algorithm, respectively) yields another notion of transactional isolation, namely snapshot isolation. We define a criterion for traces obtained under snapshot isolation and prove that the modified semantics Λ_{SI} only produces such snapshot traces.

4. We extend the formal calculus to the work flow and repair operations of Twilight STM, Λ_{TWI}. We show that without applying any Twilight operations, the transactions in Λ_{TWI} implement opacity. By applying different strategies for conflict resolution in the twilight zones, the resulting protocols are shown to yield snapshot isolation or irrevocability.

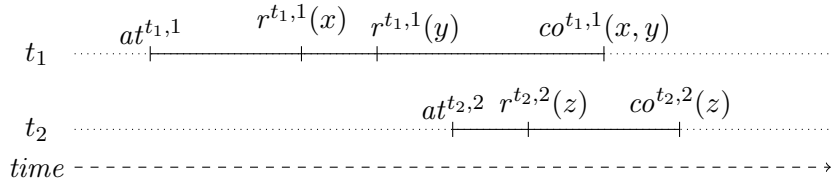
5.1. Execution traces

Let us begin with some informal examples of execution traces that provide insight into our approach. Execution traces are sequences of atomic effects that denote the beginning of a transaction ($at^{t,i}$), read accesses to memory location l within a transaction ($r^{t,i}(l)$), commits of a transaction which correspond to globally visible modifications to shared locations $\bar{l}$ ($co^{t,i}(\bar{l})$), and abort effects for unsuccessful transactions ($ab^{t,i}$). In these effects, t identifies the thread and i is a unique transaction identifier. A thread may run multiple transactions over time, but only one at a time. In most of our examples each transaction runs on a distinct thread.

To simplify reasoning, we rely on an abstract notion of time. Each effect is supposed to happen atomically at a distinct, single point in time. Further, effects can be totally ordered according to the point in time when they occur.

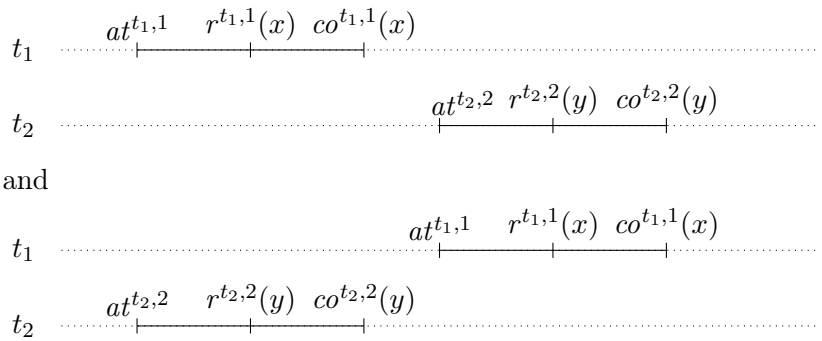
5.1.1. Successful commits

As a first example, consider the following trace:



The scheduling interleaves two transactions t_1 and t_2 that read and write disjoint locations $x \neq y \neq z$. For better readability, we separate the effects for each thread such that each dotted line shows the restriction of the trace to one thread. A full line indicates the duration of a transaction.

For this trace, there are two equivalent serial traces:



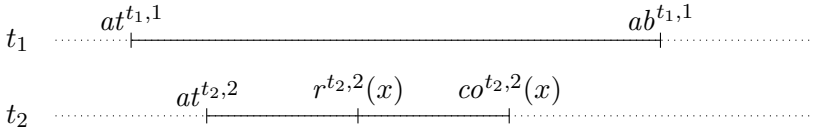
and

We consider traces to be equivalent if they correspond to evaluations of the program to the same final heap and result. In such a serial trace, each transaction conceptually operates on its own snapshot of the heap, taken at the beginning of the transaction. As both transactions were able to finish successfully, their read and write sets cannot have elements in common, and all their operations are independent. Hence both serial traces are equivalent to the original one.

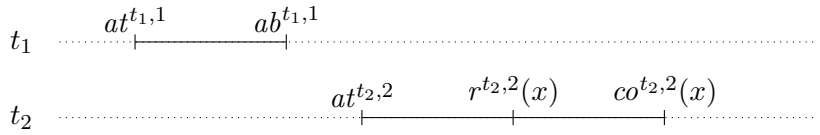
5.1.2. Read conflicts

A read conflict occurs if one transaction commits a write operation to a variable that another transaction is just about to read. In that case, the reading transaction must not proceed because its snapshot is no longer consistent with the current heap. Thus, the semantics forces the second transaction to abort.

An example of abort induced by a read conflict is depicted in the next trace.



The trace gives an example for a transaction t_1 which had to abort because of a read conflict on x (it does not produce a read effect because the read operation is never permitted, as explained above). Nevertheless, there is an equivalent serial trace:

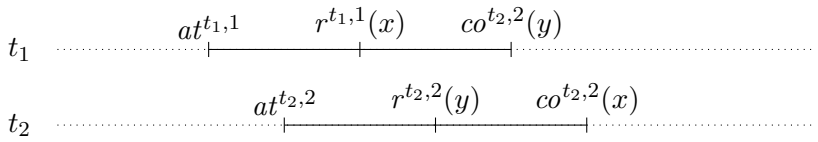


5.1.3. Snapshot isolation

Not all interleavings of threads can be decoupled into some equivalent serial trace. Take a look at the next example:

Figure 5.1. Syntax of Λ_{STM}. Gray expressions arise only during evaluation.

$$
\begin{array}{lll}
x \in \mathrm{Var} & & \\
l \in \mathrm{Ref} & & \\
v \in \mathrm{Val} & ::= & l \mid \texttt{tt} \mid \texttt{ff} \mid () \mid \lambda x.e \mid \texttt{return}\, e \\
e \in \mathrm{Exp} & ::= & v \mid x \mid e\ e \mid \texttt{if}\ e\ e\ e \mid e \gg\!= e \\
 & & \mid \texttt{spawn}\, e \mid \texttt{atomic}\, e \mid (e, W, R, i, e, \mathcal{H}) \\
 & & \mid \texttt{new}\, e \mid \texttt{read}\, e \mid \texttt{write}\, e\, e
\end{array}
$$



The trace is not serializable because a read operation is supposed to return the last value written to a location. Hence, in a serial trace the latter read operation would yield the value written and committed by the first transaction.

Algorithms that admit traces like in the last example implement a weaker isolation level called snapshot isolation. Semantically, a thread-local copy of the memory is made at the start of a transaction. The transaction operates on this private copy during its execution. At commit, all changes are merged back into the global heap. The transaction is in conflict with another transactions only if both transactions try to update the same heap locations. A detailed discussion of snapshot isolation is done in Section 5.4.

5.2. Formalizing STM

This section formalizes an STM with lazy update, where all write operations are delayed till the commit operation. The formalization, Λ_{STM}, is based on a monadic call-by-name lambda calculus with references, threads, and transactions.

5.2.1. Syntax of Λ_{STM}

Figure 5.1 contains the syntax of Λ_{STM}. A value is either a reference, a boolean, the unit constant, or a function. Expressions comprise these values, variables, function application, conditional, monadic return and bind, spawning of threads, transactions, transactions in progress (an intermediate expression not arising in source programs), and the usual operations on references.

Figure 5.2 defines the type system for Λ_{STM}. The type language consists of the types of the simply typed lambda calculus with base types boolean and unit, a reference type $\texttt{R}\,\tau$ for references pointing to values of type τ, function types, and a monadic type $\mu\,\tau$ for a monad returning values of type τ. There is a choice of two monads, `IO` for general monadic operations and `STM` for operations inside a transaction.

Figure 5.2. Typing rules of Λ_{STM}.

$$
\begin{array}{lllll}
\text{Types} & \tau & ::= & \texttt{bool} \mid () \mid \texttt{R}\,\tau \mid \tau \to \tau \mid \mu\,\tau \\
 & \mu & ::= & \texttt{IO} \mid \texttt{STM} \\
\text{Type environments} & \Gamma & ::= & \emptyset \mid \Gamma, x : \tau \\
 & \Sigma & ::= & \emptyset \mid \Sigma, l : \tau
\end{array}
$$

$$\frac{}{\Sigma|\Gamma \vdash \texttt{ff} \,:\, \texttt{bool}}\ \text{T-False} \qquad \frac{}{\Sigma|\Gamma \vdash \texttt{tt} \,:\, \texttt{bool}}\ \text{T-True} \qquad \frac{}{\Sigma|\Gamma \vdash () \,:\, ()}\ \text{T-Unit}$$

$$\frac{\Gamma(x) = \tau}{\Sigma|\Gamma \vdash x \,:\, \tau}\ \text{T-Var} \qquad \frac{\Sigma(l) = \tau}{\Sigma|\Gamma \vdash l \,:\, \texttt{R}\,\tau}\ \text{T-Ref}$$

$$\frac{\Sigma|\Gamma, x : \tau_1 \vdash e \,:\, \tau_2}{\Sigma|\Gamma \vdash \lambda x.e \,:\, \tau_1 \to \tau_2}\ \text{T-Func} \qquad \frac{\Sigma|\Gamma \vdash e_2 \,:\, \tau_1 \to \tau_2 \qquad \Sigma|\Gamma \vdash e_1 \,:\, \tau_1}{\Sigma|\Gamma \vdash e_2\ e_1 \,:\, \tau_2}\ \text{T-App}$$

$$\frac{\Sigma|\Gamma \vdash e_1 \,:\, \texttt{bool} \qquad \Sigma|\Gamma \vdash e_2 \,:\, \tau \qquad \Sigma|\Gamma \vdash e_3 \,:\, \tau}{\Sigma|\Gamma \vdash \texttt{if}\ e_1\ e_2\ e_3 \,:\, \tau}\ \text{T-If}$$

$$\frac{\Sigma|\Gamma \vdash e \,:\, \tau}{\Sigma|\Gamma \vdash \texttt{return}\ e \,:\, \mu\,\tau}\ \text{T-Return}$$

$$\frac{\Sigma|\Gamma \vdash e_1 \,:\, \mu\,\tau \qquad \Sigma|\Gamma \vdash e_2 \,:\, \tau \to \mu\,\tau'}{\Sigma|\Gamma \vdash e_1 \gg\!= e_2 \,:\, \mu\,\tau'}\ \text{T-Bind}$$

$$\frac{\Sigma|\Gamma \vdash e \,:\, \texttt{IO}\,\tau}{\Sigma|\Gamma \vdash \texttt{spawn}\ e \,:\, \texttt{IO}\,()}\ \text{T-Spawn} \qquad \frac{\Sigma|\Gamma \vdash e \,:\, \texttt{STM}\,\tau}{\Sigma|\Gamma \vdash \texttt{atomic}\ e \,:\, \texttt{IO}\,\tau}\ \text{T-Atomic}$$

$$\frac{\Sigma|\Gamma \vdash e \,:\, \texttt{STM}\,\tau \qquad \Sigma|\Gamma \vdash e' \,:\, \texttt{STM}\,\tau \qquad \Sigma \vdash W \qquad \Sigma \vdash R \qquad \Sigma \vdash \mathcal{H}}{\Sigma|\Gamma \vdash (e, W, R, i, e', \mathcal{H}) \,:\, \texttt{IO}\,\tau}\ \text{T-Txn}$$

$$\frac{\Sigma|\Gamma \vdash e \,:\, \tau}{\Sigma|\Gamma \vdash \texttt{new}\ e \,:\, \texttt{STM}\,(\texttt{R}\,\tau)}\ \text{T-Alloc} \qquad \frac{\Sigma|\Gamma \vdash e \,:\, \texttt{R}\,\tau}{\Sigma|\Gamma \vdash \texttt{read}\,e \,:\, \texttt{STM}\,\tau}\ \text{T-Deref}$$

$$\frac{\Sigma|\Gamma \vdash e_1 \,:\, \texttt{R}\,\tau \qquad \Sigma|\Gamma \vdash e_2 \,:\, \tau}{\Sigma|\Gamma \vdash\ \texttt{write}\,e_1\,e_2 \,:\, \texttt{STM}\,()}\ \text{T-Assign}$$

$$\frac{\forall l \in \mathrm{dom}(\mathcal{H}) : \mathcal{H}(l) = (v, i) \Rightarrow \Sigma|\emptyset \vdash v \,:\, \Sigma(l)}{\Sigma \vdash \mathcal{H}}$$

Figure 5.3. State related definitions for the operational semantics of Λ_{STM}.

l, l'	$\in$ Ref	
t, t'	$\in$ ThreadId	
i, j	$\in$ TxnId	
$\mathcal{P}$	$\in$ ThreadPool	$=$ ThreadId $\rightharpoonup$ Exp $\times$ TxnId
T_i	$\in$ Transaction	$=$ Exp $\times$ Store $\times$ Store $\times$ TxnId $\times$ Exp $\times$ Store
$\mathcal{H}, R, W$	$\in$ Store	$=$ Ref $\rightharpoonup$ Val $\times$ TxnId
α_i	$\in$ TxnEffect	$= \{at^{t,i}, ab^{t,i}, co^{t,i}(\bar{l}), r^{t,i}(l), \epsilon^{t,i}\}$
α	$\in$ Effect	$= \{\epsilon^t, sp^t(t)\}$

The typing judgment $\Sigma|\Gamma \vdash e : \tau$ contains two environments. Σ tracks the type of memory locations, and Γ tracks the type of variables. There is a second, heap typing judgment $\Sigma \vdash \mathcal{H}$ that relates the type of each memory location to the (closed) value stored in it. The typing rules are syntax-directed and mostly standard.

5.2.2. Operational semantics for Λ_{STM}

Figure 5.3 introduces some auxiliary definitions for the operational semantics. A program state $\mathcal{H}, \mathcal{P}$ is a pair consisting of a heap and a thread pool. A thread pool maps thread identifiers to expressions to be evaluated concurrently and a thread-local transaction counter. The execution of a program is represented by a labeled transition relation between program states.

A transaction in progress is represented by a tuple $(e, W, R, i, e', \mathcal{H}')$. It consists of the expression e that is currently evaluated, the write set W and the read set R of the transaction, a (unique) transaction identifier i, a copy of the original transaction body e', and a copy $\mathcal{H}'$ of the heap taken at the beginning of the transaction. The latter two store the relevant state at the beginning of a transaction to facilitate the consistency check and the abort operation.

A reference corresponds to a heap location. All stores (the heap, the read set, and the write set of a transaction) map a reference to a pair of a value and a transaction identifier. The transaction identifier specifies the transaction which committed or, in case of the write set, attempts to commit the value to the global store. The identifier is used in the heap and the read set to detect changes in the heap while avoiding the ABA problem such that updates with a previous value are detectable for concurrently running threads. Though the operational semantics do not require identifiers for the write set, we still employ them in our formalism to have a uniform kind of store.

$S(l)$ denotes the lookup operation of a reference l in a heap S. It implies $l \in \mathrm{dom}(S)$. The store update operation $S[l \mapsto y]$ returns a store that is identical to S, except that it maps l to y. For two stores S_1 and S_2, we write $S_1[S_2]$ for the updated version of S_1 with all entries of S_2.

Operations can have different effects α on the global state: the begin transaction

Figure 5.4. Operational semantics: Local evaluation steps.

Evaluation contexts:

$$\begin{array}{lcl} \mathcal{E} & ::= & []\ e \mid \texttt{if}\ []\ e\ e' \\ \mathcal{M} & ::= & \texttt{read}\ [] \mid \texttt{write}\ []\ e \mid []\ggg e \end{array}$$

Expression evaluation $\rightarrow$:

$$\begin{array}{rcl} (\lambda x.e)\ e' & \rightarrow & e[e'/x] \\ \texttt{if tt}\ e\ e' & \rightarrow & e \\ \texttt{if ff}\ e\ e' & \rightarrow & e' \end{array}$$

$$\frac{e \rightarrow e'}{\mathcal{E}[e] \rightarrow \mathcal{E}[e']}$$

Monadic evaluation $\curvearrowright$:

$$\texttt{return}\ e' \ggg e \ \curvearrowright\ e\ e'$$

$$\frac{e \rightarrow e'}{e \curvearrowright e'} \qquad \frac{m \curvearrowright m'}{\mathcal{M}[m] \curvearrowright \mathcal{M}[m']}$$

($at^{t,i}$), abort transaction ($ab^{t,i}$), read reference l ($r^{t,i}(l)$), and commit writing references $\bar{l}$ ($co^{t,i}(\bar{l})$) indicating operations on the global shared heap, or empty effects ($\epsilon^{t,i}$ or ϵ^t), with t a thread identifier, and i a transaction id. The empty effects ϵ^t represent monadic reductions that occur outside a transaction (see top of Figure 5.5). The spawn effect $sp^t(t')$denotes the spawning of a new thread with thread id t' by thread t.

The evaluation of a program with body e starts in an initial state $\langle\rangle, \{0 \mapsto e, 0\}$ with an empty heap and a main thread with thread identifier 0. A final state has the form $\mathcal{H}, \{0 \mapsto (v_0, i_0); \ldots; t_n \mapsto (v_n, i_n)\}$. The rules in Figures 5.4 and 5.5 define the semantics of the language constructs. In Figure 5.4, the operational semantics of local evaluation steps is defined. The rules are standard for call-by-name semantics calculi. $\mathcal{E}[\bullet]$ denotes an evaluation context for an expression and $\mathcal{M}[\bullet]$ an evaluation context for monadic expressions. We write m to indicate that an expression has a monadic type. As usual, $e[e'/x]$ denotes the capture-avoiding substitution of x by e' in e.

Figure 5.5 contains the evaluation steps on the global level. The `IO` monad is the top-level evaluation mode. Each reduction step $\stackrel{\alpha}{\Longrightarrow}$ chooses an expression from the thread pool $\mathcal{P}$. The non-determinism in this choice models an arbitrary scheduling of threads.

Spawning a thread (Spawn) creates a new entry in the thread pool and returns unit in the parent thread.

Figure 5.5. Operational semantics for Λ_{STM}: Global evaluation steps.

$$\frac{m \curvearrowright m'}{\mathcal{H}, \mathcal{P}\{t \mapsto m, i\} \overset{\epsilon^t}{\Longrightarrow} \mathcal{H}, \mathcal{P}\{t \mapsto m', i\}} \text{ IO-Monad}$$

$$\frac{t' \text{ fresh}}{\mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{spawn}\ m], i\} \overset{sp^t(t')}{\Longrightarrow} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{return}\ ()], i;\ t' \mapsto m, 0\}} \text{ Spawn}$$

$$\frac{\mathcal{H}, m, i \overset{\alpha}{\rightarrowtail} \mathcal{H}', m', i'}{\mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[m], i\} \overset{\alpha}{\Longrightarrow} \mathcal{H}', \mathcal{P}\{t \mapsto \mathcal{M}[m'], i'\}} \text{ Txn}$$

All other reductions of the monadic expression under the current heap are defined by the state transformation

$$\mathcal{H}, m, i \overset{\alpha}{\rightarrowtail} \mathcal{H}', m', i'$$

The corresponding evaluation rules are given in Figure 5.6. An atomic expression at the top-level (Atomic) creates a new transaction in progress with the expression to be evaluated, an empty read and write set, and a fresh transaction identifier (that has never been used before in a particular evaluation). Further, a copy of the expression m is needed for possible rollbacks, and a copy of the current heap to mark the beginning of the transaction.

All monadic evaluation steps can take place inside a transaction (STM-Monad).

Allocation of a new reference (Alloc) must check that the reference is not yet allocated in the heap. But it must also check that the reference is not yet allocated in any concurrently running transaction to avoid accidental overwrites when both transactions commit. This condition is indicated by $l \notin \mathcal{H}, \mathcal{P}$, eschewing a formal definition.

Write operations (Write) are straightforward. They just affect the local write set and store the value along with the current transaction identifier.

The read operation on references (Read) needs to consult the global state. If a reference cannot be read from the local read or write set, it is accessed in the current global heap. To maintain the transaction's consistency, the read operation is successful only if the value has not been updated since the transaction's beginning. The value and transaction identifier as registered in the heap for this reference are then added to the read set and the value is returned to the transactional computation.

If a reference is present in the read set, but not in the write set, then its value is taken from the read set (ReadRSet).

If the reference is present in the write set, then its value is taken from the write set, without checking the read set (ReadWSet).

Figure 5.6. Operational semantics for Λ_{STM}: Evaluation steps in transactions.

$$\frac{i' = i + 1}{\mathcal{H}, \texttt{atomic}\ m, i \overset{at^{t,i}}{\rightarrowtail} \mathcal{H}, (m, \langle\rangle, \langle\rangle, i', m, \mathcal{H}), i'} \text{ STM-Monad}$$

$$\frac{m \curvearrowright m''}{\mathcal{H}, (m, W, R, i, m', \mathcal{H}'), i \overset{\epsilon^{t,i}}{\rightarrowtail} \mathcal{H}, (m'', W, R, i, m', \mathcal{H}'), i} \text{ STM-Monad}$$

$$\frac{l \notin \mathrm{dom}(\mathcal{H}) \qquad W' = W[l \mapsto (e, i)] \qquad \mathcal{H}'' = \mathcal{H}[l \mapsto (e, i)]}{\mathcal{H}, (\mathcal{M}[\texttt{new}\ e], W, R, i, m', \mathcal{H}'), i \overset{\epsilon^{t,i}}{\rightarrowtail} \mathcal{H}'', (\mathcal{M}[\texttt{return}\ l], W', R, i, m', \mathcal{H}'), i} \text{ Alloc}$$

$$\frac{W' = W[l \mapsto (e, i)]}{\mathcal{H}, (\mathcal{M}[\,\texttt{write}\, l\, e], W, R, i, m', \mathcal{H}'), i \overset{\epsilon^{t,i}}{\rightarrowtail} \mathcal{H}, (\mathcal{M}[\texttt{return}\ ()], W', R, i, m', \mathcal{H}'), i} \text{ Write}$$

$$\frac{l \notin \mathrm{dom}(W) \cup \mathrm{dom}(R) \qquad \mathcal{H}(l) \equiv \mathcal{H}'(l) \equiv (e, j) \qquad R' = R[l \mapsto (e, j)]}{\mathcal{H}, (\mathcal{M}[\texttt{read}\, l], W, R, i, m', \mathcal{H}'), i \overset{r^{t_i,i}(l)}{\rightarrowtail} \mathcal{H}, (\mathcal{M}[\texttt{return}\ e], W, R', i, m', \mathcal{H}'), i} \text{ Read}$$

$$\frac{l \notin \mathrm{dom}(W) \qquad R(l) = (e, j)}{\mathcal{H}, (\mathcal{M}[\texttt{read}\, l], W, R, i, m', \mathcal{H}'), i \overset{\epsilon^{t,i}}{\rightarrowtail} \mathcal{H}, (\mathcal{M}[\texttt{return}\ e], W, R, i, m', \mathcal{H}'), i} \text{ ReadRSet}$$

$$\frac{W(l) = (e, i)}{\mathcal{H}, (\mathcal{M}[\texttt{read}\, l], W, R, i, m', \mathcal{H}'), i \overset{\epsilon^{t,i}}{\rightarrowtail} \mathcal{H}, (\mathcal{M}[\texttt{return}\ e], W, R, i, m', \mathcal{H}'), i} \text{ ReadWSet}$$

$$\mathcal{H}, (m, W, R, i, m', \mathcal{H}'), i \overset{ab^{t,i}}{\rightarrowtail} \mathcal{H}, \texttt{atomic}\ m', i + 1 \qquad \text{Rollback}$$

$$\frac{\mathit{check}(R, \mathcal{H}) = \mathit{ok} \qquad \mathcal{H}'' = \mathcal{H}[W] \qquad \bar{l} = \mathrm{dom}(W)}{\mathcal{H}, (\texttt{return}\ e, W, R, i, m', \mathcal{H}'), i \overset{co^{t,i}(\bar{l})}{\rightarrowtail} \mathcal{H}'', \texttt{return}\ e, i + 1} \text{ Commit}$$

Figure 5.7. Operational semantics for Λ_{STM}: Helper relations for heap checks.

$$\frac{\forall l \in \mathrm{dom}(R) : R(l) = \mathcal{H}(l)}{\mathit{check}(R, \mathcal{H}) = \mathit{ok}} \text{ Check-Ok} \qquad \frac{\exists l \in \mathrm{dom}(R) : R(l) \neq \mathcal{H}(l)}{\mathit{check}(R, \mathcal{H}) = \mathit{bad}} \text{ Check-Bad}$$

If none of the preceding three cases holds at a read, then the transaction aborts and rolls back via ROLLBACK by abandoning the transaction in progress and reinstalling the saved transaction body m' as an atomic block. In fact, this rule has no precondition so that a rollback may happen non-deterministically at any time during a transaction. This way, it is easy to extend our model with an explicit user abort or retry operation. Furthermore, this rule covers the abort both when reading fails as well as when the commit operation fails. However, for performance reasons, an actual implementation applies ROLLBACK only if no other transactional rule is applicable.

When committing (COMMIT), the heap is checked for updates to the references which are found in the transaction's read set since the start of the transaction. There are two cases:

The check is successful (CHECK-OK): None of the variables read by the transaction have been committed by another transaction in the meantime. Therefore, the transaction may publish its writes atomically to the shared heap and return to the IO monad.

The check fails (CHECK-BAD): The only applicable rule is ROLLBACK. The transaction aborts and restarts.

As for the transaction counter i, it is incremented when starting and when finishing the evaluation of a transaction in progress.

Each of the reductions also generates the appropriate effect label on the transition relation. Thus, each sequence of labeled reductions uniquely determines a sequence of labels, which we call the trace of the reduction sequence. Unlike other formalizations, the interleaving of transactions as well as the abort operations are visible in the trace.

Theorem 5.2.1 (Type soundness)**.** *The type system in Figure 5.2 is sound with respect to the operational semantics of* Λ_{STM}*.*

Proof of 5.2.1: The proof is by establishing type preservation and progress in the usual way [79]. The proof of progress relies crucially on the use of the ROLLBACK rule if the comparison of heap entries in READ or COMMIT fails. □

5.2.3. Deterministic allocation

The rule for allocating shared memory, ALLOC, requires the creation of new references. In the formalization, these references are used in two ways, in the evaluation semantics and in the effects. Semantically, the references are needed to identify heap entries. For the correctness of the system the references need to be unique, but otherwise no restrictions are given.

In the effects, the references are used to highlight the interaction of threads via shared memory. To simplify later the discussion about equivalence of traces, we

assume that the same references are used for a thread under each schedule when allocating. Similarly, we require that spawning a thread in SPAWN assigns the same thread id to a newly spawned thread under any schedule.

These deterministic generation schemes for references and identifiers can be modelled in the formal semantics by introducing thread-specific counters. We avoid cluttering the semantical framework with these auxiliary counters, but assume in the following that the generation of identifiers and references is equal under each schedule.

5.3. Opacity

The standard isolation property that most STM systems provide is *opacity* [31]. It states that any allowed interleaving of transactions must have an equivalent serialized execution. Furthermore, even aborting transactions are required to observe memory locations only in a consistent way.

In this section, we prove formally that the semantics for Λ_{STM} satisfies opacity. To this end, we give a definition for well-formedness of execution traces in terms of the effects they exhibit. We then show that reordering certain evaluation steps leads to equivalent reductions sequences. Reductions are considered equivalent if every read operation returns the same value, every commit operation commits the same values, and every transaction's outcome (abort or commit) is the same. To see which reordering yields equivalent reductions, we define a notion of dependency on effects.

In contrast to other definitions of well-formed execution traces (e.g. [76]), we do not take the values of memory locations into account. The operational semantics guarantees that each transaction is working on a consistent view of the shared memory as indexed by its time stamp. A read operation returns the last value written, either by another transaction which updated the global heap, or by the transaction itself in a local write step. Further, all write operations are published (i.e., made visible to other transactions) only after the successful commit. Therefore, the trace reflects the order of the globally visible effects of the read and write operations. The local reads and writes have no globally visible effect.

Finally, we show that all reduction sequences produced by the operational semantics are equivalent to some reduction sequence with a serial trace, up to the assignment of unique labels to the transactions. Note that we only consider finite traces which correspond to programs running a finite amount of time. For infinite traces, we are able to establish our results for all finite prefixes.

5.3.1. Effect traces

We start with a formal account on effect traces.

Definition 5.3.1 (Effect traces)**.** *A* trace $\bar{\alpha}$ *is a finite sequence* $\alpha_1 \dots \alpha_n$ *of effects* $\alpha_i \in$ *Effect for* $i \in 1, \dots, n$.

A total order on the effects $\alpha \in \bar{\alpha}$ is defined by their position in the effect trace. For $i, j \in \{1, \ldots, |\bar{\alpha}|\}$ and $i < j$, we use the abbreviation

$$\bar{\alpha} \vdash \alpha_i < \alpha_j$$

to denote that an effect α_i is happening before α_j in an trace $\bar{\alpha}$. We often abbreviate

$$\bar{\alpha} \vdash \alpha_i < \alpha_j < \alpha_k$$

if $\bar{\alpha} \vdash \alpha_i < \alpha_j$ and $\bar{\alpha} \vdash \alpha_j < \alpha_k$.

Similarly,

$$\bar{\alpha} \vdash \bar{\alpha}_1 < \bar{\alpha}_2$$

extends the relation to sets of effects if it holds pairwise for all elements in the disjoint sets $\bar{\alpha}_1$ and $\bar{\alpha}_2$, where $\bar{\alpha}_1, \bar{\alpha}_2 \subset \bar{\alpha}$.

We identify by α an arbitrary effect from a trace $\bar{\alpha}$. α^t denotes an (arbitrary) effect from thread t, and $\alpha^{t,i}$ an effect from transaction T_i in thread t. To distinguish between transactions running on different threads, we often use the transaction identifier as subscript to the thread id, for example $\alpha^{t_i,i}$.

Further, $\bar{\alpha}|_t = \{\alpha^t \in \bar{\alpha}\}$ is the subset of all effects from thread t, and $\bar{\alpha}|_{t,i} = \{\alpha^{t,i} \in \bar{\alpha}\}$ the subsets of all effects from transaction i in thread t.

Effect traces encode the scheduling of threads during the execution of a program. Additionally, non-empty effects encode the side-effects on the program state such as thread spawning or operations on the heap. Empty effects have no influence on the globally shared state. We define an operation $\langle\cdot\rangle$ which reduces a trace to its *kernel*, the ordered sequence of non-empty effects.

Definition 5.3.2. *The kernel of a trace $\langle\bar{\alpha}\rangle$ is the reduction of the trace to its non-empty effects.*

$$\begin{aligned} \langle\emptyset\rangle &= \emptyset \\ \langle\alpha, \bar{\alpha}\rangle &= \begin{cases} \langle\bar{\alpha}\rangle & \text{if } \alpha = \epsilon^{t,i} \text{ or } \alpha = \epsilon^t \\ \alpha, \langle\bar{\alpha}\rangle & \text{otherwise} \end{cases} \end{aligned}$$

Definition 5.3.3. *A trace $\bar{\alpha}$ is* equal in effects *to a trace $\bar{\beta}$ if*

$$\langle\bar{\alpha}\rangle \equiv \langle\bar{\beta}\rangle$$

A trace is well-formed if it does not violate some obvious rules related to the the order of transactional and non-transactional effects. These rules mainly concern the proper nesting of transactions and threads.

Definition 5.3.4 (Well-formed traces)**.** *A trace $\bar{\alpha}$ is* well-formed *if the following conditions hold:*

- *There is no effect for a thread before its spawn effect, unless it is the main thread.*

$$\forall t \neq 0 : \alpha^t \in \bar{\alpha} \Rightarrow \exists t' : \; sp^{t'}(t) \in \bar{\alpha} \;\wedge\; \bar{\alpha} \vdash sp^{t'}(t) < \alpha^t$$

- *For each transactional effect, there is a corresponding atomic effect in the trace.*

$$\alpha^{t,i} \in \bar{\alpha} \Rightarrow at^{t,i} \in \bar{\alpha}$$

- *There is no effect for a transaction T_i before its atomic effect.*

$$\begin{aligned} \forall at^{t,i} \in \bar{\alpha} : \quad r^{t,i}(l) \in \bar{\alpha} &\Rightarrow \bar{\alpha} \vdash at^{t,i} < r^{t,i}(l) \\ co^{t,i}(\bar{l}) \in \bar{\alpha} &\Rightarrow \bar{\alpha} \vdash at^{t,i} < co^{t,i}(\bar{l}) \\ ab^{t,i} \in \bar{\alpha} &\Rightarrow \bar{\alpha} \vdash at^{t,i} < ab^{t,i} \\ \epsilon^{t,i} \in \bar{\alpha} &\Rightarrow \bar{\alpha} \vdash at^{t,i} < \epsilon^{t,i} \end{aligned}$$

- *There is no effect for a transaction T_i after its commit effect.*

$$\begin{aligned} \forall co^{t,i}(\bar{l}) \in \bar{\alpha} : \quad r^{t,i}(l) \in \bar{\alpha} &: \bar{\alpha} \vdash r^{t,i}(l) < co^{t,i}(\bar{l}) \\ \epsilon^{t,i} \in \bar{\alpha} &: \bar{\alpha} \vdash \epsilon^{t,i} < co^{t,i}(\bar{l}) \end{aligned}$$

  Similarly, there is no effect for a transaction T_i after its abort effect.

$$\begin{aligned} \forall ab^{t,i} \in \bar{\alpha} : \quad r^{t,i}(l) \in \bar{\alpha} &: \bar{\alpha} \vdash r^{t,i}(l) < ab^{t,i} \\ \epsilon^{t,i} \in \bar{\alpha} &: \bar{\alpha} \vdash \epsilon^{t,i} < ab^{t,i} \end{aligned}$$

- *A transaction may have either a commit or an abort effect, but not both.*

$$\begin{aligned} co^{t,i}(\bar{l}) \in \bar{\alpha} \quad &\Rightarrow \quad ab^{t,i} \notin \bar{\alpha} \\ ab^{t,i} \in \bar{\alpha} \quad &\Rightarrow \quad co^{t,i}(\bar{l}) \notin \bar{\alpha} \end{aligned}$$

- *There are no non-transactional effects within a transaction.*

$$\begin{aligned} \epsilon^t \in \bar{\alpha} \quad &\Rightarrow \quad \nexists i : \bar{\alpha} \vdash at^{t,i} < \epsilon^t < co^{t,i}(\bar{l}) \;\vee\; \bar{\alpha} \vdash at^{t,i} < \epsilon^t < ab^{t,i} \\ sp^t(t') \in \bar{\alpha} \quad &\Rightarrow \quad \nexists i : \bar{\alpha} \vdash at^{t,i} < sp^t(t') < co^{t,i}(\bar{l}) \;\vee\; \bar{\alpha} \vdash at^{t,i} < sp^t(t') < ab^{t,i} \end{aligned}$$

- *Transactional effects from the same thread do not interleave.*

$$\forall t \, \forall i \neq j : \bar{\alpha} \vdash \bar{\alpha}|_{t,i} < \bar{\alpha}|_{t,j} \;\vee\; \bar{\alpha} \vdash \bar{\alpha}|_{t,j} < \bar{\alpha}|_{t,i}$$

It follows directly from the definition of well-formedness that a trace contains for each transaction exactly one atomic effect and at most one commit or abort effect.

Definition 5.3.5 (Pending transactions)**.** *A transaction T_i is* pending *in a trace $\bar{\alpha}$ if it has neither a commit nor an abort effect:*

$$ab^{t,i} \notin \bar{\alpha} \;\wedge\; co^{t,i}(\bar{l}) \notin \bar{\alpha}$$

Beside the total order that is defined by the position in a trace, another partial order connects effects based on their interdependence.

Definition 5.3.6 (Control dependency)**.** *An effect α has a* control dependency *on an effect α', $\alpha \triangleright_c \alpha'$, if α precedes α' in the control flow of the program. Hence, a control dependency is given if*

- *α and α' are effects from the same transaction and $\bar{\alpha} \vdash \alpha < \alpha'$, or*
- *$\alpha = sp^t(t')$ and α' is from thread t'.*

Definition 5.3.7 (Data dependency)**.** *An effect α has a* data dependency *on an effect α', $\alpha \triangleright_d \alpha'$ if they exhibit a read-write, write-read, or write-write dependency on the same memory location, and $\bar{\alpha} \vdash \alpha < \alpha'$. Hence, a data dependency is given in the following cases ($i \neq j$):*

- *$r^{t_i,i}(l) \triangleright_d co^{t_j,j}(\bar{l})$ and $l \in \bar{l}$*
- *$co^{t_i,i}(\bar{l}) \triangleright_d r^{t_j,j}(l)$ and $l \in \bar{l}$*
- *$co^{t_i,i}(\bar{l}) \triangleright_d co^{t_j,j}(\bar{l}')$ and $\bar{l} \cap \bar{l}' \neq \emptyset$*

Definition 5.3.8 (Dependency)**.** *An effect α is* dependent *on an effect α' if α has either a control or data dependency on α':*

$$\alpha \triangleright \alpha' \quad \textit{iff} \quad \alpha \triangleright_c \alpha' \vee \alpha \triangleright_d \alpha'$$

Effects that are not dependent on each other are called independent*.*

A transaction T_i is dependent on another transaction T_j if $\alpha \triangleright \alpha'$ for an effect α from T_i and an effect α' from T_j.

Definition 5.3.9 (Trace dependencies)**.** *Let $\bar{\alpha}$ be a well-formed trace. The trace dependencies $\Delta(\bar{\alpha})$ are defined as the set of all pairs of dependent effects in this trace:*

$$\Delta(\bar{\alpha}) = \{(\alpha_i, \alpha_j) \mid \alpha_i \triangleright \alpha_j\}$$

The trace dependencies impose a partial order on a trace. In the following section, we show how to reorder traces into serial traces while preserving this partial order.

5.3.2. Trace anomalies

For further characterization of effect traces, we follow the definition of anomalies in transactions given by Berenson et al. [8]. Our formalization of traces impedes dirty reads and dirty writes by design because updates to shared data is only visible to other transactions after a commit. Our definition of effects does not distinguish between writes and commits, but merges them together into one commit effect. We therefore can focus here on the remaining four types of anomalies.

Definition 5.3.10 (Non-repeatable reads)**.** *A transaction T_i experiences a* non-repeatable read *if*

$$\exists x : \bar{\alpha} \vdash r^{t_i,i}(x) < co^{t_j,j}(\bar{l}) < r^{t_i,i}(x) \text{ and } x \in \bar{l}.$$

Definition 5.3.11 (Lost updates)**.** *A transaction causes a* lost update *if*

$$\exists x : \bar{\alpha} \vdash r^{t_i,i}(x) < co^{t_j,j}(\bar{l}) < co^{t_i,i}(\bar{l}') \text{ and } x \in \bar{l} \cap \bar{l}'.$$

Definition 5.3.12 (Read skew)**.** *A transaction T_i exhibits a* read skew *with some other transaction T_j ($i \neq j$) if*

$$\exists x, y : \bar{\alpha} \vdash r^{t_i,i}(x) < co^{t_j,j}(\bar{l}) < r^{t_i,i}(y) \text{ and } x, y \in \bar{l}.$$

Definition 5.3.13 (Write skew)**.** *Two transactions T_i and T_j, $i \neq j$, exhibit a* write skew *in a trace $\bar{\alpha}$ if there exists locations x and y such that*

$$r^{t_i,i}(x), r^{t_i,i}(y), r^{t_j,j}(x), r^{t_j,j}(y), co^{t_i,i}(\bar{l}_1), co^{t_j,j}(\bar{l}_2) \in \bar{\alpha}$$

and

$$\bar{\alpha} \vdash at^{t_i,i} < co^{t_j,j}(\bar{l}_2)$$
$$\bar{\alpha} \vdash at^{t_j,j} < co^{t_i,i}(\bar{l}_1)$$

with $x \in \bar{l}_1$ and $y \in \bar{l}_2$.

5.3.3. Serializing effect traces

We use the characterization of anomalies to now define the isolation level of serializability. According to Berenson et al. [8], in a serializable transactional system

1. all reads are repeatable,
2. there are no lost updates,
3. there are no read skews, and
4. there are no write skews.

It is easy to see that the semantics of Λ_{STM} preclude the anomalies from Section 5.3.2. This is mainly due to the heap checks when reading or committing shared data items. In the following we give a constructive proof of how to obtain a corresponding serial execution trace for any trace that is obtained by executing a program under the semantics of Λ_{STM}.

Definition 5.3.14 (Serial traces)**.** *A well-formed trace $\bar{\alpha}$ is serial if for any two transactions T_i and T_j ($i \neq j$), all effects from T_i occur before all effects from T_j, or vice versa:*

$$\forall\, i \neq j : \bar{\alpha} \vdash \bar{\alpha}|_{t_i,i} < \bar{\alpha}|_{t_j,j} \text{ or } \bar{\alpha} \vdash \bar{\alpha}|_{t_j,j} < \bar{\alpha}|_{t_i,i}$$

In contrast to other approaches, we do not exclude aborting or pending transactions in the definition for serial traces. Therefore, we actually model opaque traces.

The definition of serial traces only orders transactional effects with respect to each other, but does not specify the relative order of non-transactional effects. Under weak atomicity, it is possible for non-transactional effects to occur during a transaction's execution whereas strong atomicity precludes this behavior. To capture the notion of strong atomicity, we introduce the notion of strongly serial traces in the following definition.

Definition 5.3.15 (Strongly serial traces)**.** *A well-formed trace $\bar{\alpha}$ is strongly serial if every effect that is ordered between two effects from the same transaction is an effect from the same transaction.*

$$\forall \alpha^{t,i}, \alpha'^{t,i} : \bar{\alpha} \vdash \alpha^{t,i} < \beta < \alpha'^{t,i} \Rightarrow \beta = \alpha'^{t,i}$$

For the construction of strongly serial traces, we are interested in equivalence classes of traces that are permutations of each other and encode the same side-effects on the program state. However, there are restrictions on what reorderings of effects are permissible. For example, the order of trace items with respect to one thread must not be changed. In short, permissible permutations respect the dependencies of effects.

Definition 5.3.16 (Equivalence of traces)**.** *A trace $\bar{\alpha}$ is* equivalent *to a trace $\bar{\beta}$ if $\bar{\alpha}$ is a permutation of $\bar{\alpha}'$ and $\Delta(\bar{\alpha}) = \Delta(\bar{\beta})$.*

Traces that have an equivalent serial trace are often called *conflict serializable* in the literature [76].

In the remainder of this section, we identify which subsequences of a trace are not serial, and specify an algorithm that moves the effects to the appropriate place.

Lemma 5.3.1 (Conflicts)**.** *Let $\bar{\alpha}$ be a serializable trace. Then $\bar{\alpha}$ is either strongly serial, or there exists an α_k such that the prefix $\alpha_1 \dots \alpha_k$ is strongly serial and*

1. *α_k and α_{k+1} are independent, or*
2. *$\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = co^{t_j,j}(\bar{l})$ with $l \in \bar{l}$.*

Proof of 5.3.1: We consider all possible combinations of effects which might occur in a well-formed trace. Suppose that $i \neq j$.
Case distinction on α_k and α_{k+1}.

- *Case* $\alpha_k = \epsilon^t$ or $\alpha_{k+1} = \epsilon^t$: serial or independent.
- *Case* $\alpha_k = \epsilon^{t_i,i}$ or $\alpha_{k+1} = \epsilon^{t_j,j}$: serial or independent.
- *Case* $\alpha_k = at^{t_i,i}$ and $\alpha_{k+1} = at^{t_j,j}$: serial.
- *Case* $\alpha_k = at^{t_i,i}$ and $\alpha_{k+1} = r^{t_i,i}(l)$: serial.

- *Case* $\alpha_k = at^{t_i,i}$ and $\alpha_{k+1} = r^{t_j,j}(l)$: independent.
- *Case* $\alpha_k = at^{t_i,i}$ and $\alpha_{k+1} = co^{t_i,i}(\bar{l})$: serial.
- *Case* $\alpha_k = at^{t_i,i}$ and $\alpha_{k+1} = co^{t_j,j}(\bar{l})$: independent.
- *Case* $\alpha_k = at^{t_i,i}$ and $\alpha_{k+1} = ab^{t_i,i}$: serial.
- *Case* $\alpha_k = at^{t_i,i}$ and $\alpha_{k+1} = ab^{t_j,j}$: independent.
- *Case* $\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = r^{t_j,j}(l')$: independent.
- *Case* $\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = r^{t_i,i}(l')$: serial.
- *Case* $\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = co^{t_i,i}(\bar{l})$: serial.
- *Case* $\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = co^{t_j,j}(\bar{l})$: If $l \in \bar{l}$, then this is the second case in the lemma. Otherwise independent.
- *Case* $\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = ab^{t_i,i}$: serial.
- *Case* $\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = ab^{t_j,j}$: independent.
- *Case* $\alpha_k = co^{t_i,i}(\bar{l})$ and $\alpha_{k+1} = co^{t_j,j}(\bar{l'})$: According to the operational semantics, it must hold that $\bar{l} \cap \bar{l'} = \emptyset$. Therefore, the effects are independent.
- *Case* $\alpha_k = co^{t_i,i}(\bar{l})$ and $\alpha_{k+1} = ab^{t_j,j}$: independent.

End case distinction on α_k and α_{k+1}. Cases that are left out violate the criterion for well-formedness. □

In our semantics, the begin of a transaction defines its relative order to other transactions. This order is only partial. Consider for example transactions that perform their operations interleaved. Interleaved transactions only commit successfully together if their operations do not conflict with each other. The following lemma shows that for these transactions, any relative order is admissible.

Lemma 5.3.2 (Permutation of effects)**.** *Let* $\bar{\alpha}$ *be a serializable trace with* $\bar{\alpha} = \bar{\alpha}' r^{t_j,j}(l) co^{t_i,i}(\bar{l})$ *where* $\bar{\alpha} \vdash at^{t_i,i} < r^{t_j,j}(l)$ *and the prefix* $\bar{\alpha}' r^{t_j,j}(l)$ *is strongly serial. Then, there exists at least one effect* α_i *in* $\bar{\alpha}'$ *such that*

$$\bar{\alpha} \vdash at^{t_i,i} < \alpha_i$$

and α_i *is from another thread.*

Further, assume $\alpha_m = at^{t_i,i}$ *and let* α_k *be the effect with the smallest index such that* $\bar{\alpha} \vdash at^{t_i,i} < \alpha_k$ *and* α_k *is from another thread. Then,* $\bar{\alpha}$ *is equivalent to a trace* $\bar{\beta}$ *where* α_k *is moved just before* $at^{t_i,i}$*:*

$$\bar{\beta} = \alpha_1 \dots \alpha_{m-1} \alpha_k at^{t_i,i} \alpha_{m+1} \dots \alpha_{k-1} \alpha_{k+1} \dots \alpha_n$$

Algorithm 5.1 Reordering transactions for opacity.

while $\bar{\alpha}$ is not strongly serial **do**
  choose α_k and α_{k+1} such that $\alpha_1 \dots \alpha_k$ is strongly serial
    and $\alpha_1 \dots \alpha_{k+1}$ is not strongly serial
  if α_k and α_{k+1} are independent **then**
    swap α_k with α_{k+1}
  else if $\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = co^{t_j,j}(\bar{l})$ with $l \in \bar{l}$ **then**
    choose α_m with the smallest index such that
      $\bar{\alpha} \vdash at^{t_i,i} < \alpha_m$ and $\alpha_m = \alpha^{t'}$ with $t \neq t'$
    move α_m before $at^{t_i,i}$
  else
    report error
  end if
end while

Proof of 5.3.2: Assume that there does not exist such an effect α_i. Because the prefix is strongly serial, the trace $\bar{\alpha}$ must have the following structure:

$$\bar{\alpha} = \alpha_{pre}\, at^{t_i,i} \left(r^{t_i,i}(\bullet) \mid \epsilon^{t_i,i}\right)^* \gamma\, at^{t_j,j} \left(r^{t_j,j}(\bullet) \mid \epsilon^{t_j,j}\right)^* r^{t_j,j}(l)\, co^{t_i,i}(\bar{l})$$

where γ is a (possibly empty) sequence of (possibly pending) transactions and non-transactional effects.

$$\gamma = \left(at^{\diamond,\star}\, (r^{\diamond,\star}(\bullet) \mid \epsilon^{\diamond,\star})^*\, (ab^{\diamond,\star} \mid co^{\diamond,\star}(\bar{\bullet}))^? \mid \epsilon^{\diamond} \mid sp^{\diamond}(\diamond)\right)^*$$

Here, $*$ abbreviates a (possibly empty) sequence of effects, ? abbreviates a choice between two effects, $\bullet$ represents some reference, $\star$ an transaction identifier, and $\diamond$ a thread identifier.

Now, let α_k be the first element of γ or, if γ is empty, let $\alpha_k = at^{t_m,m}$. As α_k is either an atomic effect or a non-transactional effect, there are no dependencies between α_k and any effect from transaction i preceding it in $\bar{\alpha}$. Moving α_k now before $at^{t_i,i}$ is therefore neither introducing nor removing any dependencies in $\bar{\alpha}$. By definition, $\bar{\beta}$ is then equivalent to $\bar{\alpha}$. □

For the proof of opacity, we define an algorithm which produces for a serializable trace an equivalent serial trace.

The following example demonstrates how the algorithm proceeds on a well-formed, serializable trace. Consider the trace

$$at^{t_1,1}\, at^{t_2,2}\, r^{t_1,1}(x)\, at^{t_3,3}\, r^{t_3,3}(y)\, co^{t_3,3}(y)\, r^{t_2,2}(x)\, co^{t_1,1}(x)\, ab^{t_2,2}\, at^{t_2,2'} \dots$$

The trace abstracts from three concurrently running transactions. Transactions 1 and 3 commit successfully whereas transaction 2 is aborted and restarted in the end. To simplify matters, we concentrate on the relevant effects and omit empty

effects that are due to thread-local and administrative reduction steps. Further, we assume that each transaction is running in its own thread.

The algorithm starts moving the read effects of transaction 1 and 2 towards their atomic effect.

$$at^{t_1,1}\ r^{t_1,1}(x)\ at^{t_2,2}\ r^{t_2,2}(x)\ at^{t_3,3}\ r^{t_3,3}(y)\ co^{t_3,3}(y)\ co^{t_1,1}(x)\ ab^{t_2,2}\ at^{t_2,2'}\ \ldots$$

While swapping the commit effect of transaction 1 to the front, it reaches this state:

$$at^{t_1,1}\ r^{t_1,1}(x)\ at^{t_2,2}\ r^{t_2,2}(x)\ co^{t_1,1}(x)\ at^{t_3,3}\ r^{t_3,3}(y)\ co^{t_3,3}(y)\ ab^{t_2,2}\ at^{t_2,2'}\ \ldots$$

As there is a read-write dependency between transaction 2 and 1 which is observable by the effects $r^{t_2,2}(x)$ and $co^{t_1,1}(x)$, a swap of these effects would yield a non-equivalent trace. However, a reordering of the effects from transaction 1 and 2 in the prefix does not introduce or remove dependencies in the trace. Therefore, the algorithm applies in this case the strategy of permuting transactional prefixes.

$$at^{t_2,2}\ r^{t_2,2}(x)\ at^{t_1,1}\ r^{t_1,1}(x)\ co^{t_1,1}(x)\ at^{t_3,3}\ r^{t_3,3}(y)\ co^{t_3,3}(y)\ ab^{t_2,2}\ at^{t_2,2'}\ \ldots$$

Finally, the abort effect of transaction 2 can be safely swapped to the front. The resulting trace corresponds to a sequential execution of the transactions in the order:

$$T_2, T_1, T_3, T_{2'}, \ldots$$

The algorithm in Figure 5.1 and has the following properties:

1. It terminates on all well-formed traces without an error.
2. For any well-formed trace, it yields an equivalent serial trace.

We prove these properties in several steps.

Lemma 5.3.3 (Termination)**.** *The algorithm terminates on all serializable traces without an error.*

Proof of 5.3.3: We define a cost function covering the two kinds of permutations done by the algorithm.

Let $\mathtt{pos}_{\bar{\alpha}}(\alpha)$ be the position of effect α in a trace $\bar{\alpha}$. For each effect α from transaction i in $\bar{\alpha}$ we define

$$d_{\bar{\alpha}}(\alpha) = \mathtt{pos}_{\bar{\alpha}}(\alpha) - \mathtt{pos}_{\bar{\alpha}}(at^{t_i,i})$$

Let n be the length of trace $\bar{\alpha}$. The cost of an effect is then given by

$$\mathtt{cost}_{\bar{\alpha}}(\alpha) = \begin{cases} d_{\bar{\alpha}}(\alpha)\left(n - \mathtt{pos}_{\bar{\alpha}}(at^{t_i,i})\right), & \text{if } \alpha \text{ is an effect from transaction } i \\ 0, & \text{if } \alpha \text{ is a non-transactional effect} \end{cases}$$

The total effect cost for a trace is given by

$$\mathtt{cost}_{\mathtt{swap}}(\bar{\alpha}) = \sum_{\alpha \in \bar{\alpha}} \mathtt{cost}_{\bar{\alpha}}(\alpha)$$

Besides the effect cost, we introduce a second cost function for traces.

$$\mathtt{cost}_{\mathtt{perm}}(\bar{\alpha}) = \left| \left\{ at^{t_i,i} \mid \exists \alpha^{t'}, t' \neq t \text{ with } \bar{\alpha} \vdash at^{t_i,i} < \alpha^{t'} < co^{t_i,i}(\bar{l}) \right\} \right|$$

Finally, the total cost of a trace is given by pairing both types of cost.

$$\mathtt{cost}\,(\bar{\alpha}) = \left(\mathtt{cost}_{\mathtt{perm}}(\bar{\alpha}), \mathtt{cost}_{\mathtt{swap}}(\bar{\alpha})\right)$$

We define an order on these trace costs as follows:

$$\mathtt{cost}\,(\bar{\alpha}_1) < \mathtt{cost}\,(\bar{\alpha}_2)$$

if $\mathtt{cost}_{\mathtt{perm}}(\bar{\alpha}_1) < \mathtt{cost}_{\mathtt{perm}}(\bar{\alpha}_2)$, or $\mathtt{cost}_{\mathtt{perm}}(\bar{\alpha}_1) = \mathtt{cost}_{\mathtt{perm}}(\bar{\alpha}_2)$ and $\mathtt{cost}_{\mathtt{swap}}(\bar{\alpha}_1) < \mathtt{cost}_{\mathtt{swap}}(\bar{\alpha}_2)$.

By Lemma 5.3.1 the following cases for α_k and α_{k+1} are possible in any iteration of Algorithm 5.1:
Case distinction on α_k and α_{k+1}.

- *Case* $\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = co^{t_j,j}(\bar{l})$: Due to the structure of the prefix as shown in the proof for Lemma 5.3.2, moving an effect of another transaction before the atomic effect of transaction j decreases the first part of the cost pair.

  $$\mathtt{cost}_{\mathtt{perm}}(\bar{\alpha}') = \mathtt{cost}_{\mathtt{perm}}(\bar{\alpha}) - 1$$

  Hence, it holds that

  $$\mathtt{cost}\,(\bar{\alpha}') < \mathtt{cost}\,(\bar{\alpha})$$

- *Case* $\alpha_k = \alpha^{t_i,i}$ and $\alpha_{k+1} = \alpha^{t_j,j}$: Before the permutation, the cost for α_k and α_{k+1} are

  $$\begin{aligned} \mathtt{cost}_{\bar{\alpha}}(\alpha^{t_i,i}) &= d_{\bar{\alpha}}(\alpha^{t_i,i})\left(n - \mathtt{pos}_{\bar{\alpha}}(at^{t_i,i})\right) \\ \mathtt{cost}_{\bar{\alpha}}(\alpha^{t_j,j}) &= d_{\bar{\alpha}}(\alpha^{t_j,j})\left(n - \mathtt{pos}_{\bar{\alpha}}(at^{t_j,j})\right) \end{aligned}$$

  Let $\bar{\alpha}'$ be the trace after swapping α_k and α_{k+1}. The costs are then in the new trace given by

  $$\begin{aligned} \mathtt{cost}_{\bar{\alpha}'}(\alpha^{t_i,i}) &= \left(d_{\bar{\alpha}}(\alpha^{t_i,i}) + 1\right)\left(n - \mathtt{pos}_{\bar{\alpha}}(at^{t_i,i})\right) \\ \mathtt{cost}_{\bar{\alpha}'}(\alpha^{t_j,j}) &= \left(d_{\bar{\alpha}}(\alpha^{t_j,j}) - 1\right)\left(n - \mathtt{pos}_{\bar{\alpha}}(at^{t_j,j})\right) \end{aligned}$$

  All other swap costs stay the same. Summing up the total costs for $\bar{\alpha}'$ yields

  $$\mathtt{cost}\,(\bar{\alpha}') = \left(\mathtt{cost}_{\mathtt{perm}}(\bar{\alpha}), \mathtt{cost}_{\mathtt{swap}}(\bar{\alpha}) + \mathtt{pos}_{\bar{\alpha}}(at^{t_i,i}) - \mathtt{pos}_{\bar{\alpha}}(at^{t_j,j})\right)$$

As the prefix $\alpha_1, ..., \alpha_k$ of $\bar{\alpha}'$ is serial, it must hold that $\bar{\alpha} \vdash at^{t_i,i} < at^{t_j,j}$. Therefore, it holds that

$$\texttt{cost}\,(\bar{\alpha}') < \texttt{cost}\,(\bar{\alpha})$$

As an aside, the permutation costs for the trace is also decreased in a swap step when atomic or commit effects are involved in the swapping.

End case distinction on α_k and α_{k+1}.

According to Lemma 5.3.1, a not strongly serial, but serializable trace introduces either a situation when a swap of effects or moving effects out of a transaction's scope becomes necessary. For a serializable trace as input, the algorithm therefore never ends up in the error case.

□

Lemma 5.3.4 (Permutation)**.** *The output of the algorithm is a permutation of the input trace.*

Proof of 5.3.4: All operations on the trace are permutations of effects. Therefore, effects are neither removed from nor added to the input trace. □

Lemma 5.3.5 (Dependencies)**.** *The algorithm does not change any dependencies in the trace.*

Proof of 5.3.5: Effects are only swapped when they are independent or when permuting transactions. In the latter case, the dependencies in the trace are respected as is shown in Lemma 5.3.2. □

Lemma 5.3.6 (Correctness of the algorithm)**.** *The output of the algorithm is an equivalent serial trace.*

Proof of 5.3.6: By 5.3.4 and 5.3.5, the output is equivalent to the input trace. By Lemma 5.3.3, the algorithm terminates on all traces from type-correct programs. In this case, the condition for entering the while loop is falsified, and therefore the trace is strongly serial. □

5.3.4. Serializable traces in Λ_{STM}

We prove now that we can permute any reduction trace from executing a program in Λ_{STM} in an admissible way such that the resulting trace is strongly serial, and that the execution of the strongly serial trace yields an equivalent final program state.

Definition 5.3.17 (Reduction trace)**.** *For a program e in Λ_{STM}, the initial state of execution is given by an empty heap and a thread pool containing the main thread with thread id 0:*

$$\mathcal{H}_0, \mathcal{P}_0 = \emptyset, [0 \mapsto (e, 0)]$$

A reduction trace $\mathcal{R}$ for e is a finite sequence of reduction steps on the initial state:

$$\mathcal{R} = \mathcal{H}_0, \mathcal{P}_0 \stackrel{\alpha_1}{\Longrightarrow} \ldots \stackrel{\alpha_n}{\Longrightarrow} \mathcal{H}_n, \mathcal{P}_n$$

The effect trace corresponding to a reduction trace is given by the sequence of effects $\alpha_1 \ldots \alpha_n$.

To compare reduction traces of a program under different schedules, we define an equivalence relation on the states as they appear in the reduction trace. Under equivalent effect traces, the heap reaches the same state as the same update operations are executed on it. Program states are equivalent either if they are (syntactically) equal for each thread or, if a thread is currently executing a transaction, the respective transaction tuples only differ on entries in the reference heap which are not part of the transactions' read set.

Definition 5.3.18 (Equivalence of program states)**.** *A program state $\mathcal{P}$ is* equivalent *to a program state $\mathcal{P}'$, $\mathcal{P} \simeq \mathcal{P}'$, if for all threads t either $\mathcal{P}(t) = \mathcal{P}'(t)$ or $\mathcal{P}(t) = \mathcal{M}[(m, W, R, i, m', \mathcal{H})]$ and $\mathcal{P}'(t) = \mathcal{M}[(m, W, R, i, m', \mathcal{H}')]$ and $\mathcal{H}|_R = \mathcal{H}'|_R$.*

Definition 5.3.19 (Equivalence of evaluation states)**.** *An evaluation state $\mathcal{H}, \mathcal{P}$ is equivalent to an evaluation state $\mathcal{H}', \mathcal{P}'$ if $\mathcal{H} = \mathcal{H}'$ and $\mathcal{P} \simeq \mathcal{P}'$.*

To obtain equivalent reduction traces with strongly serial effect traces, we apply the reorder algorithm in Algorithm 5.1.

Type-correct programs allow only certain compositions of transactional operations. The non-deterministic scheduler that decides which thread performs the next evaluation step chooses from the thread pool. With exception of the main thread, the spawn effect is emitted when a thread is added to the thread pool. It thus precedes all other effects from this thread.

Transactional effects are only produced when evaluating expressions in the `STM` monad. An $at^{t,i}$ effect is only produced when entering the `STM` monad. All read effects are produced within the `STM` part, and the evaluation of a transactional expression finishes with either an $ab^{t,i}$ or $co^{t,i}(\bar{l})$ effect.

Lemma 5.3.7. *All reduction traces of programs in Λ_{STM} are well-formed.*

Proof of 5.3.7: Let $\mathcal{R}$ be a reduction sequence for a program e:

$$\mathcal{R} = \mathcal{H}_0, \mathcal{P}_0 \stackrel{\alpha_1}{\Longrightarrow} \ldots \stackrel{\alpha_n}{\Longrightarrow} \mathcal{H}_n, \mathcal{P}_n$$

We consider the conditions for well-formedness as given in Definition 5.3.4.

- There is no effect for a thread before its spawn effect, unless it is the main thread:

  Let $\mathcal{H}_k, \mathcal{P}_k \stackrel{\alpha_k}{\Longrightarrow} \mathcal{H}_{k+1}, \mathcal{P}_{k+1}$ be a reduction step in $\mathcal{R}$ with $\alpha_k = \alpha^t$ for $t \neq 0$. By inversion of the evaluation steps $\stackrel{\alpha}{\Longrightarrow}$, the thread identifier t must be in the domain of $\mathcal{P}_k$. The only rule which adds new threads to the thread pool is Spawn. Hence, there exists a step $m < k$ such that $\alpha_m = sp^{t'}(t)$.

- For each transaction, there is an atomic effect in the trace:

  Let $\mathcal{H}_k, \mathcal{P}_k \xRightarrow{\alpha_k} \mathcal{H}_{k+1}, \mathcal{P}_{k+1}$ be a reduction step in $\mathcal{R}$ with $\alpha_k = \alpha^{t,i}$. By inversion of the evaluation rules, either $\alpha_k = at^{t,i}$ and the step produced the atomic effect, or α_k is an empty effect, a read effect, an abort or an commit effect for transaction i in thread t. Inversion of the evaluation rules in these other cases show that the expression evaluated in the thread is a transaction in progress. To start the evaluation of such an expression, the thread has to have performed an Atomic step first. Hence, there exists a step $m \leqslant k$ in $\mathcal{R}$ with $\alpha_m = at^{t,i}$.

- There is no effect for a transaction before its atomic effect:

  Let $\mathcal{H}_k, \mathcal{P}_k \xRightarrow{\alpha_k} \mathcal{H}_{k+1}, \mathcal{P}_{k+1}$ be a reduction step in $\mathcal{R}$ with $\alpha_k = \alpha^{t,i}$. As the transaction counter is incremented when starting a transaction, it holds that for all $\alpha_m = at^{t,j}$ with $k < m$ that $i < j$. Hence, it must hold that $m \leqslant k$.

- There is no effect for a transaction after its commit effect:

  Let $\mathcal{H}_k, \mathcal{P}_k \xRightarrow{\alpha_k} \mathcal{H}_{k+1}, \mathcal{P}_{k+1}$ be a reduction step in $\mathcal{R}$ with $\alpha_k = \alpha^{t,i}$. As the transaction counter is incremented when committing a transaction, it holds that for all $\alpha_m = co^{t,j}(\bar{l})$ with $m < k$ that $i < j$. Hence, it must hold that $k \leqslant m$.

- There is no effect for a transaction after its abort effect:

  This condition is shown analogously to the previous one.

- A transaction may have a commit or an abort effect, but not both:

  Let $\mathcal{H}_k, \mathcal{P}_k \xRightarrow{\alpha_k} \mathcal{H}_{k+1}, \mathcal{P}_{k+1}$ be a reduction step in $\mathcal{R}$ with $\alpha_k = ab^{t,i}$. Assume that there is a reduction step with $\alpha_m = co^{t,j}(\bar{l})$ for $k < m$. As the transaction counter is incremented for each abort, it must hold that $i < j$. The other direction is shown analogously.

- There are no non-transactional effects within a transaction:

  Let $\mathcal{H}_k, \mathcal{P}_k \xRightarrow{\alpha_k} \mathcal{H}_{k+1}, \mathcal{P}_{k+1}$ be a reduction step in $\mathcal{R}$ with $\alpha_k = \epsilon^t t$. Assume that the previous reduction step for thread t is given at $m < k$ with $\alpha_m = \alpha^{t,j}$.

  By inversion of the evaluation rules, non-transactional effects are only given in rules IO-Monad and they require the evaluation of a monadic expression. Again, inversion of rules yields that either $\alpha_m = co^{t,j}(\bar{l})$ or $\alpha_m = ab^{t,j}$.

  The case for $\alpha_k = sp^t(t')$ is shown analogously.

- Transactional effects from the same thread do not interleave:

  Let $\mathcal{H}_k, \mathcal{P}_k \xRightarrow{\alpha_k} \mathcal{H}_{k+1}, \mathcal{P}_{k+1}$ be a reduction step in $\mathcal{R}$ with $\alpha_k = \alpha^{t,i}$. Assume that the previous reduction step for thread t is given at $m < k$ with $\alpha_m = \alpha^{t,j}$ and $i \neq j$, and that there exists a latter reduction step at m' for this thread with $\alpha_{m'} = \alpha^{t,j}$. As the transaction counter is monotonically increasing, it

follows that $j \leqslant i \leqslant j$, and thus it must hold that $j = i$, in contradiction to the assumption.

□

Similar to Algorithm 5.1, we can apply a reordering algorithm on reduction traces. As before, it is based on changing the order of different executions steps by swapping either subsequent steps or moving a non-transactional or atomic step before another atomic step.

When swapping the evaluation steps of different threads, the program state remains equal for each thread in most cases. Effect-free operations ($\alpha_i = \epsilon^t$ or $\alpha_j = \epsilon^{t,i}$) are either pure or work on local (transactional) state. Therefore, these steps can get swapped with any operation while resulting in the same heap and thread pool. Also, reduction steps which result in an abort only modify the transactions' local state. The same holds for read operations.

When swapping a commit effect with an atomic effect, the heap copy taken at the transaction's begin differs for the locations that are updated in the commit. However, the consistency checks for the read accesses in the original trace yielding read effects ensure that the transaction starting with the atomic effect does not read any of the modified data items. Hence, the consistency checks in the trace after swapping the effects also succeed. So, the evaluation on the new heap copy with the committed values is equivalent to the original one. The final state in the new reduction is therefore equivalent, but not necessary equal to final state in the original trace.

When swapping an effect with a spawn effect, the thread pool is extended with the new thread already in the prior state. Otherwise, only thread-local state is changed. For thread spawning and allocation of new heap entries, the deterministic allocation scheme as described in Section 5.2.3 is essential when reordering two spawn steps or two allocation steps.

Finally, the changes in the reduction states for the second kind of trace restructuring can be obtained by handling the multi-step movement of the effect as several individual swaps and accumulating the state modifications.

Theorem 5.3.1 (Opacity)**.** *Let e be a type-correct program in Λ_{STM}. Further, let $\mathcal{R}$ be a sequence of reductions*

$$\mathcal{H}_0, \mathcal{P}_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \mathcal{H}_n, \mathcal{P}_n$$

with $\mathcal{H}_0, \mathcal{P}_0 = \emptyset, \{0 \mapsto e, 0\}$. Then there exists an equivalent sequence $\mathcal{R}'$ of the form

$$\mathcal{H}_0, \mathcal{P}_0 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_n} \mathcal{H}'_n, \mathcal{P}'_n$$

such that $\bar{\alpha}(\mathcal{R}')$ is strongly serial and $\mathcal{H}_n, \mathcal{P}_n$ is equivalent to $\mathcal{H}'_n, \mathcal{P}'_n$.

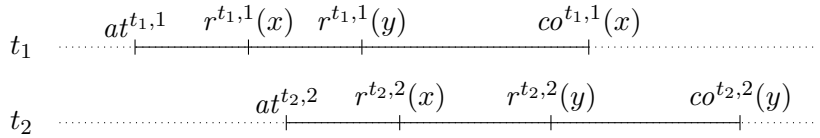
5.4. Snapshot Isolation

The semantics of serializability is easy to reason about, but it is rather restrictive with respect to the set of valid schedules. A less restrictive scheme is obtained by confining the check for intermediate updates at commit time to the write set of the transaction. Such a system is said to exhibit *snapshot isolation* semantics, a popular concurrency control notion in data bases [8]. The basic idea behind snapshot isolation is that each transaction is guaranteed to work on a consistent memory snapshot in isolation from each other, and that there are no lost updates. In the context of STM, snapshot isolation can be used to implement data structures where operations are checked for conflicts on a higher level. A typical use case are container data structures like lists or trees where insertions of different elements commute on the level of semantics, but the implementation yields non-commuting memory access patterns (see also Section 3.3.2).

Definition 5.4.1 (Consistent snapshot)**.** *A transaction operates on a* consistent memory snapshot *if there is no update of a variable between the begin of the transaction and a read effect of this variable in a transaction.*

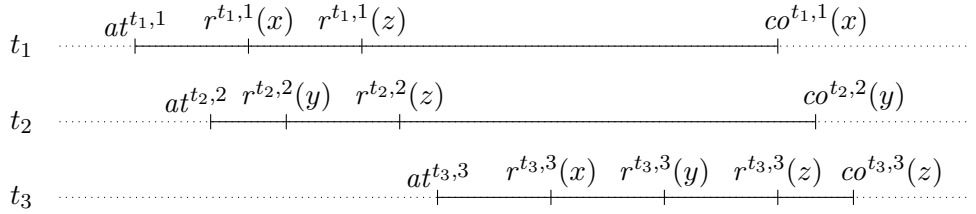
$$\forall r^{t_i,i}(l) \in \bar{\alpha} : \nexists co^{t_j,j}(\bar{l}) \text{ with } \bar{\alpha} \vdash at^{t_i,i} < co^{t_j,j}(\bar{l}) < r^{t_i,i}(l) \text{ and } l \in \bar{l}$$

Serializable traces are trivially also valid in snapshot isolation as read and write effects of transactions are not interleaved. For further examples of snapshot traces consider the following execution trace:



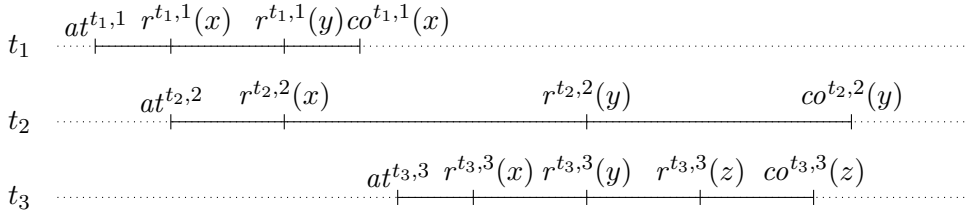
Transaction T_1, running in thread t_1, and transaction T_2, running in thread t_2, operate on the same memory snapshot, and update different memory locations. Yet, there is a read-write dependency from T_1 on T_2 because T_1 is committing x which is then read by T_2. Vice versa, there is also a read-write dependency from T_2 on T_1 due to their operations on y. Therefore, the transactions cannot be serialized by re-ordering their traces. This mutual read-write dependency on transactions is known as write skew anomaly in the literature [8].

To detect such a kind of anomaly, it does not suffice to consider only pairs of transactions in isolation. In the following example, three transactions operate on the same memory snapshot, again committing on different locations.



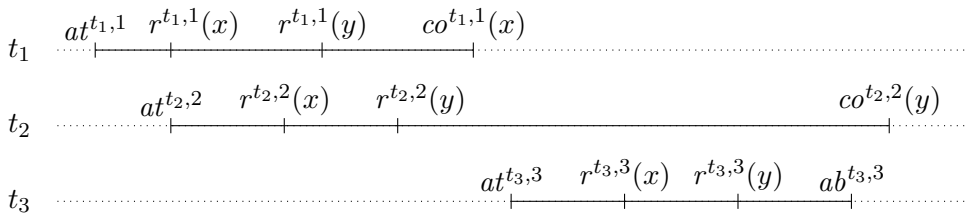
There is no write skew between transactions T_1 and T_2. However, these transactions are related via their read-read dependency on location z which is updated in transaction T_3. T_1 and T_2 are both having a write skew with T_3, T_1 and T_2 cannot be serialized with respect to each other. To reflect this read-read dependency in combination with write-skew, the snapshot relationship must contain the transitive closure of transactions that exhibit a write skew anomaly.

Further, this transitive closure needs to be extended with transactions that only partly share data dependencies, different from write skews. The situation is depicted in this example:



Transaction T_3 can neither be ordered after T_2 because it read location y which is updated by T_2, nor may it be ordered before T_2 due to its write-read dependency on transaction T_1. The underlying reason why the reordering is invalid is given in the write skew of T_1 and T_2. Thus, T_2 and T_3 are also snapshot related.

By definition, only transactions that commit successfully can exhibit write skew anomalies. For opaque systems, aborting transactions that operate on the same memory snapshot as another transaction can be ordered before this transaction. The situation changes when the aborting transaction operates on a different snapshot as this does not allow serializing the aborting transactions with respect to the committing one. Consider the following trace:



As in the previous example, it is neither possible to move transaction T_3 before nor after T_2. This shows that both committing and aborting transactions can be snapshot related.

5.4.1. Operational semantics for Λ_{SI}

Figure 5.8 shows an alternative implementation of Check-Ok and Check-Bad. Replacing the original relations in Figure 5.7 with these, the resulting STM algorithm implements snapshot isolation. We call the language where the semantics is defined by the adapted rules in the following Λ_{SI} to distinguish it from its opaque counterpart Λ_{STM}.

Notice that an alternative semantics of the read access to transactional variables

Figure 5.8. Operational semantics: Heap check for snapshot isolation.

$$\frac{\forall l \in \mathrm{dom}(R) \cap \mathrm{dom}(W) : R(l) = \mathcal{H}(l)}{check(R, W, \mathcal{H}) = ok}\ \textsc{Check-Ok}$$

$$\frac{\exists l \in \mathrm{dom}(R) \cap \mathrm{dom}(W) : R(l) \neq \mathcal{H}(l)}{check(R, W, \mathcal{H}) = bad}\ \textsc{Check-Bad}$$

for snapshot isolation would be to return the entry in the transaction local heap copy without checking for the current value in the heap. This construction also provides a consistent memory snapshot for transactional execution. In practice, this can be achieved for example by keeping multiple versions of each variable. To keep the formalizations of the different isolation levels comparable, we refrain here from modeling such a multi-versioning scheme.

5.4.2. Snapshot isolation for Λ_{SI}

We now formally define the notion of snapshot isolation in terms of traces.

Definition 5.4.2 (Snapshot isolation)**.** *A transactional system implements* snapshot isolation *if in its traces*

1. *all reads are repeatable,*
2. *there are no read skews, and*
3. *there are no lost updates.*

Lemma 5.4.1. *All reads in Λ_{SI} are repeatable.*

Proof of 5.4.1: In Λ_{SI}, as in Λ_{STM}, each read location and value is registered in the read set after the first global read access (Read). All subsequent read access either retrieve the value from the read set (ReadRSet) or the write set (ReadWSet). So, any effect trace contains at most one read effect for a location in a transaction, and non-repeatable reads are not possible. □

Lemma 5.4.2. *There is no read skew in Λ_{SI}.*

Proof of 5.4.2: In the Read rule, there is a test for intermediate updates to the heap. The value in the global heap can only have changed when another transaction committed to the location. In this case, the transaction aborts. Therefore, the global read only succeeds if there is no read skew possible. □

Lemma 5.4.3. *There are no lost updates in* Λ_{SI}.

Proof of 5.4.3: In COMMIT, there is a check for intermediate updates to the heap:

$$\forall l \in \mathrm{dom}(R) \cap \mathrm{dom}(W) : R(l) = \mathcal{H}(l)$$

This check fails if another transaction committed to the location between the global read of the value that added it to the read set and the commit. Therefore, the commit only succeeds if there are no lost updates possible. □

Theorem 5.4.1 (Snapshot isolation for Λ_{SI})**.** *The formal system* Λ_{SI} *implements snapshot isolation.*

Proof of 5.4.1: The proof follows immediately from Lemmas 5.4.1, 5.4.2, and 5.4.3. □

5.4.3. Snapshot traces

In Section 5.3, we have shown how to transform serializable traces into a canonical form, namely serial traces. For traces from systems with snapshot isolation, we now introduce also a canonical form which we call *snapshot trace*.

The main difference between serializability (or opacity) and snapshot isolation is the presence of write skews. Starting from their definition, we derive several criteria that prevent the serialization of a snapshot trace.

By Definition 5.3.13, two transactions T_i and T_j, $i \neq j$, exhibit a *write skew* anomaly in a trace $\bar{\alpha}$ if there exist locations x and y, $x \neq y$ such that

$$r^{t_i,i}(x), r^{t_i,i}(y), r^{t_j,j}(x), r^{t_j,j}(y), co^{t_i,i}(\bar{l}_1), co^{t_j,j}(\bar{l}_2) \in \bar{\alpha}$$

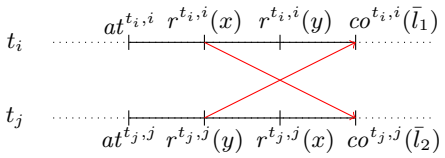
and

$$\bar{\alpha} \vdash at^{t_i,i} < co^{t_j,j}(\bar{l}_2)$$
$$\bar{\alpha} \vdash at^{t_j,j} < co^{t_i,i}(\bar{l}_1)$$

with $x \in \bar{l}_1$ and $y \in \bar{l}_2$.

We abbreviate a write skew as $T_i \sim_{\bar{\alpha}} T_j$.

The general situation is visualized in the following sketch:

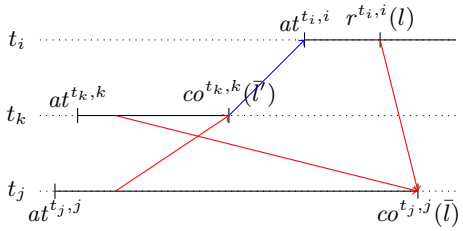


Lemma 5.4.4. *The write skew anomaly* $\sim_{\bar{\alpha}}$ *defines a symmetric relation on the set of transactions in a trace* $\bar{\alpha}$.

Proof. The symmetry follows directly from Definition 5.3.13. □

As shown in the examples in the beginning of this section, transactions exhibiting a write skew anomaly cannot be serialized with respect to each other due to their read-write dependencies. The existence of write skews in a trace can introduce further problems when trying to serialize an execution trace.

Consider the situation in the following sketch assuming that $l \in \bar{l}$:

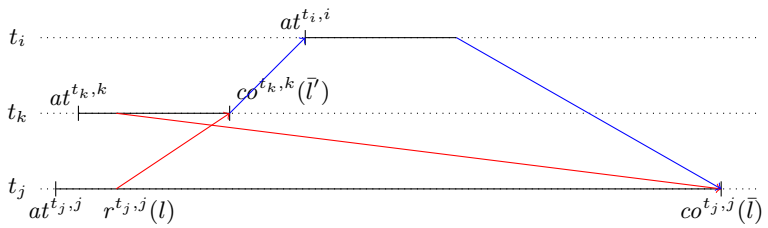


Even though transaction T_i does not have a write skew with transaction T_j, it cannot be ordered before it as depends on another transaction T_k (indicated by the blue arrow) which has a write skew with T_j. Due to its read-write dependency with T_j, it also cannot be ordered after T_j.

Definition 5.4.3 (Snapshot connected)**.** *A transaction T_i is* snapshot connected *to another transaction T_j,$T_i \ltimes T_j$, if*

1. *$T_i \sim T_j$, or*
2. *$T_i \triangleright T_j$ and there exists a transaction T_k such that $T_k \triangleright T_i$ and $T_k \ltimes T_j$.*

The situation is depicted in the following sketch:



Lemma 5.4.5. *Transactions that are snapshot connected cannot be serialized.*

Proof of 5.4.5: We consider the two possible cases from the definition.

- If $T_i \sim T_j$, the dependency $r^{tj,j}(l) \triangleright_d co^{t_j,j}(\bar{l})$ prohibits to order T_i before T_j. Serializing T_j before T_i is not possible by the symmetric argumentation.
- By definition, there is a transaction T_k with $T_k \ltimes T_j$, and $T_k \triangleright T_i$. Applying this lemma inductively, T_k cannot be ordered before T_j. The dependency $T_k \triangleright T_i$ does not allow to serialize T_i before T_k. Therefore, it is not possible to order T_i before T_j.

  Also, transaction T_i cannot be ordered after T_j because of T_i depends on T_j.

□

As serial traces serve as a kind of normal form for serializable traces, we next introduce the notion of a normal form for traces of systems implementing snapshot isolation. To this end we require the effects of one transaction grouped together as closely as possible.

Definition 5.4.4 (Snapshot traces)**.** *A well-formed trace $\bar{\alpha}$ is a* snapshot trace *if for each transaction T_j in $\bar{\alpha}$ it holds that*

1. *$at^{t_j,j}$ is followed directly by all effects of T_j, or*
2. *the $at^{t_j,j}$ is followed by all read and empty effects of T_j and there exists an transaction T_i with $\bar{\alpha} \vdash at^{t_j,j} < at^{t_i,i} < co^{t_j,j}(\bar{l})$ such that T_i is snapshot connected to T_j and if $co^{t_i,i}(\bar{l}') \in \bar{\alpha}$, then $\bar{l} \cap \bar{l}' = \emptyset$.*

The definition of snapshot trace ensures that all transactions are serialized with respect to each other unless they are snapshot connected. In particular, it disallows any non-transactional effect between the atomic and any following read or empty effects of a transaction. In the case of snapshot connection between transactions, only the final effect of a transaction, i.e. the commit or abort effect, may be separated from its other effects. The last condition about the committed modification ensures that there are no lost updates as the snapshot connected transactions are only allowed to update different locations.

Obviously, serialized traces are snapshot traces as there is no interleaving of transactional effects from different threads possible.

Corollary 5.4.1. *A serialized trace is a snapshot trace.*

Lemma 5.4.6. *A system yielding snapshot traces implements snapshot isolation.*

Proof of 5.4.6: By definition, snapshot traces do not allow lost updates. Further, it holds that no effect from another transaction is allowed between the reads of each transaction. Hence, neither read skew nor non-repeatable reads are possible. □

To show that traces produced by well-typed programs in Λ_{SI} are snapshot traces, we follow a similar path as in the proof for opaqueness of Λ_{STM}. As for opaque traces, it is possible to reorder traces under certain conditions to obtain equivalent snapshot traces.

The reordering algorithm incrementally derives a relation $\fallingdotseq$ of snapshot connected transactions by adding pair of snapshot connected transactions to it. The following formal work is parameterized by this relation, and can be applied to any subset of $\fallingdotseq$ (e.g. thus during the construction of $\fallingdotseq$).

Definition 5.4.5 (Snapshot admissible)**.** *Given a relation $\fallingdotseq$ between transactions, a well-formed trace $\bar{\alpha}$ is* snapshot admissible under $\fallingdotseq$ *if*

1. *$\bar{\alpha}$ is a snapshot trace, and*

2. *if transaction T_i is snapshot connected with T_j, then $T_i \fallingdotseq T_j$.*

Lemma 5.4.7 (Conflicts)**.** *A trace $\bar{\alpha}$ taken from an execution of a program in Λ_{SI} is either snapshot admissible under some $\fallingdotseq$, or there is a prefix $\bar{\alpha}'$ such that $\bar{\alpha}' = \alpha_1 \dots \alpha_k$ is snapshot admissible under $\fallingdotseq$ and*

- *α_k and α_{k+1} are independent, or*
- *$\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = co^{t_j,j}(\bar{l})$ with $l \in \bar{l}$.*

Proof of 5.4.7: We consider all possible combinations of effects which might occur in a well-formed trace in Λ_{SI}.

Clearly, all traces of length one are snapshot admissible. Consider now all traces containing more than two effects and let $\bar{\alpha}' = \alpha_1 \dots \alpha_k$ be the snapshot admissible prefix under some $\fallingdotseq$.
Case distinction on α_k and α_{k+1} where $i \neq j$.

- *Case* $\alpha_k = \epsilon^{t_i,}$ or $\alpha_{k+1} = \epsilon^{t_j,}$: snapshot admissible or independent.
- *Case* $\alpha_k = \epsilon^{t_i,j}$ or $\alpha_{k+1} = \epsilon^{t_j,j}$: snapshot admissible or independent.
- *Case* $\alpha_k = at^{t_i,i}$ and $\alpha_{k+1} = at^{t_j,j}$: snapshot admissible.
- *Case* $\alpha_k = at^{t_i,i}$ and $\alpha_{k+1} = r^{t_i,i}(l)$: snapshot admissible.
- *Case* $\alpha_k = at^{t_i,i}$ and $\alpha_{k+1} = r^{t_j,j}(l)$: independent.
- *Case* $\alpha_k = at^{t_i,i}$ and $\alpha_{k+1} = co^{t_i,i}(\bar{l})$: snapshot admissible.
- *Case* $\alpha_k = at^{t_i,i}$ and $\alpha_{k+1} = co^{t_j,j}(\bar{l})$: independent.
- *Case* $\alpha_k = at^{t_i,i}$ and $\alpha_{k+1} = ab^{t_i,i}$: snapshot admissible.
- *Case* $\alpha_k = at^{t_i,i}$ and $\alpha_{k+1} = ab^{t_j,j}$: independent.
- *Case* $\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = r^{t_j,j}(l')$: independent.
- *Case* $\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = r^{t_i,i}(l')$: snapshot admissible.
- *Case* $\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = co^{t_i,i}(\bar{l})$: snapshot admissible.
- *Case* $\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = co^{t_j,j}(\bar{l})$: If $l \in \bar{l}$, then this is the second case in the lemma. Otherwise independent.
- *Case* $\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = ab^{t_i,i}$: snapshot admissible.
- *Case* $\alpha_k = r^{t_i,i}(l)$ and $\alpha_{k+1} = ab^{t_j,j}$: independent.

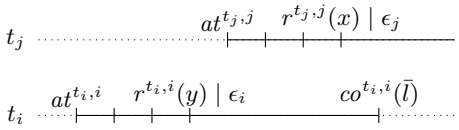
- *Case* $\alpha_k = co^{t_i,i}(\bar{l})$ and $\alpha_{k+1} = co^{t_j,j}(\bar{l}')$: According to the operational semantics, it must hold that $\bar{l} \cap \bar{l}' = \emptyset$. Therefore, the effects are independent.

- *Case* $\alpha_k = co^{t_i,i}(\bar{l})$ and $\alpha_{k+1} = ab^{t_j,j}$: independent.

End case distinction on α_k and α_{k+1} where $i \neq j$. Again, cases that are left out violate the well-formedness criterion. □

Lemma 5.4.8 (Permutation of independent transactions)**.** *Let* $\bar{\alpha}$ *be a well-formed trace with* $\bar{\alpha} = \bar{\alpha}' co^{t_i,i}(\bar{l})$ *and* $\bar{\alpha}'$ *is snapshot admissible under some* $\leftrightarrows$*. Further, let* T_j *be a transaction with* $\bar{\alpha} \vdash at^{t_i,i} < at^{t_j,j} < co^{t_i,i}(\bar{l})$ *and there* $\nexists k$ *with* $\bar{\alpha} \vdash at^{t_i,i} < at^{t_k,k} < at^{t_j,j}$*.*

Then, either T_i *and* T_j *have a write skew, or trace* $\bar{\alpha}$ *is equivalent to a trace* $\bar{\beta}$ *with* $\bar{\beta} \vdash \alpha^{t_j,j} < at^{t_i,i}$ *for all effects* $\alpha^{t_j,j}$ *of transaction* T_j*.*



Proof of 5.4.8: There are no dependencies between $at^{t_j,j}$ and any $r^{t_i,i}(l)$, nor $at^{t_j,j}$ and $at^{t_i,i}$, nor any $r^{t_i,i}(l)$ and any $r^{t_j,j}(l)$.

If $co^{t_i,i}(\bar{l}) \in \bar{\alpha}$ and there is a dependency of a $r^{t_i,i}(l)$ to $co^{t_i,i}(\bar{l})$, then the transactions have a write skew. Otherwise, T_j does not depend on T_i and all effects may be reordered because they are independent. □

Algorithm 5.2 defines a reordering algorithm to transform traces into the canonical snapshot form. To provide some intuition on the permutations done in Algorithm 5.2, we consider the individual steps that are performed on the following trace.

$$at^1\ at^2\ r^1(x)\ r^1(y)\ r^2(x)\ r^2(y)\ at^3\ r^3(z)\ at^4\ r^4(z)\ r^4(x)\ co^1(x)\ co^2(y)\ ab^4\ co^3(z)$$

For better readability, we assume that each transaction is run in a separate thread and omit the thread identifier.

Starting with an empty relation $\leftrightarrows$ of snapshot connected transactions, the algorithm detects the prefix $at^1\ at^2\ r^1(x)$ to be not admissible and reorders the independent effects at^2 and $r^1(x)$. Similarly, the other read effects of T_1 are moved towards at^1, yielding

$$at^1\ r^1(x)\ r^1(y)\ at^2\ r^2(x)\ r^2(y)\ at^3\ r^3(z)\ at^4\ r^4(z)\ r^4(x)\ co^1(x)\ co^2(y)\ ab^4\ co^3(z)$$

The admissible prefix ends here with $r^4(x)$ which is followed by $co^1(x)$. The atomic effect between at^4 and at^1 which is closest to at^4 is at^3. Because T_4 does

Algorithm 5.2 Reordering transactions for snapshot traces.

```
≎ ← ∅
while ᾱ is not snapshot admissible under ≎ do
    choose α_k and α_{k+1} such that α_1 ... α_k is snapshot admissible under ≎
        and α_1 ... α_{k+1} is not
    if α_k is not in conflict with α_{k+1} then
        swap α_k with α_{k+1}
    else if α_k = r^{t_i,i}(l) and α_{k+1} = co^{t_j,j}(l̄) then
        TxnPerm(i,j)
    else
        signal error
    end if
end while

method TxnPerm(i,j)
    if ∄ at^{t_k,k} : ᾱ ⊢ at^{t_j,j} < at^{t_k,k} < at^{t_i,i} then
        if write skew between T_i and T_j and not T_i ≎ T_j then
            add (T_i, T_j) to ≎
        else
            move at^{t_i,i} before at^{t_j,j}
        end if
    else
        take at^{t_n,n} such that ∄ at^{t_k,k} : ᾱ ⊢ at^{t_n,n} < at^{t_k,k} < at^{t_i,i}
        if T_n ▷ T_i then
            if T_n ≎ T_j then
                add (T_i, T_j) to ≎
            else
                TxnPerm(n,j)
            end if
        else
            move at^{t_i,i} before at^{t_n,n}
        end if
    end if
end
```

not depend on T_3, the effect at^4 is moved before at^3. With the following recursive calls to TxnPerm, at^4 finally ends up at the head of the trace.

$$at^4\ at^1\ r^1(x)\ r^1(y)\ at^2\ r^2(x)\ r^2(y)\ at^3\ r^3(z)\ r^4(z)\ r^4(x)\ co^1(x)\ co^2(y)\ ab^4\ co^3(z)$$

The next steps move again the read effects of T_4 immediately behind at^4, such that the trace is then in this order:

$$at^4\ r^4(z)\ r^4(x)\ at^1\ r^1(x)\ r^1(y)\ at^2\ r^2(x)\ r^2(y)\ at^3\ r^3(z)\ co^1(x)\ co^2(y)\ ab^4\ co^3(z)$$

Now, the commit of T_1 is permuted with the preceding effects resulting in the trace

$$at^4\ r^4(z)\ r^4(x)\ at^1\ r^1(x)\ r^1(y)\ at^2\ r^2(x)\ co^1(x)\ r^2(y)\ at^3\ r^3(z)\ co^2(y)\ ab^4\ co^3(z)$$

with admissible prefix

$$at^4\ r^4(z)\ r^4(x)\ at^1\ r^1(x)\ r^1(y)\ at^2\ r^2(x).$$

At this point, the algorithm detects the write skew between T_1 and T_2 and adds $(1, 2)$ to $\div$. The prefix

$$at^4\ r^4(z)\ r^4(x)\ at^1\ r^1(x)\ r^1(y)\ at^2\ r^2(x)\ co^1(x)$$

is admissible under the extended $\div$, and in the next iteration $r^2(y)$ is again moved before $co^1(x)$.

Finally, swapping the commits and aborts at the end of the trace towards the other effects of the transaction to which they belong, the trace is snapshot admissible:

$$at^4\ r^4(z)\ r^4(x)\ ab^4\ at^1\ r^1(x)\ r^1(y)\ at^2\ r^2(x)\ r^2(y)\ co^1(x)\ co^2(y)\ at^3\ r^3(z)\ co^3(z)$$

The algorithm in Figure 5.2 has the following properties:

1. It terminates on all traces that are produced by well-typed programs in Λ_{SI}.
2. For any trace input from a well-typed program in Λ_{SI}, it yields an equivalent trace which is snapshot admissible.

Lemma 5.4.9 (Termination)**.** *The algorithm terminates on all traces of type-correct programs in Λ_{SI} without an error.*

Proof of 5.4.9: By Lemma 5.4.7, the program never reaches the case for signaling the error.

For proving termination, we define a cost function which closely resembles the cost function in the proof of Lemma 5.3.3. Again, the total cost of a trace is given by pairing the cost for moving atomic effects and swapping neighboring effects:

$$\mathtt{cost}_{SI}(\bar{\alpha}) = \big(\mathtt{cost}_{\mathtt{permSI}}(\bar{\alpha}), \mathtt{cost}_{\mathtt{swap}}(\bar{\alpha})\big)$$

and we define a total order on these trace costs as:

$$\texttt{cost}_{SI}(\bar{\alpha}_1) < \texttt{cost}_{SI}(\bar{\alpha}_2)$$

iff $\texttt{cost}_{\texttt{permSI}}(\bar{\alpha}_1) < \texttt{cost}_{\texttt{permSI}}(\bar{\alpha}_2)$, or

$$\texttt{cost}_{\texttt{permSI}}(\bar{\alpha}_1) = \texttt{cost}_{\texttt{permSI}}(\bar{\alpha}_2) \wedge \texttt{cost}_{\texttt{swap}}(\bar{\alpha}_1) < \texttt{cost}_{\texttt{swap}}(\bar{\alpha}_2).$$

The $\texttt{cost}_{\texttt{swap}}(\bar{\alpha})$ measure is defined as for the proof of Lemma 5.3.3.

For $\texttt{cost}_{\texttt{permSI}}(\bar{\alpha})$ we use an adapted version of the $\texttt{cost}_{\texttt{perm}}(\bar{\alpha})$:

$$\texttt{cost}_{\texttt{permSI}}(\bar{\alpha}) = \big|\{\, at^{t_i,i} \mid \exists \alpha^{t'}, t' \neq t \text{ with } \bar{\alpha} \vdash at^{t_i,i} < \alpha^{t'} < co^{t_i,i}(\bar{l}) \text{ and } (T_i, T_{t'}) \notin \fallingdotseq \}\big|$$

The cost measure $\texttt{cost}_{\texttt{permSI}}()$ tracks the number of transactions that are not serialized with respect to another transaction, taking out those transactions that are snapshot-connected as they cannot be serialized.

In each iteration of Algorithm 5.2, either two effects are swapped (resolving in a decrease of $\texttt{cost}_{\texttt{swap}}()$), or the subroutine TxnPerm is called.

Following the control flow of TxnPerm, it holds that then either a pair of transactions is added to $\fallingdotseq$ or atomic effects of transactions are permuted, or the subroutine is called recursively. The depth of recursion is delimited by the length of the trace section between $at^{t_j,j}$ and $at^{t_i,i}$, so that each call to TxnPerm terminates after a finite number of recursive calls. Hence, each call to TxnPerm from the while loop reduces $\texttt{cost}_{\texttt{permSI}}()$.

Combining the costs, each iteration of the while loop reduces the total cost $\texttt{cost}_{SI}(\bar{\alpha})$, and thus the algorithm terminates after a finite number of iteration steps.

□

Lemma 5.4.10 (Permutation)**.** *The output of the algorithm is a permutation of the input trace.*

Proof of 5.4.10: All operations on the trace only permute the effects, but do not change, add or remove elements. □

Lemma 5.4.11 (Dependencies)**.** *The algorithm does not change any dependencies of the effects in the trace.*

Proof of 5.4.11: Effects are only swapped when they are independent or when permuting transactions. In the latter case, the dependencies in the trace are respected as is shown in Lemma 5.4.8. □

Theorem 5.4.2 (Snapshot traces for Λ_{SI})**.** *Let $\mathcal{P}_0$ be a type-correct program in Λ_{SI}. Further, let $\mathcal{R}$ be a sequence of reductions*

$$\mathcal{H}_0, \mathcal{P}_0 \stackrel{\alpha_1}{\Longrightarrow} \ldots \stackrel{\alpha_n}{\Longrightarrow} \mathcal{H}_n, \mathcal{P}_n.$$

Then, there exists an equivalent sequence $\mathcal{R}'$ of the form

$$\mathcal{H}_0, \mathcal{P}_0 \stackrel{\alpha'_1}{\Longrightarrow} \ldots \stackrel{\alpha'_n}{\Longrightarrow} \mathcal{H}'_n, \mathcal{P}'_n$$

such that $\bar{\alpha}(\mathcal{R}')$ is snapshot admissible and $\mathcal{H}_n, \mathcal{P}_n$ is equivalent to $\mathcal{H}'_n, \mathcal{P}'_n$.

The proof of Theorem 5.4.2 follows the same reasoning as in Section 5.3.4: We apply the algorithm for reordering traces into a snapshot admissible form to the traces of $\mathcal{R}$. Because the algorithm only requires the permutation of independent effects, the preceding lemmata yield that the result is an equivalent reduction sequence with a snapshot trace trace.

In the same way that serial traces allow easier reasoning about transactional executions when compared to serializable traces, snapshot traces are a simpler canonical form for snapshot traces. Reducing the number of possible interleavings for effects aids programmers in reasoning about the interaction of concurrently running transactions.

5.5. Formalization of Twilight

This chapter closes with a formalization of the Twilight STM. In short, Twilight STM splits the code of a transaction into a (functional) atomic phase, which behaves as in Λ_{STM}, and an (imperative) twilight phase. Code in the twilight phase executes before the decision about a transaction's fate (restart or commit) is made and can affect its outcome based on the actual state of the execution environment. The Twilight API has operations to detect and repair read inconsistencies as well as operations to overwrite previously written variables. It also permits the embedding of I/O operations in such a way that the I/O operation is executed exactly once.

In the actual implementation, twilight code may run concurrently with other transactions including their twilight code. Related work on formalizations of transactional memory ([1], [54]) require transactions to be executed in a sequential fashion, thus following the proposal of Single Global Lock semantics for transactions [53].

The formalization of Twilight STM, called Λ_{TWI}, admits an interleaved execution of concurrently running threads and transactions, similar to Λ_{STM} and Λ_{SI}. It restricts the possible interleaving of threads in such a way that the twilight code of each transaction runs solo, i.e., all other threads are stalled while a transaction executes its twilight code. It is therefore possible for a transaction to observe updates by other transactions when reaching the twilight zone.

Figure 5.9. Syntax of Λ_{TWI}. Expressions marked in gray arise only during evaluation.

$$
\begin{array}{lll}
x \in \mathrm{Var} & & l \in \mathrm{Ref} \\
v \in \mathrm{Val} & ::= & l \mid \texttt{tt} \mid \texttt{ff} \mid () \mid \lambda x.e \mid \texttt{return}\ e \mid \texttt{error} \\
e \in \mathrm{Exp} & ::= & v \mid x \mid e\ e \mid \texttt{if}\ e\ e\ e \\
 & & \mid \texttt{spawn}\ e \mid \texttt{atomic}\ e \mid e \gg\!= e \mid e \ggg e \\
 & & \mid (e, W, R, i, e, \mathcal{H}) \mid (e, W, R, i, e, f) \\
 & & \mid \texttt{new}\ e \mid \texttt{read}\ e \mid \texttt{write}\ e\ e \\
 & & \mid \texttt{update}\ e\ e \mid \texttt{reread}\ e \mid \texttt{inconsistent}\ e \\
 & & \mid \texttt{reload} \mid \texttt{ignoreUpdates} \mid \texttt{IOtoSTM}\ e \mid \texttt{retry}
\end{array}
$$

5.5.1. Syntax

Figure 5.9 presents the syntax of Λ_{TWI}. In addition to the standard operations that were described in Section 5.2, there is now a special bind operator $\ggg$ for entering the twilight zone. The `error` value indicates that a thread is stuck in an erroneous state.

The extended syntax of Λ_{TWI}provides also repair operations for modifying the heap in the twilight zone. Variables that have been read or modified in the body of the transaction can be modified via `update` . `reread` yields the value that a variable is currently associated with a location in the read set. The operation `inconsistent` compares the state of the transaction in the read set with its counterpart in the global heap. A consistent snapshot of the read set with the values that are currently in the heap can be obtained with `reload`. The `ignoreUpdates` operator allows a transaction to disregard updates by other transactions during conflict detection. With `IOtoSTM` e, an irrevocable expression e can be embedded into the twilight zone. Finally, the `retry` method issues a restart of the transaction.

As in the type system for Λ_{STM}, Σ tracks the type of memory locations, and Γ tracks the type of variables. Figure 5.10 shows only the rules that differ from the ones in Figure 5.2. The type system now comprises two other kinds of monad, the `TWI` and the `TXN` monad. The expressions that are evaluated as transactions are now all of monadic type `TXN` (see rule T-Atomic). An instance of the `TXN` monad consists of a transactional body from the `STM` monad and twilight code from the `TWI` monad. The rule T-TwiBind deals with the switch from a transaction's body to the associated twilight zone. Expressions of type `TWI` τ may only be used within the twilight code of a transaction as they require special concurrency guarantees.

5.5.2. Operational Semantics for Λ_{TWI}

Figure 5.11 introduces state relations for Λ_{TWI}.

A transaction T_i is a tuple $(e, W, R, i, e, \mathcal{H})$. As before, it consists of the expression that is currently evaluated, the write set, and the read set of the transaction, a (unique) transaction identifier, a copy of the whole expression that is to be eval-

Figure 5.10. Extension of typing rules of Λ_{TWI}.

$$\text{Types:}\quad \tau ::= \texttt{bool} \mid () \mid \texttt{R}\,\tau \mid \tau \to \tau \mid \mu\,\tau \qquad \mu ::= \texttt{IO} \mid \texttt{TXN} \mid \texttt{STM} \mid \texttt{TWI}$$

$$\frac{}{\Sigma|\Gamma \vdash \texttt{error} : \tau}\ \text{T-Error} \qquad \frac{\Sigma|\Gamma \vdash e : \texttt{TXN}\,\tau}{\Sigma|\Gamma \vdash \texttt{atomic}\ e : \texttt{IO}\,\tau}\ \text{T-Atomic}$$

$$\frac{\Sigma|\Gamma \vdash e_1 : \texttt{STM}\,\tau \qquad \Sigma|\Gamma \vdash e_2 : \texttt{bool} \to \tau \to \texttt{TWI}\,\tau'}{\Sigma|\Gamma \vdash e_1 \ggg e_2 : \texttt{TXN}\,\tau'}\ \text{T-TwiBind}$$

$$\frac{\Sigma|\Gamma \vdash e : \texttt{TXN}\,\tau \qquad \Sigma|\Gamma \vdash e' : \texttt{TXN}\,\tau \qquad \Sigma \vdash W \qquad \Sigma \vdash R \qquad \Sigma \vdash \mathcal{H}}{\Sigma|\Gamma \vdash (e, W, R, i, e', \mathcal{H}) : \texttt{IO}\,\tau}\ \text{T-Txn}$$

$$\frac{\Sigma|\Gamma \vdash e : \texttt{TWI}\,\tau \qquad \Sigma|\Gamma \vdash e' : \texttt{TXN}\,\tau \qquad \Sigma \vdash W \qquad \Sigma \vdash R}{\Sigma|\Gamma \vdash (e, W, R, i, e', f) : \texttt{IO}\,\tau}\ \text{T-TwiTxn}$$

$$\frac{\Sigma|\Gamma \vdash e_1 : \texttt{R}\,\tau \qquad \Sigma|\Gamma \vdash e_2 : \tau}{\Sigma|\Gamma \vdash \texttt{update}\ e_1\ e_2 : \texttt{TWI}\,()}\ \text{T-Update} \qquad \frac{\Sigma|\Gamma \vdash e : \texttt{R}\,\tau}{\Sigma|\Gamma \vdash \texttt{reread}\ e : \texttt{TWI}\,\tau}\ \text{T-Reread}$$

$$\frac{\Sigma|\Gamma \vdash e : \texttt{R}\,\tau}{\Sigma|\Gamma \vdash \texttt{inconsistent}\ e : \texttt{TWI}\,\texttt{bool}}\ \text{T-Incons}$$

$$\frac{}{\Sigma|\Gamma \vdash \texttt{reload} : \texttt{TWI}\,()}\ \text{T-Reload}$$

$$\frac{}{\Sigma|\Gamma \vdash \texttt{ignoreUpdates} : \texttt{TWI}\,()}\ \text{T-IgnoreUpdates}$$

$$\frac{}{\Sigma|\Gamma \vdash \texttt{retry} : \texttt{TWI}\,\tau}\ \text{T-Retry} \qquad \frac{\Sigma|\Gamma \vdash e : \texttt{IO}\,\tau}{\Sigma|\Gamma \vdash \texttt{IOtoSTM}\ e : \texttt{TWI}\,\tau}\ \text{T-Safe}$$

Figure 5.11. Operational semantics of Λ_{TWI}: State related definitions.

$$\begin{aligned}
T &\in \text{Txn} &&= \text{Exp} \times \text{Store} \times \text{Store} \times \text{TxnId} \times \text{Exp} \times \text{Store}\\
T' &\in \text{TwiTxn} &&= \text{Exp} \times \text{Store} \times \text{Store} \times \text{TxnId} \times \text{Exp} \times \text{Flag}\\
f &\in \text{Flag} &&= \{ok, bad\}\\
\alpha_i &\in \text{TxnEffect} &&= \cdots \cup \{\alpha^{t,i} \mid \alpha \in \text{Effect}\}\\
s &\in \text{State} &&= \text{Heap} \times \text{Program} \times \text{ThreadId}
\end{aligned}$$

Figure 5.12. Operational semantics of Λ_{TWI}: Evaluation contexts and local evaluation steps.

Evaluation contexts:

$$\begin{array}{lcl} \mathcal{E} & ::= & []\ e \mid \texttt{if}\ []\ e\ e' \\ \mathcal{M} & ::= & \texttt{new}\ e \mid \texttt{read}\ [] \mid \texttt{write}\ []\ e \\ & \mid & \texttt{reread}\ [] \mid \texttt{update}\ []\ e \mid \texttt{inconsistent}\ [] \mid [] \gg\!= e \mid [] \gg\!\gg\!= e \end{array}$$

Expression evaluation $\rightarrow$:

$$\begin{array}{rcl} (\lambda x.e)\ e' & \rightarrow & e[e'/x] \\ \texttt{if tt}\ e\ e' & \rightarrow & e \\ \texttt{if ff}\ e\ e' & \rightarrow & e' \end{array}$$

$$\frac{e \rightarrow e'}{\mathcal{E}[e] \rightarrow \mathcal{E}[e']}$$

Monadic evaluation $\curvearrowright$:

$$\begin{array}{lcl} \texttt{return}\ e \gg\!= e' & \curvearrowright & e'\ e \\ \texttt{error} \gg\!= e & \curvearrowright & \texttt{error} \end{array}$$

$$\frac{e \rightarrow e'}{e \curvearrowright e'} \qquad \frac{m \curvearrowright m'}{\mathcal{M}[m] \curvearrowright \mathcal{M}[m']}$$

uated transactionally for rollbacks, a copy of the heap taken at the beginning of the transaction or during a reload. Now, additionally, the Λ_{TWI} calculus requires another kind of transaction tuple which does not contain a heap copy, but instead a flag denoting the transaction's status. An *ok* flag indicates that a transaction's read set variables are consistent with the current heap, a *bad* flag denotes some inconsistency between the transaction's read set and the current heap. The set of effects is extended with non-transactional effects that are embedded into a transactional effect. An example for such an effect is $(sp^t(t'))^t_i$ denoting the the spawning of a thread in an embedded I/O monad.

An execution state consists of a heap, a thread pool with expressions that are concurrently evaluated, and a thread identifier to denote the thread that is currently executing the twilight code of a transaction. If there is no such thread, we indicate it with $-$. The evaluation of a program starts in an initial configuration $\langle\rangle, \{0 \mapsto e\}, -$ with an empty heap, a main thread t_0, and no twilight thread identifier set. A final configuration has the form $\mathcal{H}, \{0 \mapsto v_0, \ldots, t_n \mapsto v_n\}, -$. In contrast to the operational semantics of Λ_{STM}, an evaluation step in Λ_{TWI} can produce more than one effect (e.g. ReloadBad).

The rules in Figures 5.12-5.19 define the semantics of the language constructs.

Figure 5.13. Operational semantics of Λ_{TWI}: Global evaluation steps.

$$\frac{m \curvearrowright m'}{\mathcal{H}, \mathcal{P}\{t \mapsto m\}, - \xRightarrow{\epsilon^t} \mathcal{H}, \mathcal{P}\{t \mapsto m'\}, -} \text{IO-Monad}$$

$$\frac{t' \text{ fresh}}{\mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\}], -\texttt{spawn}\ m \xRightarrow{sp^t(t')} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{return}\ ()], t' \mapsto m\}, -} \text{Spawn}$$

$$\frac{\mathcal{H}, m \overset{\alpha}{\rightarrowtail} \mathcal{H}', m', s}{\mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[m]\}, - \xRightarrow{\alpha} \mathcal{H}', \mathcal{P}[t \mapsto \mathcal{M}[m']], s} \text{Txn}$$

$$\frac{\mathcal{H}, m \overset{\alpha}{\twoheadrightarrow} \mathcal{H}', m', s}{\mathcal{H}, \mathcal{P}[t \mapsto \mathcal{M}[m]], t \xRightarrow{\alpha} \mathcal{H}', \mathcal{P}[t \mapsto \mathcal{M}[m']], s} \text{TxnTwi}$$

In Figure 5.12, the standard rules for the call-by-name core calculus with monads are defined. $\mathcal{E}[\bullet]$ and $\mathcal{M}[\bullet]$ denote the evaluation context for expressions and monadic expressions, respectively. As an additional rule, the error statement is passed through the monad without further evaluation of statements.

The evaluation rules for the IO monad and the transaction body in Figures 5.13 and 5.14 are similar to the operational semantics for Λ_{STM}. They mainly differ with respect to the twilight flag that is recorded in the system's state. Execution steps at top level or within a transaction choose an arbitrary thread (Txn), unless there is a transaction is currently executing its twilight zone. In this case, the corresponding thread is chosen for the next step (TxnTwi). $\rightarrowtail$ denotes the evaluation relation for transactional bodies, $\twoheadrightarrow$ for twilight zones.

To simplify the rules, we refer for the twilight semantics to a further unspecified scheme for generating transaction identifiers instead of passing a transaction identifier i as part of the thread state. Using thread-local counters is one way to implement such a scheme for identifier generation.

Figure 5.15 shows the evaluation of expressions within the twilight zone. Before committing, the transaction must switch from the `STM` monad to the `TWI` monad with the twilight bind $\ggg$. At this point, the heap is checked for updates to the references that are in the transaction's read set. There are two cases:

Rule TwiOk applies if the check is successful, this is, none of the heap locations read by the transaction have been updated by another transaction in the meantime. It sets the twilight flag to *ok*.

Rule TwiBad applies if the check fails. It sets the transaction's twilight flag to *bad*. In the `TWI` monad, the transaction's twilight state can be set to *ok* with

Figure 5.14. Operational semantics of Λ_{TWI}: Transactional body.

$$\frac{i \text{ fresh}}{\mathcal{H}, \texttt{atomic}\ m \overset{at^{t,i}}{\rightarrowtail} \mathcal{H}, (m, \langle\rangle, \langle\rangle, i, m, \mathcal{H}), -} \textsc{Atomic}$$

$$\frac{m \curvearrowright m''}{\mathcal{H}, (m, W, R, i, m', \mathcal{H}') \overset{\epsilon^{t,i}}{\rightarrowtail} \mathcal{H}, (m'', W, R, i, m', \mathcal{H}'), -} \textsc{STM-Monad}$$

$$\frac{W' = W[l \mapsto (e, i)] \qquad l \notin \mathcal{P}, \mathcal{H}}{\mathcal{H}, (\mathcal{M}[\texttt{new}\ e], W, R, i, m', \mathcal{H}') \overset{\epsilon^{t,i}}{\rightarrowtail} \mathcal{H}, (\mathcal{M}[\texttt{return}\ l], W', R, i, m', \mathcal{H}'), -} \textsc{Alloc}$$

$$\frac{W' = W[l \mapsto (e, i)]}{\mathcal{H}, (\mathcal{M}[\,\texttt{write}\, l\, e], W, R, i, m', \mathcal{H}') \overset{\epsilon^{t,i}}{\rightarrowtail} \mathcal{H}, (\mathcal{M}[\texttt{return}\ ()], W', R, i, m', \mathcal{H}'), -} \textsc{Write}$$

$$\frac{R' = R[l \mapsto (e, j)] \qquad l \notin \mathrm{dom}(W) \cup \mathrm{dom}(R) \qquad \mathcal{H}(l) \equiv \mathcal{H}'(l) \equiv (e, j)}{\mathcal{H}, (\mathcal{M}[\texttt{read}\, l], W, R, i, m', \mathcal{H}') \overset{r^{t_i,i}(l)}{\rightarrowtail} \mathcal{H}, (\mathcal{M}[\texttt{return}\ e], W, R', i, m', \mathcal{H}'), -} \textsc{Read}$$

$$\frac{l \notin \mathrm{dom}(W) \qquad R(l) = (e, i)}{\mathcal{H}, (\mathcal{M}[\texttt{read}\, l], W, R, i, m', \mathcal{H}') \overset{\epsilon^{t,i}}{\rightarrowtail} \mathcal{H}, (\mathcal{M}[\texttt{return}\ e], W, R, i, m', \mathcal{H}'), -} \textsc{ReadRSet}$$

$$\frac{W(l) = (e, i)}{\mathcal{H}, (\mathcal{M}[\texttt{read}\, l], W, R, i, m', \mathcal{H}') \overset{\epsilon^{t,i}}{\rightarrowtail} \mathcal{H}, (\mathcal{M}[\texttt{return}\ e], W, R, i, m', \mathcal{H}'), -} \textsc{ReadWSet}$$

$$\mathcal{H}, (m, W, R, i, m', \mathcal{H}') \overset{ab^{t,i}}{\rightarrowtail} \mathcal{H}, \texttt{atomic}\ m', - \quad \textsc{Rollback}$$

Figure 5.15. Operational semantics of Λ_{TWI}: Control flow to and in the twilight zone.

$$\frac{check(R, \mathcal{H}) = ok}{\mathcal{H}, (\mathcal{M}[\texttt{return}\ e \ggg m], W, R, i, m', \mathcal{H}') \overset{\epsilon^{t,i}}{\rightarrowtail} \mathcal{H}, (\mathcal{M}[m\ \texttt{tt}\ e], W, R, i, m', ok), t} \textsc{TwiOk}$$

$$\frac{check(R, \mathcal{H}) = bad}{\mathcal{H}, (\mathcal{M}[\texttt{return}\ e \ggg m], W, R, i, m', \mathcal{H}') \overset{\epsilon^{t,i}}{\rightarrowtail} \mathcal{H}, (\mathcal{M}[m\ \texttt{ff}\ e], W, R, i, m', bad), t} \textsc{TwiBad}$$

$$\mathcal{H}, (\mathcal{M}[\texttt{reload}], W, R, i, m', ok) \overset{\epsilon^{t,i}}{\twoheadrightarrow} \mathcal{H}, (\mathcal{M}[\texttt{return}\ ()], W, R, i, m', ok), t \quad \textsc{ReloadOk}$$

$$\frac{j \text{ fresh} \qquad R = \{l \mapsto \mathcal{H}(l) | l \in \mathrm{dom}(R)\}}{\mathcal{H}, (\mathcal{M}[\texttt{reload}], W, R, i, m', bad) \overset{ab^{t,i}, at^{t,j}, \overline{r^{t,j}(l)}}{\twoheadrightarrow} \mathcal{H}, (\mathcal{M}[\texttt{return}\ ()], W, R, j, m', ok), t} \textsc{ReloadBad}$$

$$\mathcal{H}, (\mathcal{M}[\texttt{ignoreUpdates}], W, R, i, m', f) \overset{\epsilon^{t,i}}{\twoheadrightarrow} \mathcal{H}, (\mathcal{M}[\texttt{return}\ ()], W, R, i, m', ok), t \quad \textsc{IgnoreUpdates}$$

$$\mathcal{H}, (\texttt{retry}\ , W, R, i, m', f) \overset{ab^{t,i}}{\twoheadrightarrow} \mathcal{H}, \texttt{atomic}\ m', - \quad \textsc{Retry}$$

$$\mathcal{H}, (\texttt{return}\ e, W, R, i, m', bad) \overset{ab^{t,i}}{\twoheadrightarrow} \mathcal{H}, \texttt{atomic}\ m', - \quad \textsc{CommitFail}$$

$$\frac{\mathcal{H}' = \mathcal{H}[W']}{\mathcal{H}, (\texttt{return}\ e, W, R, i, m', ok) \overset{co^{t,i}(\bar{l})}{\twoheadrightarrow} \mathcal{H}', \mathcal{P}[t \mapsto \mathcal{M}[\texttt{return}\ e]], -} \textsc{Commit}$$

Figure 5.16. Operational semantics of Λ_{TWI}: Heap checks.

$$\frac{\forall l \in \mathrm{dom}(R) : R(l) = \mathcal{H}(l)}{check(R, \mathcal{H}) = ok} \qquad \frac{\exists l \in \mathrm{dom}(R) : R(l) \neq \mathcal{H}(l)}{check(R, \mathcal{H}) = bad}$$

$$\frac{\forall l \in \mathrm{dom}(R) \cap \mathrm{dom}(W) : R(l) = \mathcal{H}(l)}{check(R, W, \mathcal{H}) = ok} \qquad \frac{\exists l \in \mathrm{dom}(R) \cap \mathrm{dom}(W) : R(l) \neq \mathcal{H}(l)}{check(R, W, \mathcal{H}) = bad}$$

a `reload` or `ignoreUpdates`. Thus, the transaction can repair or ignore its inconsistencies and still commit successfully.

The definition of the helper function for the heap check is in Figure 5.16. The following statement m takes a boolean value and the return value of the transaction body. The boolean value reveals the outcome of the consistency check to the transaction's execution context (`tt` for a successful, `ff` for a failed check). Further, the thread identifier in the global state is set to the identifier of the thread executing the transaction.

If there are no inconsistencies, the `reload` operation does not change the internal state of the transaction. Otherwise, entries in the read set are replaced by their counterparts in the global heap. This corresponds semantically to an abort of the transaction and the start of a new transaction which adopts the reads set and write set, as well as the execution context of the aborted predecessor. The annotated effects reflect this by emitting the abort effect for the transaction, the begin effect for the new transaction, and a list of all read effects as listed in the read set. Also, the transaction's state is now found consistent with respect to the current heap and is flagged with *ok*.

In a similar way, `ignoreUpdates` puts the transaction into a committable state by setting the state flag to *ok*. Because no new values are observed, nor global operations performed, the empty effect is emitted to the trace.

With `retry` , the transaction is aborted and reverted. Correspondingly, an abort effect is emitted to the execution trace.

When the twilight zone has been reduced to a return statement, the rule for commit transfers its entries from the write set to the global heap (Commit). A corresponding commit effect containing the locations of the modified variables is emitted, and the thread identifier in the global state which indicated that a twilight zone is executed is reset.

If the transaction has been found inconsistent with respect to the global heap when entering its twilight zone, and it has not obtained an update of the read variables via `reload` or explicitly ignored updates via `ignoreUpdates`, the commit fails (CommitFail). The transaction formally aborts and is restarted completely.

In Figure 5.17, the rules for evaluating an IO monad embedded into a transaction are shown. With `IOtoSTM` , an statement to be evaluated in the IO monad can be lifted into the twilight zone of a transaction. The statement, with the exception of `atomic` expressions (see IOtoSTMErr below), is evaluated in a top level environment. The effects are transferred to the enclosing transaction. When a new thread is spawn (IOtoSTMSpawn), it is added to the system's thread pool for later execution. IOtoSTMEnd returns to the execution context of the enclosing transaction.

Figure 5.18 shows the rules for repair operations in the twilight zone. Evaluating `inconsistent` l yields the result of comparing the value for l in the read set with the one in the global heap. Similarly to Write, the `update` operation replaces the value in the write set (Update), while the `reread` operation returns the value for a reference in the read set (Reread).

Figure 5.17. Operational semantics of Λ_{TWI}: Embedding of I/O operations.

$$\frac{e \neq \texttt{atomic}\ m' \qquad \mathcal{H}, \{t \mapsto e\}, - \xRightarrow{\alpha} \mathcal{H}, \{t \mapsto e'\}, -}{\mathcal{H}, (\mathcal{M}[\texttt{IOtoSTM}\ e], W, R, i, m, f) \overset{\alpha^{t,i}}{\twoheadrightarrow} \mathcal{H}, (\mathcal{M}[\texttt{IOtoSTM}\ e'], W, R, i, m, f), t} \text{ IOtoSTM}$$

$$\frac{\mathcal{H}, \{t \mapsto e\}, - \xRightarrow{\alpha} \mathcal{H}, \{t \mapsto e'; t' \mapsto e''\}, -}{\mathcal{H}, (\mathcal{M}'[\texttt{IOtoSTM}\ e], W, R, i, m, f), t \xRightarrow{\alpha^{t,i}} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{IOtoSTM}\ e'], W, R, i, m, f)]; t' \mapsto e''\}, t} \text{ IOtoSTMSpawn}$$

$$\mathcal{H}, (\mathcal{M}[\texttt{IOtoSTM}\ (\texttt{return}\ e)], W, R, i, m, f) \overset{\epsilon^{t,i}}{\twoheadrightarrow} \mathcal{H}, (\mathcal{M}[\texttt{return}\ e], W, R, i, m, f), t \quad \text{IOtoSTMEnd}$$

Figure 5.18. Operational semantics of Λ_{TWI}: Repair operations.

$$\frac{R(l) \equiv \mathcal{H}(l)}{\mathcal{H}, (\mathcal{M}[\texttt{inconsistent}\ l], W, R, i, m', f) \overset{\epsilon^{t,i}}{\twoheadrightarrow} \mathcal{H}, (\mathcal{M}[\texttt{return ff}], W, R, i, m', f), t} \text{ InconsFalse}$$

$$\frac{R(l) \neq \mathcal{H}(l)}{\mathcal{H}, (\mathcal{M}[\texttt{inconsistent}\ l], W, R, i, m', f) \overset{\epsilon^{t,i}}{\twoheadrightarrow} \mathcal{H}, (\mathcal{M}[\texttt{return tt}], W, R, i, m', f), t} \text{ InconsTrue}$$

$$\frac{W' = W[l \mapsto (e, i)] \qquad l \in \mathrm{dom}(W)}{\mathcal{H}, (\mathcal{M}[\texttt{update}\ l\ e], W, R, i, m, f) \overset{\epsilon^{t,i}}{\twoheadrightarrow} \mathcal{H}, (\mathcal{M}[\texttt{return}\ ()], W', R, i, m, f), t} \text{ Update}$$

$$\frac{R(l) = (e, j)}{\mathcal{H}, (\mathcal{M}[\texttt{reread}\ l], W, R, i, m, f) \overset{\epsilon^{t,i}}{\twoheadrightarrow} \mathcal{H}, (\mathcal{M}[\texttt{return}\ e], W, R, i, m, f), t} \text{ Reread}$$

Figure 5.19. Operational semantics of Λ_{TWI}: Error states.

$$\frac{l \notin \mathrm{dom}(R)}{\mathcal{H}, (\mathcal{M}[\texttt{inconsistent}\ l], W, R, i, m', f) \overset{ab^{t,i}}{\twoheadrightarrow} \mathcal{H}, \texttt{error}, -}\ \textsc{InconsErr}$$

$$\frac{l \notin \mathrm{dom}(W)}{\mathcal{H}, (\mathcal{M}[\texttt{update}\ l\ e], W, R, i, m, f) \overset{ab^{t,i}}{\twoheadrightarrow} \mathcal{H}, \texttt{error}, -}\ \textsc{UpdateErr}$$

$$\frac{l \notin \mathrm{dom}(R)}{\mathcal{H}, (\mathcal{M}[\texttt{reread}\ l], W, R, i, m, f) \overset{ab^{t,i}}{\twoheadrightarrow} \mathcal{H}, \texttt{error}, -}\ \textsc{RereadErr}$$

$$\mathcal{H}, (\mathcal{M}[\texttt{IOtoSTM atomic}\ m'], W, R, i, m, f) \overset{ab^{t,i}}{\twoheadrightarrow} \mathcal{H}, \texttt{error}, -\qquad \textsc{IOtoSTMErr}$$

Figure 5.19 defines the rules which lead to erroneous states in the system. Errors are induced by invalid read or write operations inside the twilight zone as depicted in Figure 5.19. A read (or write) operation is illegal in the twilight code if its location has not been read (or written) in the preceding `STM` phase of the transaction. Another source for errors is the nesting of transactions within `IOtoSTM`. As there is no obvious good semantics nested transactions [57], we follow here the semantics that is specified for Haskell's STM and other STMs, and dynamically reject the evaluation of a nested `atomic` m.

Errors abort the enclosing transaction. When they are propagated to the top-level, they terminate the execution of the associated thread.

We can again apply standard techniques to establish progress and preservation for Λ_{TWI}:

Theorem 5.5.1 (Type soundness)**.** *The type system in Figure 5.10 is sound with respect to the operational semantics of* Λ_{TWI}*.*

5.5.3. Semantics of Twilight transactions

Twilight STM does not only provide an enriched interface for programming transactions, it also allows weakening of isolation semantics of transactions. In database transactions, it is common to have several levels of isolation. Weakening of the isolation level can have undesirable and unexpected effects.

To aid the programmer in employing relaxed isolation semantics, all twilight transactions adhere to the principle of consistency. Therefore, the operational semantics does not allow zombie transactions that are doomed to fail, or exhibit all kind of undesired behavior due to inconsistent memory snapshots violating data invariants.

Lemma 5.5.1 (Consistency)**.** *A twilight transaction always operates on a consistent memory snapshot.*

Proof of 5.5.1: The consistency of the transaction's memory snapshot can only be violated by reading variables that were updated on the global heap since the beginning of the transaction. We therefore have to consider all rules that operate on the global heap. These rules are easy to identify as they emit read and commit effects.
Case distinction on rules accessing the global heap.

- *Case* Atomic: When starting the transaction, a copy of the global heap is obtained. This operation is atomic and cannot be interleaved by modifications of the heap.
- *Case* Read: The rule Read checks upon each first access to a reference if it is consistent with the variables that have been read so far. Therefore, each reference is compared to its counterpart in the reference copy of the heap that has been acquired when starting the transaction. Only if the current heap contains the same value as the heap copy, the value has not changed, and the read operation is successfully performed.
- *Case* ReloadBad: The reload of the read set is performed in an atomic operation that cannot be interleaved with any update operation. Both the local copy of the heap and the read set are updated with the current values in the heap.
- *Case* Commit: All update operations that are issued by a transaction get published to the global heap via Commit. As the commit is performed as one indivisible operation, the heap's consistency is not violated, and no inconsistent state can be observed by another transaction.

End case distinction on rules accessing the global heap. □

Starting from a program which employs standard atomic blocks, how does adding a twilight zone influence the program's semantics? Given the guarantee of consistent memory snapshots, the programmer can specify the desired isolation semantics for each program in Λ_{TWI} individually. In the next sections, we show how operations in the twilight zone can define the isolation level of opacity and snapshot isolation by transforming STM monads from Λ_{STM} in Λ_{TWI}.

5.5.4. Opacity in Λ_{TWI}

Implementing opacity in TwilightSTM is straightforward by transforming the code statically with $[\![\cdot]\!]_o$. The transformation extends the atomic blocks with an (empty) twilight zone which simply returns the result of evaluating the STM monad in the block. All other expressions are not changed. A formal definition of the transformation can be found in Figure 5.20.

Figure 5.20. Λ_{TWI}: Twilight zones for opacity.

$$
\begin{array}{rcl}
[\![x]\!]_o & = & x \\
[\![\texttt{tt}]\!]_o & = & \texttt{tt} \\
[\![\texttt{ff}]\!]_o & = & \texttt{ff} \\
[\![()]\!]_o & = & () \\
[\![\lambda x.e]\!]_o & = & \lambda x.[\![e]\!]_o \\
[\![e_1\ e_2]\!]_o & = & [\![e_1]\!]_o\ [\![e_2]\!]_o \\
[\![\texttt{if}\ e_1\ e_2\ e_3]\!]_o & = & \texttt{if}\ [\![e_1]\!]_o\ [\![e_2]\!]_o [\![e_3]\!]_o \\
[\![\texttt{return}\ e]\!]_o & = & \texttt{return}\ [\![e]\!]_o \\
[\![e_1 \gg\!= e_2]\!]_o & = & [\![e_1]\!]_o \gg\!= [\![e_2]\!]_o \\
[\![\texttt{spawn}\ e]\!]_o & = & \texttt{spawn}\ [\![e]\!]_o \\
[\![\texttt{new}\ e]\!]_o & = & \texttt{new}\ [\![e]\!]_o \\
[\![\texttt{read}\ e]\!]_o & = & \texttt{read}\ [\![e]\!]_o \\
[\![\texttt{write}\ e_1\ e_2]\!]_o & = & \texttt{write}\ [\![e_1]\!]_o\ [\![e_2]\!]_o \\
[\![\texttt{atomic}\ m]\!]_o & = & \texttt{atomic}\ ([\![m]\!]_o \ggg\!= \lambda x.\lambda y.\texttt{return}\ y)
\end{array}
$$

Theorem 5.5.2 (Opacity for Twilight transactions)**.** *The execution trace of a program m in Λ_{STM} is equivalent in effects to a trace of the transformed program $[\![m]\!]_o$ in Λ_{TWI}.*

Proof of 5.5.2: The proof is done by induction on evaluation steps.

The rules IO-Monad, Spawn, Atomic, STM-Monad, Alloc, Write, Read, as well as ReadWSet and ReadRSet in Λ_{STM} have an equal evaluation rule with the same name in Λ_{TWI} which is taken when evaluating expressions in the `IO` and `STM` monad. The rules in Λ_{TWI} merely extend the system and transaction state with twilight flags. In all the rules that define evaluation outside the `TWI` monad the flags are not set.

The scheduling for threads in Λ_{STM} can be simulated in Λ_{TWI} as the transaction which executes a twilight zone is running solo. It cannot be interleaved with evaluation steps from other threads.

The only differences arise when performing the commit operation in Λ_{STM} and its equivalent in Λ_{TWI}, namely the twilight bind and the following evaluation of the twilight zone.

We now consider the state of the system in Λ_{STM} where a thread has evaluated an atomic block to the transaction tuple $(\texttt{return}\ e, W, R, i, m', \mathcal{H}')$, and the scheduling chose this thread for the next step.
Case distinction on the applicable rules.

- *Case* COMMIT: The rule requires that $check(R, \mathcal{H}) = ok$. It then yields in the execution trace the following step:

$$\begin{array}{rl} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e, W, R, i, m', \mathcal{H}')]\} \\ \overset{co^{t,i}(\bar{l})}{\Longrightarrow} & \mathcal{H}'', \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{return}\ e]\} \end{array}$$

  where $\mathcal{H}'' = \mathcal{H}[W]$ and $\bar{l} = \mathrm{dom}(W)$.

- *Case* ROLLBACK: If the check failed, that is, $check(R, \mathcal{H}) = bad$, or a non-deterministic choice requires the transaction to abort, the execution trace continues with a rollback:

$$\begin{array}{rl} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e, W, R, i, m', \mathcal{H}')]\} \\ \overset{ab^{t,i}}{\Longrightarrow} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic}\ m']\} \end{array}$$

End case distinction on the applicable rules.

Now, consider the rules that are applicable in Λ_{TWI} when evaluating the corresponding expression

$$(\texttt{return}\ e \ggg \lambda x.\lambda y.\texttt{return}\ y, W, R, i, m', \mathcal{H}')$$

Case distinction on the applicable rules.

- *Case* TWIOK: The rule requires that $check(R, \mathcal{H}) = ok$. The evaluation then proceeds with these steps:

$$\begin{array}{rl} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e \ggg \lambda x.\lambda y.\texttt{return}\ y, W, R, i, m', \mathcal{H}')]\}, - \\ \overset{\epsilon^{t,i}}{\Longrightarrow} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda x.\lambda y.\texttt{return}\ y)\ \texttt{tt}\ e, W, R, i, m', ok)]\}, t \\ \overset{\epsilon^{t,i}}{\Longrightarrow} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda y.\texttt{return}\ y)\ e, W, R, i, m', ok)]\}, t \\ \overset{\epsilon^{t,i}}{\Longrightarrow} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e, W, R, i, m', ok)]\}, t \\ \overset{co^{t,i}(\bar{l})}{\Longrightarrow} & \mathcal{H}', \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{return}\ e]\}, - \end{array}$$

  with $\mathcal{H}' = \mathcal{H}[W]$ and $\bar{l} = \mathrm{dom}(W)$

- *Case* TWIBAD: The rule requires that $check(R, \mathcal{H}) = bad$. Hence, at commit the transaction failed verification because the empty twilight zone does not perform any repair or ignore the inconsistencies.

$$\begin{array}{rl} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e \ggg \lambda x.\lambda y.\texttt{return}\ y, W, R, i, m', \mathcal{H}')]\}, - \\ \overset{\epsilon^{t,i}}{\Longrightarrow} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda x.\lambda y.\texttt{return}\ y)\ \texttt{ff}\ e, W, R, i, m', bad)]\}, t \\ \overset{\epsilon^{t,i}}{\Longrightarrow} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda y.\texttt{return}\ y)\ e, W, R, i, m', bad)]\}, t \\ \overset{\epsilon^{t,i}}{\Longrightarrow} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e, W, R, i, m', bad)]\}, t \\ \overset{ab^{t,i}}{\Longrightarrow} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic}\ m']\}, - \end{array}$$

- *Case* ROLLBACK: As in Λ_{STM}, the transaction aborts and restarts.

$$\begin{array}{rl} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e \ggg\!= \lambda x.\lambda y.\texttt{return}\ y, W, R, i, m', \mathcal{H}')]\}, - \\ \xrightarrow{ab^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic}\ m']\}, - \end{array}$$

End case distinction on the applicable rules.

Each execution trace in Λ_{STM} has an equivalent counterpart in Λ_{TWI}: if the transaction commits successfully in Λ_{STM}, there is an equivalent execution trace in Λ_{TWI} which commits and results in the same final global state. In the same way, an abort in Λ_{STM} can be simulated by an abort in Λ_{TWI}.

For each of these execution traces, the effect traces are equivalent in effects, as the effect traces in Λ_{TWI} contain only additional empty effects. □

5.5.5. Snapshot isolation in Λ_{TWI}

To implement snapshot isolation, transactions need to operate on a consistent memory snapshot. Further, as explained in Section 5.4, the entries in the write set must be checked for intermediate updates between the beginning of the transaction and its commit.

By Lemma 5.5.1, the operational semantics of Λ_{TWI} exacts memory consistency. Therefore, the twilight code just needs to specify operations that obviate lost updates.

To simplify the transformation in the formal calculus, we enlarge the formal language Λ_{TWI} with a new primitive, `wsetCons`. The operation `wsetCons` tests for inconsistencies in the write set. In an implementation of TwilightSTM, this operation can easily be provided as a primitive. Alternatively, all references to the variables that are modified can be dynamically tagged (see Section 4.2.4), and tested individually for inconsistencies with `inconsistent` .

We can define a transformation from Λ_{SI} to Λ_{TWI} which preserves the operational semantics by yielding traces that are equivalent in effects. As with $[\![\cdot]\!]_o$, only the atomic blocks are transformed:

$$\begin{array}{rcl} [\![\texttt{atomic}\ m]\!]_s & = & \texttt{atomic}\ [\![m]\!]_s \ggg\!= \lambda x.\lambda y.\texttt{wsetCons} \gg\!= \\ & & \qquad \lambda b.\texttt{if}\ b\ (\texttt{ignoreUpdates} \gg\!= \lambda z.\texttt{return}\ y)\ (\texttt{retry}\ )) \end{array}$$

All other expressions are transformed recursively, analogously to $[\![\cdot]\!]_o$.

Theorem 5.5.3 (Snapshot Isolation for Twilight transactions)**.** *The execution trace of a program m in Λ_{SI} is equivalent in effects to a trace of the transformed program $[\![m]\!]_s$ in Λ_{TWI}.*

Proof of 5.5.3: As in the proof of 5.5.2, all rules but the commit rule in Λ_{SI} have an equivalent counterpart in Λ_{TWI}.

Figure 5.21. Extending $\Lambda_{TWI}$with snapshot operations.

Syntax:

$$e \in \text{Exp} ::= \cdots \mid \texttt{wsetCons}$$

Typing rules:

$$\frac{}{\Sigma|\Gamma \vdash \texttt{wsetCons} : \texttt{TWI}\,()}\ \text{T-WSetCons}$$

Operational semantics:

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{wsetCons}], W, R, i, m', f)] \qquad check(R, W, \mathcal{H}) = ok}{\mathcal{H}, \mathcal{P}, t \overset{\epsilon^{t,i}}{\Longrightarrow} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return tt}], W, R, i, m', f)]\}, t}\ \text{WSetCons}$$

$$\frac{\mathcal{P}(t) = \mathcal{M}[(\mathcal{M}'[\texttt{wsetCons}], W, R, i, m', f)] \qquad check(R, W, \mathcal{H}) = bad}{\mathcal{H}, \mathcal{P}, t \overset{\epsilon^{t,i}}{\Longrightarrow} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\mathcal{M}'[\texttt{return ff}], W, R, i, m', f)]\}, t}\ \text{WSetIncons}$$

We again consider the possible execution steps at commit time in both formalizations and show that they yield equivalent results.

Consider the state of the system in Λ_{SI} where a thread has evaluated an atomic block to the transaction tuple $(\texttt{return}\ e, W, R, i, m', \mathcal{H}')$, and the scheduling chooses this thread for executing the next step.
Case distinction on the applicable rules Λ_{STM}.

- *Case* Commit: The rule requires that $check(R, W, \mathcal{H}) = ok$. It then yields in the execution trace the following step:

  $$\mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e, W, R, i, m', \mathcal{H}')]\} \overset{co^{t,i}(\bar{l})}{\Longrightarrow} \mathcal{H}'', \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{return}\ e]\}$$

  where $\mathcal{H}'' = \mathcal{H}[W]$ and $\bar{l} = \text{dom}(W)$.

- *Case* Rollback: If the check failed, i.e. $check(R, W, \mathcal{H}) = bad$, or a nondeterministic choice of rules requires the transaction to abort, the execution trace is continued with a rollback:

  $$\mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e, W, R, i, m', \mathcal{H}')]\} \overset{ab^{t,i}}{\Longrightarrow} \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic}\ m']\}$$

End case distinction on the applicable rules Λ_{STM}.

As before, we distinguish between the rules that are applicable in Λ_{TWI} when evaluating the corresponding transformed expression:

$$\begin{aligned} m \quad = \quad & \texttt{return}\ e \ggg\!= \lambda x.\lambda y.\texttt{wsetCons} \ggg\!= \\ & \lambda b.\texttt{if}\ b\ (\texttt{ignoreUpdates} \ggg\!= \lambda z.\texttt{return}\ y)\ (\texttt{retry}\ ) \end{aligned}$$

Case distinction on the applicable rules in Λ_{SI}.

- *Case* TwiOk: The rule requires that $check(R, \mathcal{H}) = ok$. The evaluation then proceeds with these steps:

$$\begin{array}{ll} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(m, W, R, i, m', \mathcal{H}')]\}, - \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda x.\lambda y.\texttt{wsetCons} \gg\!= \ldots)\ \texttt{tt}\ e, W, R, i, m', ok)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda y.\texttt{wsetCons} \gg\!= \ldots)\ e, W, R, i, m', ok)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{wsetCons} \gg\!= \lambda b. \ldots, W, R, i, m', ok)]\}, t \end{array}$$

  As $check(R, \mathcal{H})$ implies $check(R, W, \mathcal{H})$, the execution continues with the rule WSetCons.

$$\begin{array}{ll} \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return tt} \gg\!= \lambda b. \ldots, W, R, i, m', ok)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda b.\texttt{if}\ b\ \ldots)\ \texttt{tt}, W, R, i, m', ok)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{if tt}\ \ldots\ \ldots, W, R, i, m', ok)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{ignoreUpdates} \gg\!= \ldots, W, R, i, m', ok)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ () \gg\!= \ldots, W, R, i, m', ok)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda z.\texttt{return}\ e)\ (), W, R, i, m', ok)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return}\ e, W, R, i, m', ok)]\}, t \\ \xrightarrow{co^{t,i}(\bar{l})} & \mathcal{H}', \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{return}\ e]\}, - \end{array}$$

- *Case* TwiBad: The rule requires that $check(R, \mathcal{H}) = bad$.

$$\begin{array}{ll} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(m, W, R, i, m', \mathcal{H}')]\}, - \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda x.\lambda y.\texttt{wsetCons} \gg\!= \ldots)\ \texttt{ff}\ e, W, R, i, m', bad)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda y.\texttt{wsetCons} \gg\!= \ldots)\ e, W, R, i, m', bad)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{wsetCons} \gg\!= \lambda b. \ldots, W, R, i, m', bad)]\}, t \end{array}$$

  The two possible consistency states for the write set yield these cases:

  Case distinction on $check(R, W, \mathcal{H})$.

  - *Case* $check(R, W, \mathcal{H}) = ok$: The rule WSetCons is the only applicable

rule.

$$\begin{array}{rl} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{wsetCons} \gg\!= \lambda b. \ldots, W, R, i, m', bad)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return tt} \gg\!= \lambda b. \ldots, W, R, i, m', bad)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda b.\texttt{if } b \ \ldots\ )\ \texttt{tt}, W, R, i, m', bad)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{if tt } \ldots\ \ldots, W, R, i, m', bad)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{ignoreUpdates} \gg\!= \ldots, W, R, i, m', bad)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return } () \gg\!= \ldots, W, R, i, m', ok)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda z.\texttt{return } e)\ (), W, R, i, m', ok)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return } e, W, R, i, m', ok)]\}, t \\ \xrightarrow{co^{t,i}(\bar{l})} & \mathcal{H}', \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{return } e]\}, - \end{array}$$

– *Case* $check(R, W, \mathcal{H}) = bad$: The rule WSetIncons is the only applicable rule.

$$\begin{array}{rl} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{wsetCons} \gg\!= \lambda b. \ldots, W, R, i, m', bad)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{return ff} \gg\!= \lambda b. \ldots, W, R, i, m', bad)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[((\lambda b.\texttt{if } b \ \ldots\ )\ \texttt{ff}, W, R, i, m', bad)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{if ff } \ldots\ \ldots, W, R, i, m', bad)]\}, t \\ \xrightarrow{\epsilon^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(\texttt{retry }, W, R, i, m', bad)]\}, t \\ \xrightarrow{ab^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic } m']\}, - \end{array}$$

End case distinction on $check(R, W, \mathcal{H})$.

- *Case* Rollback: As in Λ_{STM}, the transaction is aborted and restarted.

$$\begin{array}{rl} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[(m, W, R, i, m', \mathcal{H}')]\}, - \\ \xrightarrow{ab^{t,i}} & \mathcal{H}, \mathcal{P}\{t \mapsto \mathcal{M}[\texttt{atomic } m']\}, - \end{array}$$

End case distinction on the applicable rules in Λ_{SI}.

Again, no interleavings are possible when executing the twilight zone, so the traces for the execution in Λ_{TWI} and Λ_{SI} have an equivalent counterpart in the respective other formal language which yields effect traces that are equal in effects. □

5.5.6. Irrevocability in Λ_{TWI}

A major restriction of many transactional systems is the lack of support for executing irrevocable actions whose effects cannot be rolled back. These are in general I/O

operations, such as system calls, that need to be executed inside transactions. It is possible to embed such calls into the twilight zone of transactions in Λ_{TWI}. The following lemma illustrates how this can be done.

Lemma 5.5.2 (Irrevocability)**.** *Let*

$$T = m_1 \gg\!= \lambda x.\texttt{ignoreUpdates} \gg\!= \lambda y.\texttt{IOtoSTM}\, m_2 \gg\!= \lambda z.m_3$$

be a well-typed transaction in Λ_{TWI}. Then m_2 is evaluated at most once in each run of the transaction unless m_3 is contains an explicit call to `retry` *.*

Proof of 5.5.2: A transaction is restarted when it either contains a call to `retry` in the twilight zone, or it applies Rollback during execution of the transactional body, or when its evaluation of the twilight zone ends in applying the rule CommitFail.

In the first case, by the assumption made in the lemma, only m_1 may contain a `retry` operator. Also, in the second case, the rule Rollback can only be applied when evaluating m_1. In both cases, the monadic expression m_2 is not executed before the restart is performed.

Consider the following cases upon evaluating `ignoreUpdates`:
Case distinction on entering the twilight zone.

- *Case* TwiOk: The state flag is set to *ok*. Then, the state is unchanged when executing `ignoreUpdates`.

- *Case* TwiBad: The state flag is set to *bad*. In IgnoreUpdates, the flag is finally set to *ok*.

End case distinction on entering the twilight zone.

There is no evaluation rule which switches the transaction's flag to the *bad* state. Hence, if no error is thrown or no call to `retry` occurs in m_3, the rule Commit is applied, and the transaction commits successfully. Hence, the monadic expression m_2 is only evaluated once. □

Besides `ignoreUpdates`, also the `reload` operator allows a transaction to switch into an irrevocable mode and to obtain an updated snapshot of the read set. These operations also interact in such a way that calling `ignoreUpdates` before `reload` prevents the transaction from obtaining the current, possibly updated values. A programmer should therefore take care that each execution path in the twilight zone has preferably only one of these operations. The implementation of Twilight STM in Haskell 6.3 ensures this with its parametrized monads and a special *Safe* monad with irrevocable semantics.

5.5.7. The power of Twilight operations

In the last sections, we have shown how different semantics can be implemented for a transaction in $\Lambda_{TWI}$by employing twilight operations such as `ignoreUpdates`, `reload`, or `wsetCons`.

The programmer should be aware that these operations can also be (mis-) used to implement semantics that are usually undesirable. For example, the transaction

$$\texttt{atomic}\,\{\texttt{read}\,x \gg\!= \lambda v.\,\texttt{write}\,x\,(v+1) \gg\!\!\gg \lambda z.\texttt{ignoreUpdates}\}$$

can cause lost updates if another transaction commits to x concurrently.

Though, to simplify the reasoning about the interleaving and possible interaction of transactions, Λ_{TWI} provides only consistent memory snapshots for each transaction. This means that neither read skews nor non-repeatable reads can be observed. The interleaving can partially be observed by inspecting the read set's state and updated values, and the twilight zone may then react on these observations as the programmer specified.

Chapter 6.

Implementation

We implemented Twilight STM by extending the TL2 algorithm [20]. Like TL2, the implementation relies on a global counter/timer `T`. Each shared variable is associated with a version number that represents the time of its last modification. The first (transactional) read of a variable creates an entry in the read set comprising the variable, its value, and its version number at the time of the read operation. Write operations to shared variables are performed lazily. They are first recorded locally in the transaction's write set and their publication to shared memory is delayed to the commit phase.

To evaluate Twilight STM's usability and performance, we implemented the algorithm in form of libraries for several programming languages. The first implementation is a conservative extension of the TL2 algorithm for the C programming language. A program not using twilight code has therefore the same transactional guarantees and semantics as TL2. This implementation can be used with any platform supporting *pthreads*. Its correct application in a program hinges solely on programming conventions, which are neither checked nor statically enforced for the time being.

We implemented also a reference implementation in Java. This implementation is object-based and provides an object-oriented callback-style API: a transaction body is represented as an object with two methods, **`transactional`**`()` and **`twilight`**`()`. Executing such a transaction body means to open a transaction, run **`transactional`**`()`, prepare to commit (and restart if needed), and run the **`twilight`**`()` method, which leads to a commit or a restart. The body object also serves as a container to communicate values between the transactional code and the twilight code.

Finally, we investigated the usability of Twilight in Haskell. In contrast to the other implementations, Haskell's type system provides the means to statically check and enforce the correct usage of the API. Using parameterized monads, a transaction is separated into a declarative transactional and an imperative twilight phase. Special read and write handles enforce strong atomicity, and eliminate the possibility of erroneous and unsafe variable access. Further, the scope of a tag is statically limited to the scope of a transaction, thus preventing the programmer from incorrect usage. Because of the static guarantees, the Haskell code requires less error handling.

6.1. C

The API for the C implementation is given in Listing 3.1. It is introduced in detail in Section 3.2. The code builds on the *pthreads* library which implements the POSIX standard for thread management.

In this section, we give an account of some details concerning the implementation with respect to the C programming language.

Weak atomicity

In accordance to the nature of C, the C implementation of Twilight STM has word-granular conflict detection and resolution. This admits the usage of STM even in the presence of pointer arithmetic. As many other C implementations of STM, Twilight STM provides weak atomicity. Shared memory can be accessed either through the transactional read and write methods, or directly without any synchronization. In the latter case, concurrently running threads can exhibit data races, and it is not specified which values a thread obtains. In particular, it may observe inconsistencies due to partial updates of memory locations during a transaction's commit. The programmer is therefore strongly advised to omit the detour via the STM synchronization only during phases of single-threaded execution, for example when initializing data structures before threads are spawned.

Timestamps and locks

Twilight STM is a time-stamp based STM algorithm. The globally shared timer emits timestamps for marking the updates to shared memory. It is incremented for each successfully committing transaction with a CAS operation. Read-only transactions do not increment the timer.

Long running-applications might cause the timer to overflow at some point. In this case, the timer needs to be reset to zero and the timestamps of the transactionally shared variables similarly need to be re-initialized. We did not observe any overflows when evaluating the benchmarks.

To keep the memory overhead low, the STM library does not associate timestamps and locks to each memory word. Instead, when running **`stm_start`**, a static hash map is allocated containing this transactional meta data. Each memory address is mapped with a simple hash function to an entry in this map. The size of the hash map can be adapted to the application's workload to reduce the probability of phantom conflicts because of shared timestamps and locks.

For the entries in the hash map, the two lowest bits are used for encoding the lock status of the location (free, reserved, locked), the other bits contain the current timestamp. When entering the twilight zone, the transaction tries to set the lock bits for each variable in its write set from free to reserved with a CAS operation. No further CAS operations are required.

Transactional memory access

Due to the decoupling of locks and the shared memory locations, it is not possible to atomically read the memory word and the versioned lock. The read operation therefore starts with reading the versioned locks. If the lock and timestamp allow proceeding, it reads the desired memory location, then again reads the versioned lock. If the versioned lock has changed, the protected memory has possibly changed, and the read could have returned inconsistent values. Hence, the transaction has to abort.

To stop concurrently running transactions from reading free'ed memory or running into ABA problems, the memory chunk that is to be deallocated is added to the write set of the transaction. When committing, the timestamps of all words in the respective area are incremented, thus invalidating further read accesses.

In some few cases, it is possible that a transaction deallocates a memory location just after another transactions found the versioned lock in a state admitting a read access. When the reading transaction now accesses the memory location in question, it may cause a segmentation fault. To prevent a program from terminating in this situation, Twilight STM masks the segfault signal during the execution of **`stm_read`**.

Rollback of transactions

Twilight STM in C accomplishes the rollback of transactions with the combination of `setjmp/longjmp`. The `setjmp` method is called when a transaction is started and saves the calling environment. When aborting a transaction, first the transaction's metadata is cleared to prevent space leaks. Then, a call to `longjmp` reinstalls the program state to the pre-transactional situation.

Changes to local variables that are introduced before the start of the transaction are not reverted. For enforcing a rollback on these variables, a special version of local write is available which saves the pre-transactional states before modifying the variables.

6.1.1. Evaluation

Twilight STM is intended for situations where a transaction is not allowed to restart, in particular when I/O operations are involved. This feature is powerful, but it cannot be measured quantitatively. Another typical use for Twilight STM is contention management based on insight into the system's state dynamics. Thus, the experiments for evaluating the Twilight STM concentrate on the contention management aspect. To show the competitiveness of Twilight STM, we compare the performance of the C library for Twilight STM with the TL2 reference implementation provided with a selection of applications from the STAMP benchmark [19].

The applications can be characterized as follows:

- The *k-means* algorithm is used for grouping objects into a fixed number of

clusters such that each object belongs to the cluster with the nearest mean. The benchmark provides a multi-threaded clustering algorithm where each thread processes a partition of the objects iteratively. Transactions protect the update of the cluster center that occurs during each iteration. The amount of contention among threads depends on the number of clusters, with larger values resulting in less frequent conflicts.

The k-means benchmark has short transactions with small read and write sets. Only a small percentage of execution time is spent in transactions, and contention on shared memory is rather low.

- In the *labyrinth* benchmark, threads calculate in parallel paths in a three-dimensional uniform grid which represents a maze. Each calculation of such a path is enclosed in a transaction, and conflicts occur when two threads pick overlapping paths. To reduce these conflicts, transactions operate on thread-local copies of the grid. Only when a thread wants to add a path to the global grid, it revalidates by re-reading all the grid points along the new path. If validation fails, the transaction aborts and starts with a new, updated copy of the global grid.

  The labyrinth benchmark features long transactions with very large read and write sets, inducing a high contention rate. Almost all of the code is executed within transactions.

- The *ssca2* application is taken form the Scalable Synthetic Compact Applications Benchmark suite. It constructs an efficient graph data structure using adjacency arrays and auxiliary arrays for presenting large, directed, weighted multi-graph. In the transactional version of SSCA2, threads add concurrently nodes to the graph, using transactions to protect accesses to the adjacency arrays.

  The workload characteristics are similar to k-means: comparatively short transactions with small read and write sets.

- The *vacation* benchmark emulates a travel reservation system on a kind of in-memory data base. Each threads randomly reserves, updates, or cancels travel items such as cars, hotel rooms or flights for a customer as long as they are still available.

  By adjusting the parameters, different workloads and contention loads for the transactions can be arranged. For our evaluation we chose the settings as advised by the STAMP paper, which amounts to medium sized read and write sets, and low or medium contention.

The benchmark machine contains two AMD Opteron processors 6174 (12 Cores, 2.20 GHz, 64.0 GByte on 4 nodes). It runs a Linux operating system and GHC 6.12.3. The processors support hyper-threading, but because of spin-locking in several parts of the STM algorithm, the scalability of the applications is impeded

when using more than 12 threads. The following figures give the median taken from 10 runs of each application.

Figure 6.1 shows the results for running the benchmarks linked against an STM library implementing the TL2 algorithm, and with the Twilight STM library. The applications do not use any twilight zones, but are restricted to the standard STM operations.

The speedup graphs (left column) show the scaling behavior normalized with respect to the single-threaded STM version in each case. With increasing the number of threads, Twilight STM shows good scaling behavior, in particular when compared with TL2. For the real execution time (middle column), TL2 outperforms Twilight STM for the applications with small transactional workloads and few threads with a factor of 2, whereas Twilight STM shows better performance for the labyrinth benchmarks with its large read and write sets. With the exception of the ssca2 application, Twilight reduces the number of aborts significantly, thus decreasing the amount of restarts (right column). The reason for this difference lies in Twilight's distinction between three different lock states. Whereas TL2 transactions acquire exclusive access already before checking the read set, Twilight only reserves the locks, thus allowing concurrent transactions to proceed with reading locations that are associated to these locks. If the transaction has to abort because the read set check was unsuccessful, the other transactions have progressed and have a chance to commit later, while they have to restart in TL2.

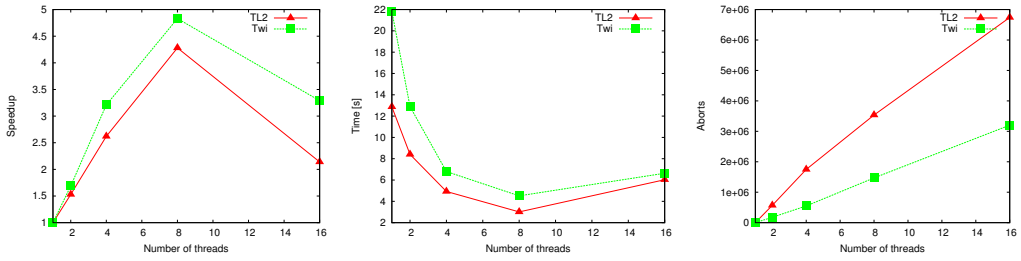
Both, TL2 and Twilight, induce a high synchronization overhead in the benchmark applications. In comparison, the single-threaded execution without any synchronization is up to 5 times faster than the single-threaded execution of the STM versions. A detailed analysis of the STM and single-threaded execution without synchronization overhead can be found in [19].

One reason for Twilight's lower performance on the applications with small transactional workload is because the Twilight library has been optimized for high workloads. Read and write sets are internally implemented with hash maps with an initial size of 64 elements. Thus, the overhead in starting and ending a transaction is comparatively high in applications like kmeans and ssca2. These results suggest that adapting STM implementations to different kinds of workload can influence an application to a great extend.

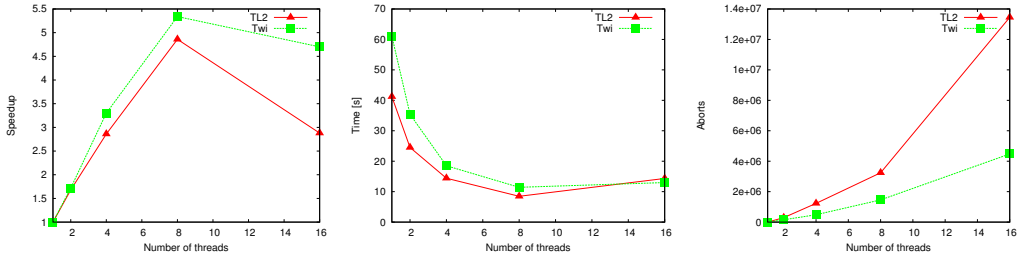
To measure the effect of repair operations, we extended the implementation for the linked lists code provided by STAMP with tagging as explained in Section 3.3.2. We then added twilight zones to the vacation benchmark that prevents restarts if changes to the list structure, containing available reservation items, do not invalidate the data consistence.

Figure 6.2 shows the results for TL2, and Twilight with and without twilight zone. As the figures suggest, the time needed to detect and repair possible inconsistencies is equal to the time needed to rollback and re-execute the conflicting transactions. The results from the vacation benchmark indicate that only applications with high contention spots and where repair is less expensive than rollback can benefit from Twilight with respect to performance.

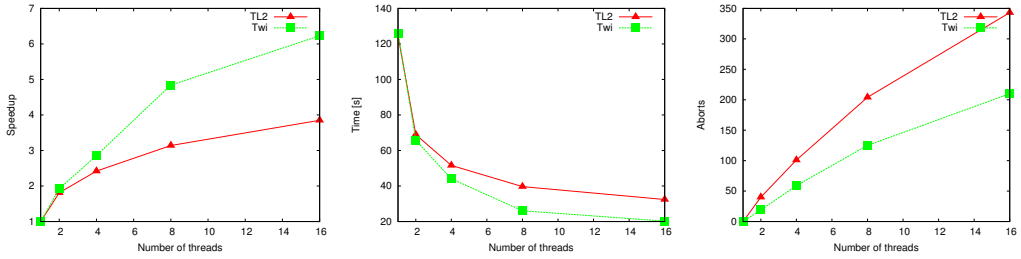
Figure 6.1. Twilight STM: STAMP benchmark suite - kmeans, labyrinth, ssca2.



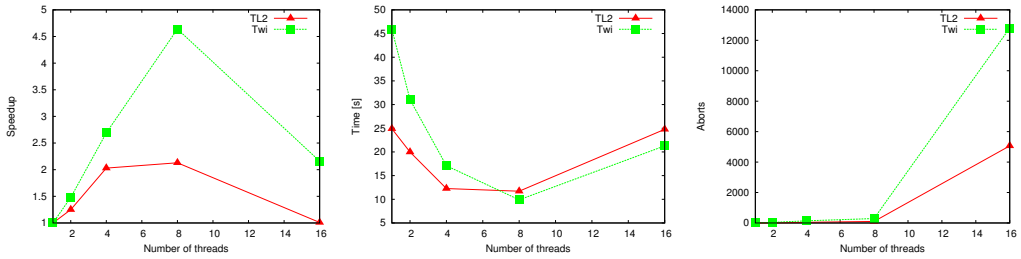
(a) kmeans-high.



(b) kmeans-low.

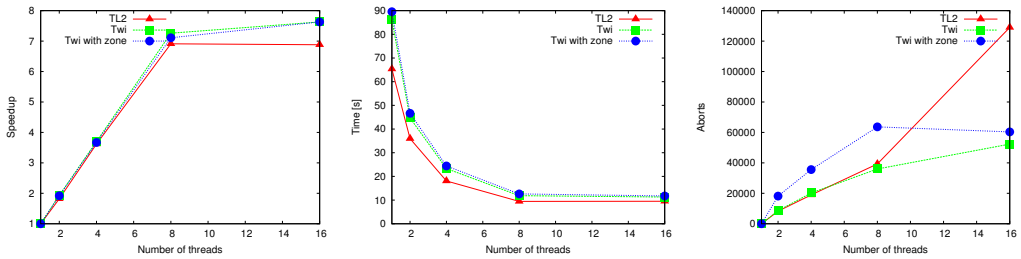


(c) labyrinth.

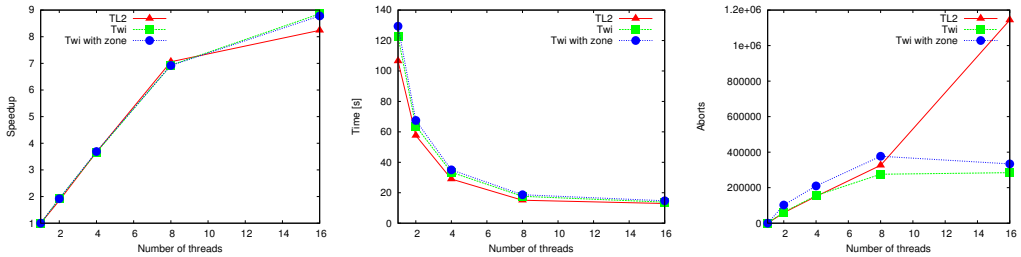


(d) ssca2.

Figure 6.2. Twilight STM: STAMP benchmark suite - vacation.

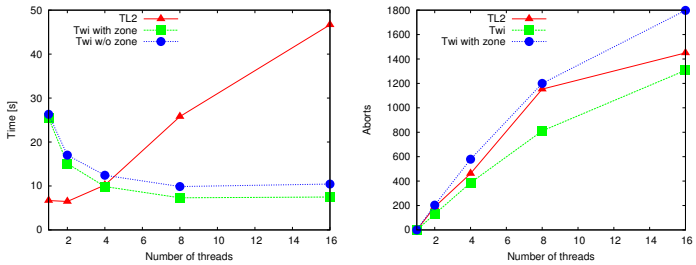


(a) vacation-low.



(b) vacation-high.

Figure 6.3. Twilight STM: Micro benchmark - Singly-linked list.



Listing 6.1 Twilight API for Java.

```
public class Transaction<R> {
  public static <R> R execute(TransactionBody<R> body,
      Transaction<?> parent) {..}
  public <T> void write(GlobalVar<T> var, T value){...}
  public <T> T read(GlobalVar<T> var) {...}
  public void restart(Transaction<?> bad) {...}
  public Set<GlobalVar<?>> inconsistencies(){...}
  public void reload(){...}

  public static abstract class TransactionBody<R> {
    protected Transaction<R> stm;
    abstract public void transactional();
    abstract public R twilight();
  }
}
```

In Figure 6.3, the execution times and aborts of running a micro benchmark on a singly-linked list against the three types of STM libraries. After filling a list with 20,000 elements from a key range of 1 to 40,000, 1,000 random operations (insert, delete, lookup) are performed on the data structure. The version with twilight zones uses tagging and conflict resolution as explained in Section 3.3.2. Each data point in the plot is obtained as the median of 10 runs.

For this kind of application, Twilight STM is much faster than TL2. When using 16 threads, the version with twilight zone has up to 14% less aborts, and is accordingly 28% faster than the Twilight version where no twilight zones are used.

6.2. Java

The Java implementation of Twilight, JTwilight, transfers the Twilight idea to an object-oriented language. The reference implementation makes use of Java's extensive concurrency libraries and is fully contained in just 600 LOC (including comments).

The API of JTwilight is given in Listing 6.1. Twilight transactions are instances of the `Transaction<R>` class, where `R` is the type of the transaction's return value. The code to be executed transactionally is defined by instances of the abstract class `TransactionBody<R>`. The atomic part is specified as the method body of the `transactional()` method, the twilight code is given in `twilight()`.

Different from the C implementation, JTwilight operates with object-granularity. All objects that are transactionally managed are enclosed in a `GlobalVar<T>` wrapper which contains a reference to the actual object, the time stamp of the last

modification, and a reentrant lock for guaranteeing mutual exclusion during the commit.

An example of a Twilight transaction in this object-oriented style is shown in Listing 6.2. The methods `putForks` and `getForks` atomically try to acquire an release the forks connected to a philosopher. The `Void` type is needed for transactions not returning any value (as seen in the example, these transactions simply return `null`).

The JTwilight implementation is a reference implementation which is not optimized for fast execution. Instead, we used it to experiment with nested Twilight transactions. The nature of twilight zones with its irreversibility aspects makes open nesting an appropriate choice for the nesting semantics. Following the suggestions in [57], reads and writes are transferred between child transactions and their parents in the following way:

- Objects that are present in an ancestor's read set are updated with the versions committed by the child. Thus, later accesses of the ancestor to the same data does not lead to conflicts.

- Objects read by a child transaction are added to the read set of its parent. Thus, the stack of nested transactions appears to have been executing on one memory snapshot.

- Modifications registered in a transaction's write set are not visible to a child transaction to prevent leakage of intermediates states. Thus, intermediate state is not accessible by the child transaction and cannot be leaked to the global state.

Listing 6.3 exercises the visibility of updates in a small example. Running the code fragment, it yields the following output:

```
z : xyz
y : modified by child
z : modified by parent
```

To reduce the number of restarts, a consistency check is done before starting a nested transaction. Further, when preparing the commit, the accumulated read set of all ancestors is again checked. If a conflict occurs on an object that is not in the current transaction's read set, the transaction does not enter its twilight zone, but all transactions up to the one that induced the conflict are aborted and restarted.

Incorporating nesting of transactions into Twilight STM leads to a sophisticated visibility semantics. The version implemented in JTwilight prevents leaking of intermediate state, yet leads to a growth in the read set. Other semantical choices for forwarding and retaining information between the parent and the child transaction are possible, but might violate the principles of consistency and isolation.

Listing 6.2 JTwilight: Dining Philosophers.

```
public class DiningPhilosophers {
  static Vector<GlobalVar<Boolean>> table;
  static Philosopher[] philosophers;
  private static class Philosopher extends Thread {
    int id;
    private void putForks() {
      Transaction.execute(
       new Transaction.TransactionBody<Void>() {
        public void transactional() {
          stm.write(table.get(id),Boolean.TRUE);
          stm.write(table.get((id +1) % NUM_PHILOS),true);
        }
        public Void twilight() {
          System.out.println("Philo "+ id +" finished.");
          return null;
        }
      },null);
     }

     private void getForks() {
       Transaction.execute(
        new Transaction.TransactionBody<Void>() {
         public void transactional() {
           getFork(id);
           getFork((id + 1) % NUM_PHILOSOPHERS);
         }
         private void getFork(int id) {
           boolean available = stm.read(table.get(id));
           if (available) {
             stm.write(table.get(id),Boolean.FALSE);
           } else {
             stm.restart(stm);
           }
         }
         public Void twilight() {
           System.out.println("Philo "+ id +" is eating.");
           return null;
         }
       },null);
     } ...
}
```

Listing 6.3 JTwilight: Nested transactions.

```
final GlobalVar<String> y = new GlobalVar<String>("abc");
final GlobalVar<String> z = new GlobalVar<String>("xyz");
Transaction.execute(
 new Transaction.TransactionBody<Void>() {
  int a;
  public void transactional() {
    a = stm.read(y);
    stm.write(z, "modified by parent");

    Transaction.execute(
      new Transaction.TransactionBody<Void>() {
      String b;
      public void transactional() {
        stm.write(y, "modified by child");
        b = stm.read(z);
      }
      public Void twilight() {
        System.out.println("z: "+ b);
        return null;
      }
    }, stm);
  }

  public Void twilight() {
    System.out.println("y: " + stm.read(y));
    System.out.println("z: " + stm.read(z));
    return null;
  }
}, null);
```

6.3. Haskell

In the Twilight library for Haskell, the STM data type is a composition of the IO monad with error and state monads. The state maintains a global counter to obtain unique time stamps that mark writes to transactionally managed memory.

The API employs Haskell's type system to restrict operations to the atomic or the twilight phase, depending on where they are safe to use.

We now have a closer look at the API which is given in Listing 6.4.

The Parameterized Monad STM

The parameterized monad $STM\ t\ p\ q\ a$ encapsulates a computation as the body of a transaction. The type parameters describe a computation of this type more closely:

- t is a static transaction identifier which restricts the scope of tags and variable handles to one transaction using monadic encapsulation [48];
- p and q statically indicate the phase of the transaction in the STM monad before and after the computation;
- a denotes the result type of the computation.

The function $atomically :: (\forall\ t.STM\ t\ p\ q\ a) \rightarrow IO\ a$ creates a new transactional scope with a fresh static transaction identifier t and executes its body computation atomically. The operations *gbind* and *gret* generalize the $\gg=$ and *return* operations of the standard *Monad* class to parameterized monads [44][1].

Twilight distinguishes three different phases in a transactional scope, which are indicated by the instantiation of the p and q parameters.

- Code in the *Atm* (atomic) phase enjoys full transactional execution. GHC's STM implementation [35] provides only this phase. Code that runs in the atomic phase is fully isolated from external changes to variables and vice versa. It always sees memory in a consistent state.
- In the *Twi* (twilight) phase, the consistency of the variable values read within the *Atm* phase of the transaction may be checked with respect to their current values. In the presence of inconsistencies a transaction is doomed to fail, unless the programmer switches explicitly to the safe phase.
- Once the *Safe* phase is reached, the transaction does not fail anymore unless explicitly requested by the programmer via *retry*. The twilight code has exclusive access to the variables in the transaction's write set. This isolation guarantee ensures that the I/O effects coincide with the outcome of the

[1] Our examples exploit GHC's convenient customization feature of the **do** notation through a simple local redefinition of $\gg=$ and *return* by *gbind* and *gret*.

Listing 6.4 Twilight API for Haskell.

```
  -- STM data
data STM t p q a  = ...   -- abstract type of computations
data TVar a       = ...   -- transactional variables
data RTwiVar t a  = ...   -- handle for rereading
data WTwiVar t a  = ...   -- handle for rewriting
  -- static states of a transaction
data Atm
data Twi
data Safe
  -- STM parameterized monad
atomically      :: (∀ t.STM t p q a) → IO a
gbind           :: STM t p q a → (a → STM t q s b) → STM t p s a
gret            :: STM t p p a
retry           :: STM r p q a
  -- transfer between phases
twilight        :: STM t Atm Twi Bool
reload          :: STM t Twi Safe ()
tryCommit       :: STM t Twi Safe ()
ignoreUpdates   :: STM t Twi Safe ()
  -- read and write operations
newTVar         :: a → STM t p p (TVar a)
readTVar        :: TVar a → STM t Atm Atm a
writeTVar       :: TVar a → a → STM t Atm Atm a
readTVarTwi     :: TVar a → Tag t a → STM t Atm Atm (a, RTwiVar t a)
writeTVarTwi    :: TVar a → a → STM t Atm Atm (WTwiVar t a)
rewriteTwiVar   :: WTwiVar t a → a → STM t p p ()
rereadTwiVar    :: RTwiVar t a → STM t p p a
  -- tags
newTag          :: STM t Atm Atm (Tag t)
markTVar        :: TVar a → Tag t → STM t Atm Atm ()
isInconsistent  :: Tag t → STM t p p Bool
  -- embedding IO
unsafeTwiIO     :: IO a → STM t p p a
safeTwiIO       :: IO a → STM t Safe Safe a
```

transaction. In this phase, the code may perform operations for repairing inconsistencies. It is also possible to safely perform non-reversible operations like I/O.

Each STM operation is indexed by its start and end phases. Hence, the type checker guarantees that it is not possible to perform the operations out of order or in the wrong phase. The *twilight* operation switches from the *Atm* phase to the *Twi* phase. Similarly, *reload* finishes the *Twi* phase and starts the *Safe* phase. The operation *tryCommit* also switches from *Twi* to *Safe*, but it aborts and restarts if the transaction is still in an inconsistent state. Otherwise, it proceeds in the *Safe* phase.

Reading and Writing Shared Memory

Similar to GHC's STM implementation, a shared memory location which can be accessed within a transaction has type *TVar a*. It encapsulates a value of type *a*. The operation *newTVar* creates a new transactional variable with an initial value. Within an atomic section, a *TVar a* is accessed via *readTVar* for reading and via *writeTVar* for writing.

In addition to these standard operations, the API provides special read and write operations for Twilight. The operation *readTVarTwi* returns the current value of type *a* as well as a handle of type *RTwiVar t a* where *t* is the current transaction identifier. This handle is associated to the same location as the underlying *TVar* and it may be used in the *Safe* phase to read the new value of the variable if it was the cause of an inconsistency. Similarly, the atomic write operation *writeTVarTwi* returns a write handle of type *WTwiVar t a* to enable writing this variable in the *Safe* phase.

After entering the Twilight zone, transactional variables can only be read or written via the read and write handles. Admitting the standard read operation on a *TVar* might yield a memory location that has not been touched in the preceding *Atm* phase. This can violate the property that a transaction operates on a consistent snapshot. Similarly, the *Safe* phase must not write variables that have not yet been written to in the *Atm* phase. We impose this restriction to keep the transaction's read and write set constant, which is required to guarantee deadlock freedom.

The operation *rereadTwiVar* returns the value of a variable as it is currently found in the read set. Within the *Atm* phase, it returns the same value as the *readTVar* operation on the associated *TVar*. After issuing a *reload*, in the *Safe* phase, the *rereadTVar* operation may return a new value if the underlying variable has caused an inconsistency and *reload* has obtained a new value for it. The old value is no longer available. The operation *rewriteTwiVar* updates the variable corresponding to the *WTwiVar* handle, but this update only takes effect when the transaction commits.

Tags

A tag *Tag t a* identifies a group of variables. The operation *newTag* returns a fresh tag without any variables attached to it. Its scope is restricted to the execution of the STM monad with static transaction identifier t. A *TVar* is added to a tag group either through *writeTVarTwi*, *readTVarTwi*, or *markTVar*.

In the Twilight zone, the programmer can apply *isInconsistent* to a tag to determine whether the tag is associated to an inconsistent variable.

Embedding I/O into STM

The STM monad shipped with GHC prohibits performing I/O within a transaction because I/O might violate the transactional semantics. Yet, it is sometimes desirable to include "transaction-safe/harmless" actions, like reading the system time or printing debug output, into transactional code. Like GHC's *unsafeIOtoSTM*, the operation *unsafeTwiIO* injects an I/O action into the STM monad without giving guarantees about when and how often the action may be executed. In contrast, an action performed with *safeTwiIO* in the *Safe* phase of a transaction is guaranteed to be executed exactly once.

The Twilight implementation in Haskell relies on a *parameterized monad* [44] to separate and order the different phases of a transaction. Further, we use monadic encapsulation [48] to restrict the scope of tags and handles to single transactions.

Heterogeneous collections

Twilight STM tracks the read and write set of a transaction to provide consistent memory snapshots and reduce contention on the global state. Both the read and write sets are heterogeneous collections of *TVar*s containing values of different types. Through the use of phantom types, the API enforces that values and references are consistently typed. Internally, we treat the values as having an unknown type and can safely coerce them back to the appropriate type.

Enumerating *TVar*s

The representation of a *TVar* in Twilight contains a lock to grant exclusive access to the variable. The underlying locking protocol requires that these locks are ordered to avoid deadlocks of the system. However, in Haskell, no pointer type (not even *StablePointer*) is an instance of the *Ord* class, so that we are forced to use integers to enumerate and order all *TVar*s. This numbering introduces a bottleneck in the implementation and consumes space, but we are not aware of a better solution without reimplementing *IORef*s and *MVar*s.

6.3.1. Comparison with GHC's STM

GHC is shipped with an implementation for Software Transactional Memory. It provides the interface to an STM monad, uses *TVar*s as shared memory locations

for transactions, and includes several interesting enhancements of STM, like the *orElse* operator, and the restarting of failed transactions only after modifications to the variables that were read. Handling of transactional memory is implemented directly in the run-time system.

GHC's STM prohibits the use of I/O operations by means of the type system. Using the STM monad to encapsulate all read and write operations, the effect of a transaction is restricted to modifications of transactional variables. Yet, GHC offers the unsafe lifting of I/O operations into the STM monad via $unsafeIOToSTM :: IO\ a \rightarrow STM\ a$, which is considered a highly dangerous loophole that breaks the functional guarantee.

In contrast, Twilight STM for Haskell is implemented as a library. This approach enables easy modification and testing of features, but decreases performance drastically. Also, GHC's STM implementation does not offer the Twilight features to repair potentially failing transactions.

6.3.2. Evaluation

The benchmark machine contains two AMD Opteron processors 6174 (12 Cores, 2.20 GHz, 64.0 GByte on 4 nodes). It runs a Linux operating system and GHC 6.12.3.

To evaluate our implementation of Twilight STM in Haskell, we adapted the Haskell benchmark that has been collected by Perfumo and others and has been used in several case studies on the GHC's STM implementation (e.g. [59]). Changes were made to the type signatures to adapt to Twilight's parameterized monads. Further, for some programs in the benchmark suite, the atomic blocks were extended with twilight zones that preserve the semantics of the program. All figures show the mean of 50 runs with the same parameter setting.

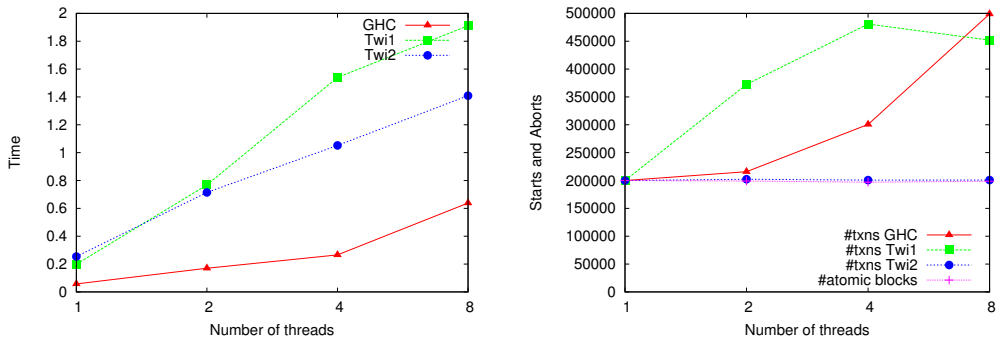
To reduce the influence of measuring as small as possible, the abort and commit counts are done in an unsafe way with a simple variable of type *IORef Int*. There is no extra synchronization involved (as would be when using an *MVar* or similar), yet a small percentage of updates to these counters are lost as can be seen in Figures 6.7(c) and 6.5(c).

Further, the speedup graphs show the scaling behavior normalized with respect to the single-threaded STM version in each case. A comparison of implementations using different concurrency strategies can be found in [21], for example.

Single transactional variable

Figure 6.4 shows the results on a small program which operates an transactional integer variable that is shared among multiple threads. It tries to perform in parallel a task that is inherently sequential: each thread is reading the variable and incrementing it by one. In the standard STM version, each conflicting access to the variable leads to a restart of the transactions (GHC and Twi1). For the version with twilight zone (Twi2), a small repair action is employed to reduce the number of

Figure 6.4. Twilight Haskell: Micro benchmark - Update operations on a single variable.



(a) Execution time.

(b) Starts and Aborts.

restarts. In case of intermediate updates, the counter is reread and the actual value then incremented. This repair effectively serializes the execution of the threads.

As expected, the performance of the program degrades with an increasing number of threads due to high contention (Figure 6.4(a)). Note that the number of restarts in the version with repair is drastically reduced (Figure 6.4(b)). The few restarts occur in a situation where the *readTVar* method sees that the variable is currently locked during a thread's commit.

Also, observe that in the single-threaded case the execution time for this simple update operation differs between GHC and Twi by a factor of 4. This gives an indication what the actual difference between the built-in and the library based version are. In examples where more transactional variables are involved it seems that the overhead in organizing these variables in the read and write sets is even worse.

Linked list

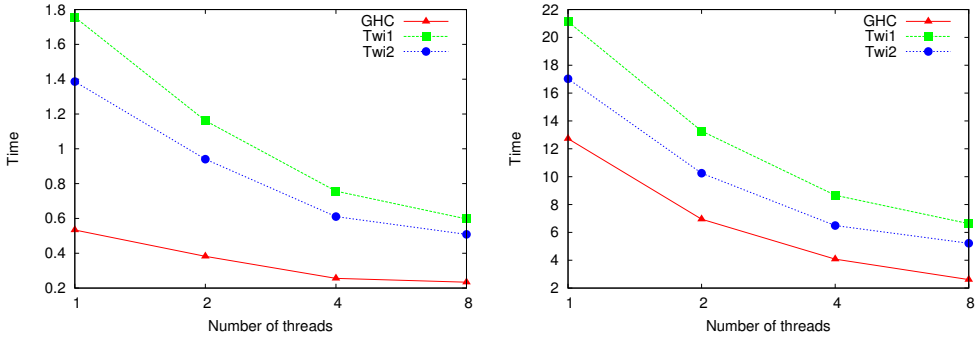
In the linked list benchmark, each thread atomically lookup, insert, or delete an element in a shared single-linked list where the keys of the list nodes are sorted. The version in which the atomic blocks are extended with twilight zones (Twi2) implements the fine-grained conflict detection as presented in Section 3.3.2.

Figure 6.5 displays runs of 100.000 operations on a shared list. The lists have a maximal capacity of 100 and 1000 elements, and they were initially filled with 50 and 500 keys. To simulate typical usage patterns of lists, the benchmark threads performed 90% lookup operations, 5% insertions, and 5% deletions.

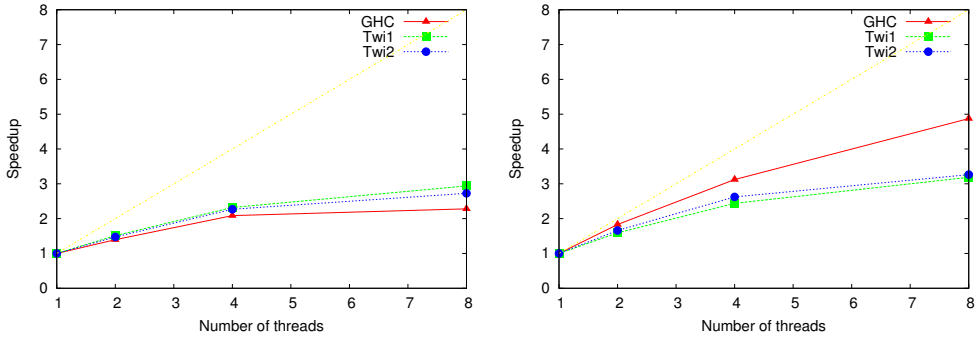
Again, not surprisingly, the program does not scale well when the number of threads is increased though the percentage of update operations is small. This holds especially for the shorter list.

Notice that both, the execution time and the number of aborts, are reduced when comparing the version with and without twilight zones. In particular, the execution

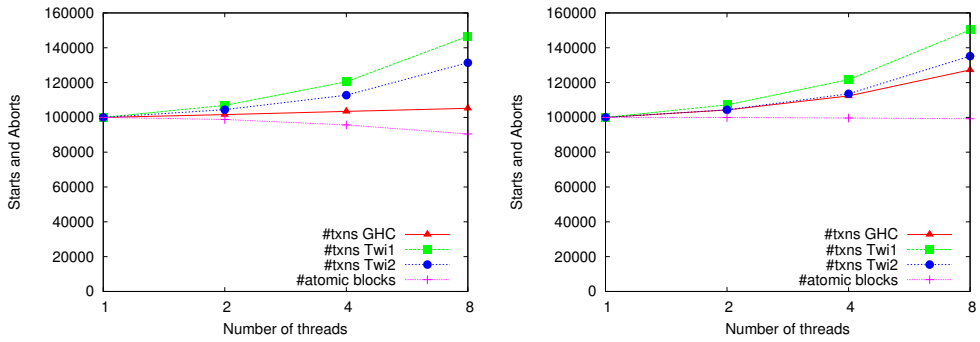
Figure 6.5. Micro benchmark: Linked list with 100 and 1000 elements, 20% update.



(a) Execution time.

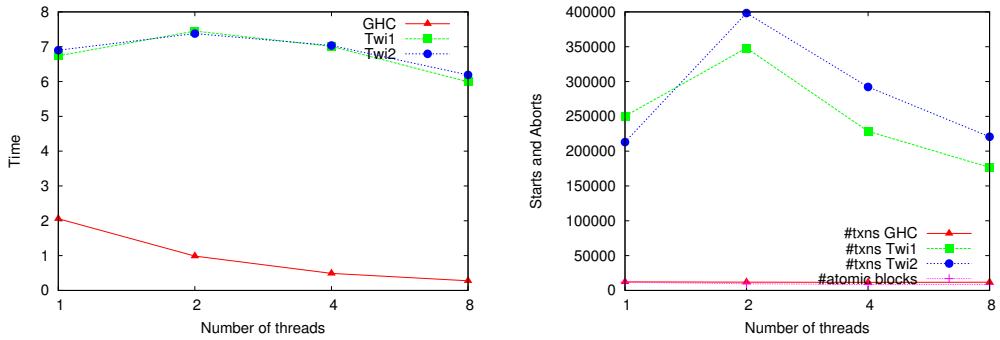


(b) Scaling.



(c) Starts and Aborts.

Figure 6.6. Twilight Haskell: Benchmark - Sudoku.



(a) Execution time.

(b) Starts and Aborts.

time is reduced by 15% to 21%.

Binary tree

The benchmark results in Figure 6.7 show operations on an unbalanced binary tree with a maximum of 1000 elements. The graphs in the left column were obtained by 20% updates (i.e. 10% insertions and 10% deletions), the right column shows 50% update operations. Again, the data structure was initialized with 500 elements.

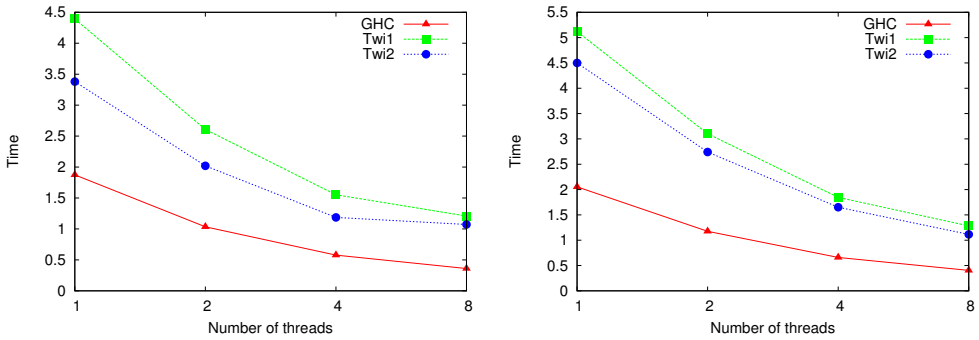
As anticipated, the scaling behavior of this program is better than with the linked list. The Twi2 version with twilight code gives again an improvement of about 15% to 20% over the standard version (Twi1).

Sudoku

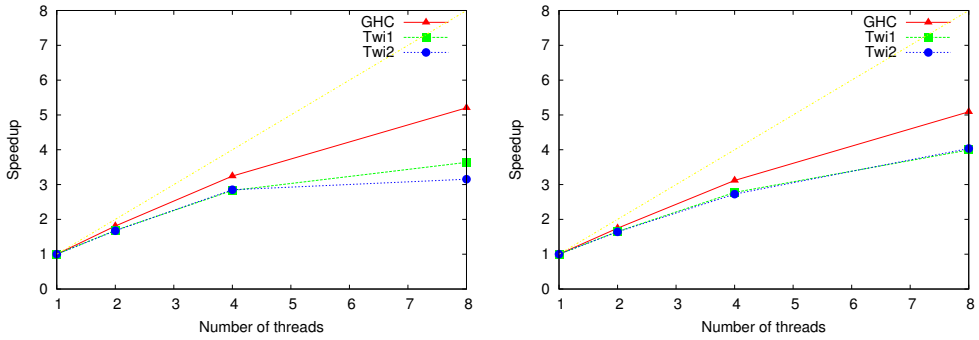
Finally, the sudoku benchmark gives some interesting insight into the different nature of GHC's STM and Twilight STM. To reduce the number of doomed transactions, the STM of GHC ensures the serialization of transactions with conflicting accesses to some variables. It only restarts a transaction if changes to some variable in the read set have been confirmed. This conflict management is rather expensive as transactions have to register themselves to all elements in their read set.

In contrast, Twilight STM optimistically restarts a transaction immediately after an abort. This scheme can lead to repeated aborts and restarts. Yet, if the contention on variables is low and transaction can run in parallel, this spinning of transactions has little impact. The situation changes if there is a mismatch between the number of active threads and processor cores. The running transactions, including the aborting ones, then have to share the computation power. This effect of spinning is even increased as the committing transactions are making also less progress in this case. In particular, the locks that are associated with the transactional variables are held longer which again leads to more aborts of conflicting transactions.

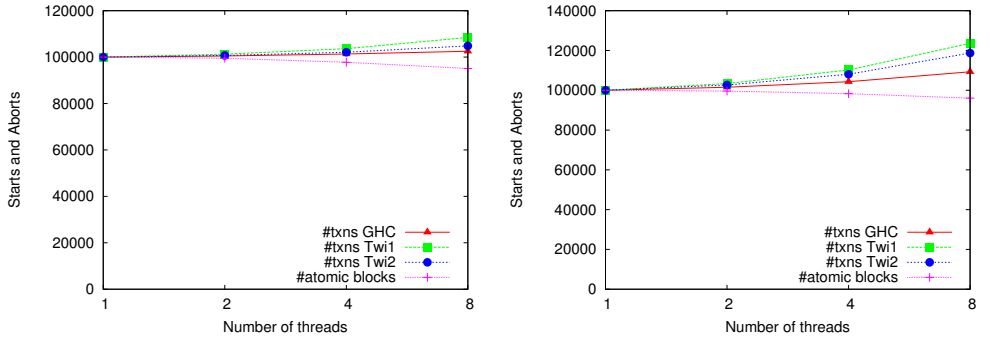
Figure 6.7. Binary tree with 1000 elements, 20% and 50% update.



(a) Execution time.



(b) Scaling.



(c) Starts and Aborts.

In the sudoku solver, threads that execute a transaction are spawned recursively until a solution for the given board is found. The actual number of spawned threads depends on the initial board state and the scheduling of the threads. For the three configurations that are used in the benchmark, it ranges from 100 to more than 11.000 threads spawned. Therefore, in Figure 6.6(b) a high rate of aborts is already observable in the single-threaded case.

Part II.

Decent STM

Chapter 7.

Introduction

Developing scalable software for multi-core processors is considerably harder than producing software that utilizes only one core. Developing software for distributed system introduces even more complexity to the software engineering process. A distributed system consists of several processing units that are connected with the help of a communication infrastructure. Synchronization of the shared global state involves communication either between individual nodes or with some dedicated central nodes. Standardized protocols, like OpenMP and MPI, reduce the complexity of programming these (often heterogeneous) systems. Yet, when programming against these interfaces, the exchange of data has to be issued explicitly as part of the program logic.

An additional complication is introduced by the high latency and instability of the communication layer. In a distributed setting, protocols need to take the increased latency and possible failure of nodes into account. Systems that rely on central components can suffer from performance bottlenecks and are susceptible to irrecoverable system failures.

In this second part of the thesis, we recast the STM paradigm for a distributed setting. Decent STM is a fully decentralized object-based STM algorithm. It entirely avoids locking and centralized components during execution of the program. The shared memory system is the union of all globally accessible objects (GAOs). All operations on GAOs involve only message-based communication between the transactions and a decentralized memory system. Delayed communication, e.g., caused by retransmissions in the transport layer of the network, only affects performance, but not consistency of the shared global state. The Decent STM algorithm is thus well suited even for large scale distributed systems.

As explained in detail in Section 5, transactional read and write accesses create dependencies on the shared data items. The standard correctness criterion for transactions, 1-copy-serializability, imposes strong restrictions on the ordering of transactions. The Decent STM algorithm employs a multi-version scheme for GAOs where transactions work on local copies of shared data, which are later transferred back to the decentral memory system. Transactions obtain lazily a consistent memory snapshot during their execution. Each modification of an object creates a new version. Upon a successful commit, this new version typically replaces the previous version of that data. The system keeps a limited list of committed versions for all shared data, the *version history* list. When reading from a GAO, a transaction

obtains a version which is consistent with all GAOs it has read so far. As we show in Section 11, the concept of the history list pays off because the effect of avoiding the read conflicts more than compensates for the increased aborts which are due to commit failures.

The price for using the version history is a higher rate of failed commits due to intermediate commits and an increased memory overhead. Limiting the version history reduces both, the likeliness of aborts at commit time caused by intermediate writes, and the overhead to store the version history. For pruning the histories, it is either possible to keep always a fixed number of versions for each GAO or remove all versions that are no longer obtainable by any running transactions. In the latter case, a transaction has to abort on a read operation, because it can always read a previous version that does not conflict with the data read so far.

Committing to the decentralized memory system involves a more sophisticated protocol as multiple parties have to reach consensus on which transactions are allowed to commit in the case of conflicts. Decent STM solves the commit consensus problem with a novel distributed randomized consensus protocol. It is based on voting messages involving only the concurrently committing transactions and the nodes of the distributed memory system to which new versions are committed. Other transactions may read the tentatively committed versions, but they have to abort if the consensus protocol does not elect the corresponding tentative version.

Decent STM implements as default the weaker snapshot isolation semantics. Snapshot isolation is a popular isolation level in replicated database systems because it allows long-running read-only transactions to safely coexist with short transactions updating the state. Further, it requires less strict consistency checks which are rather expensive in a distributed setting (see Section 5.4). For compatibility with applications requiring serializability of transactions, the Decent STM system allows either to switch to serializable snapshot isolation [18] or to revert to a classic two-phase commit protocol.

This thesis focuses on the core ideas of a distributed STM. Accordingly, we describe the Decent STM algorithm in a plain manner without going into detail with respect to optimizations. Nevertheless, Decent STM has been designed with several enhancements in mind, such as the possibility to cache, replicate and migrate the data structures so that they match the available memory and processor resources more efficiently.

Outline

- Chapter 8 starts with introducing the basic components of Decent STM, GAOs (Section 8.1) and transactions (Section 8.2). It then shows how a consistent memory snapshot can be obtained incrementally from the single object versions in Section 8.3. Section 8.4 explicates the pruning of version history lists, and Section 8.5 sets the commit phase forth.

- Chapter 9 is dedicated to the problem of commit consensus. After formally

defining the problem (Section 9.1), the randomized consensus commit protocol of Decent STM is delineated in Section 9.2.

- Chapter 10 presents the Decent STM system from a practical perspective. It outlines the modules of a Decent STM system in Section 10.1, and gives the STM interface for the object-system in Section 10.2. In Section 10.3, a tool to simplify the development of programs for Decent STM is introduced. The JTransactifier statically rewrites Java byte code into transactional style based on annotations on which objects are globally shared and where the code contains atomic code blocks.

- Chapter 11 describes and evaluates two implementations of Decent STM in Java: an implementation of the algorithm on a multi-core machine using in-memory communication (Section 11.1), and an implementation for multiple machines in a network (Section 11.2).

Chapter 8.

The Architecture of Decent STM

This chapter introduces the core algorithms of Decent STM. The main challenge in the distributed environment of Decent STM is to refrain from installing central components that impede the scalability of the total system. The Decent STM algorithm distinguishes two kinds of operations: thread-local operations on non-shared, private data, and global operations on shared, distributed memory.

The system enforces strong atomicity on all objects by statically distinguishing between thread-local and transactionally shared objects. This approach is usually referred to as *partitioning*.

The basic Decent STM architecture therefore consists of two components which control different aspects of the distributed memory: the globally accessible objects whose union represent the shared memory, and the transactions that operate on these shared objects by reading and modifying them.

8.1. Globally Accessible Objects

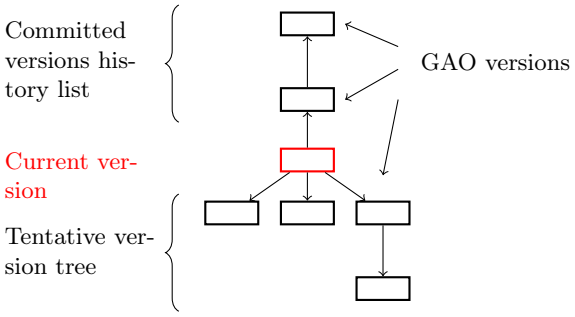
A globally accessible object (GAO) is a distributed data structure for shared-memory objects. It is an amalgamation of multiple versions of the data stored in the object.

Figure 8.1 sketches the structure of a GAO. The GAO versions belong either to the *version history list* or the *tentative version tree*. The version history list contains one or more subsequent successfully committed versions of the GAO. The head element of the list is the last committed version, often called the *current version*. The current version further is the root element of the tentative version tree. Beside the current version, this tree consists of the GAO versions that have not been successfully committed yet. The elements directly beneath the root participate in an ongoing commit consensus protocol. If transactions are allowed to proceed tentatively while the consensus has not been reached yet, they may obtain one of the tentative versions during their execution and can commit further versions, thus building the tree structure. Once consensus has been reached, the unsuccessful branches are pruned and the corresponding transactions must re-execute.

Each GAO is addressable by a globally unique identifier (GID) and is accessible by any transactions holding such a pointer to this object.

The representation of a GAO version consists of

Figure 8.1. Decent STM: Structure of a GAO.



- a unique GAO version identifier (GVID),
- the GID of the GAO to which it is associated,
- the actual data items to be stored,
- a reference to the version's predecessor,
- the transaction identifier (TID) of the transaction that wrote the GAO version, and
- a list of the TIDs of transactions that have read this GAO version and have yet neither committed nor aborted.

It suffices for the GVID to be unique among all GVIDs of the corresponding GAO. In combination with the GID we then obtain globally unique identifiers for the versions. The reference to the previous GAO version builds the committed versions history list of the corresponding GAO.

Furthermore, except for the TID list, the data structure of a GAO version is immutable. Thus, it can be replicated, for example in some local cache. After a successful commit or an abort, the TID of the then no-longer active transaction is removed from all GAO versions in the read set of the respective transaction. As the TID lists are only required for pruning the version history (see Section 8.4), delayed updates for synchronizing the TID lists do not interfere with the STM mechanism. At worst, this could introduce a performance degradation.

As the Decent STM algorithm is an object-based STM algorithm, the data stored in a GAO are in general serialized objects, though it is possible to use different representations for the versioned memory chunks.

8.2. Transactions

Transactions execute the atomic code blocks of an application. To this end, a transaction obtains local object copies (LOCs) of the GAOs. A LOC is a copy of the GAO version which is kept in thread-local memory for read and write accesses by

a transaction. When reading a GAO, a transaction obtains a LOC that corresponds to a particular version of that GAO which is consistent with all LOCs obtained so far. A successful commit then turns each LOC that the transaction has modified into a new version of that GAO.

To track the dependence relation of GAO versions and check for conflicts, the following meta data is kept for each transaction:

- a globally unique transaction identifier (TID),
- a read set, containing the LOCs of the GAOs that were read,
- a write set, containing the modified LOCs of the GAOs that were written,
- a create set, containing the initial versions of GAOs that are published when the transaction commits, and
- a check set, containing GVIDs of all GAOs accessed for consistency checks.

Together, the read set, write set, and create set constitute the memory snapshot on which the transaction performs its operations.

The following operations constitute the basic work flow of a transaction:

Start a transaction: This creates a transaction object with a new unique TID and empty read set, write set, create set, and check set.

Read a GAO: If a LOC for the GAO to be read already exists in the read set, the transaction reads the data from the LOC. Otherwise, it sends a fetch request to the corresponding GAO and waits for the response. Once this response arrives, it adds the received LOC into the read set, merges its check set with the check set received, and reads the data from the LOC.

Modify a GAO: The transaction writes the new values in the LOC and enters it to the write set if it has not been done before. The Decent STM algorithm assumes that a GAO has been read or created before it is modified.

Instantiate a GAO: The transactions creates a LOC and initializes it to its default value as defined by the object's data layout. Committing such a LOC creates both the GAO and its initial version.

Finish a transaction: When a transaction ends, it sends a commit request to all GAOs that have been modified by the transactions. If the commit request is positively acknowledged by the GAOs in question, the transaction is finished. Otherwise, it reverts its state and re-executes. The transaction object needs to be kept, first for a potential roll-back, later for providing the read and write sets that link the GAO versions. It may be disposed when the corresponding GAO versions are disposed.

A transaction may encounter unresolvable conflicts either when fetching a GAO from the global shared memory or upon commit. In both cases, the node that executes the transaction must perform a roll-back: the transaction-local state is reverted, and the local heap and stack frame are restored to the pre-transaction state.

Figure 8.1 gives the pseudo code for the thread-local operations (for the commit, see Section 8.5). The procedures **SEND** and **RECEIVE** represent the communication operations between a GAO and a transaction. A **SEND** procedure takes as arguments the receiver of the message, the type of the message, and possibly further parameters depending on the message type. A **RECEIVE** procedure yields the sender of the message, the message type, and data that has been transmitted in the message. When illustrating the algorithms, we assume that the details of message construction, serialization of objects, and network communication are hidden behind dedicated proxy objects.

The method STMSTART generates a unique identifier for a new transaction and initializes the transaction's local metadata, such as the read set, write set, create set, and check set.

When reading a GAO, first the write set and read set are checked for possible local object copies. If there is no local copy available, a read request is send to the corresponding GAO. The read request takes as parameter the current read set as tuples of GID and GVID. The actual values of the versions do not have to be included. (For further details on the necessity and employment of the read set, consider Section 8.3.2). Upon delivery of a LOC and its check set, the local meta data is updated with this information and the local copy is returned to the application. When the read request fails, the transaction is aborted.

The STMWRITE operation registers modified values in the local write set unless they have been created by the current transaction. When creating a GAO via STMCREATE, the transaction generates a unique GID, transforms the object into an LOC, and enters it into the create set. Until it is published during a commit, it is only locally accessible.

In comparison to timestamp-based STMs, the Decent STM algorithm treats the fetching and commit phase substantially differently, as they involve communication with the distributed memory system. We therefore have a closer look at these operations.

8.3. Fetching a GAO version

Decent STM uses a lazy snapshot algorithm to detect and resolve conflicts during transactional reads. A memory snapshot consists of the GAO versions that are fetched from the GAOs with the read operations.

Before we take a more formal view on how conflicts are detected and resolved during transactional reads, we illustrate the problem first with some examples.

Algorithm 8.1 Decent STM: Thread-local operations of a transaction.

```
method STMSTART()
    tid ← generate unique TID
    readset,writeset,createset,checkset ← ∅
end

method STMREAD(GID g)
    loc ← LOCALREAD(g)
    if loc ≠ null then return loc
    else
        reads ← {(gid, v) | gid ∈ readset.keys, v = readset.get(gid).version}
        SEND(g, FETCH, reads)
        if RECEIVE(g, DELIVER, loc, check) then
            readset.add(g, loc)
            checkset.merge(check)
            return loc
        else if RECEIVE(g, READFAIL) then
            abort transaction
        end if
    end if
end

method LOCALREAD(GID g)
    loc ← writeset.get(g)
    if loc = null then loc ← createset.get(g)
    end if
    if loc = null then loc ← readset.get(g)
    end if
    return loc
end

method STMWRITE(LOC loc)
    if writeset.get(loc.g) = null & createset.get(loc.g) = null then
        writeset.put(loc.g, loc)
    end if
end

method STMCREATE(Object obj)
    g ← generate unique GID
    loc ← create new LOC from obj
    createset.put(g, loc)
    return g
end
```

8.3.1. Example

We denote with v_i^j a GAO version, where i is the identifier of the transaction that committed the version, and j is the identifier of the GAO to which the version is associated.

Figure 8.2 displays the GAO dependencies as they evolve during the execution of the following program:

- Assume that a transaction T_{12} initializes GAO G_3, while another transaction T_{42} in parallel created GAO G_1 and G_2. Next, transaction T_{50} reads both G_1 and G_2, and commits to both GAOs updated version (Figure 8.2(a)).

- Now, transaction T_{70} reads GAO G_2. There are two versions for this GAO available, v_{42}^2 and v_{50}^2. To ensure linearizability of the memory accesses, the transaction obtains the most recent version, v_{50}^2. Later, it sends a read request to GAO G_3. Assume that GAO G_3 has been updated by transaction T_{60} while transaction T_{70} has been dealing with GAO G_2. Both versions, v_{12}^3 and v_{60}^3, would lead to a consistent memory snapshot for T_{70}. Again, the most recent version is chosen for answering T_{70}'s read request, thus linearizing transaction T_{60} before transaction T_{70}. The transaction then continues and updates GAO G_2 and G_3 (Figure 8.2(b)).

- A transaction T_{76} starts off with reading v_{70}^2. Meanwhile, G_1 and G_2 are updated by T_{83} (Figure 8.2(c)). When issuing a read request to G_1, the most recent version v_{80}^1 is inconsistent with T_{76}'s read set at this point. Hence, it has to revert to the older version v_{50}^1. The last read access to G_3 returns v_{70}^3. It can finally update and commit to GAO G_3.

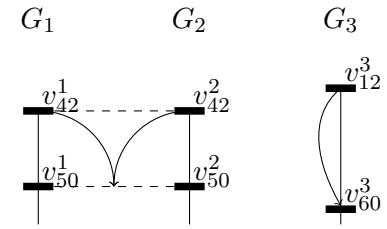
Note that it would not have been possible for transaction T_{76} to commit on G_1 or G_2 as this would cause a lost update. Further, the update of G_3 is allowed under snapshot isolation and serializable snapshot isolation, but not when the STM utilizes a standard two-phase-commit protocol. In this case, the intermediate update to G_2 by T_{83} would preclude a successful commit of T_{76}.

When constructing the memory snapshot for transaction T_{76}, reverting to a previous version of a GAO prevented the transaction from an abort. In fact, read-only transactions never have to abort if all versions are kept for each GAO. However, cutting down the version histories is necessary to reduce the overall memory consumption. This pruning of the version histories can be done in a conservative way such that a transaction is never forced to rollback due to a missing version. For example, the versions v_{12}^1 and v_{12}^2 are never accessed by a transaction whose first read access has been performed after the commit of transaction T_{42}. More details on pruning the version histories can be found in Section 8.4.

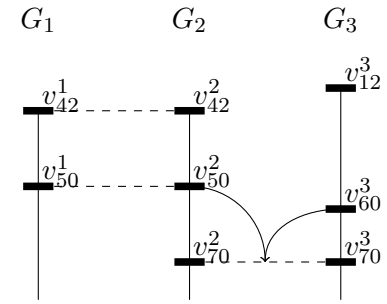
8.3.2. Constructing consistent memory snapshots

The basic algorithm for constructing a consistent memory snapshot is given in Algorithm 8.2. A GAO receives requests for reading and commit from transactions in

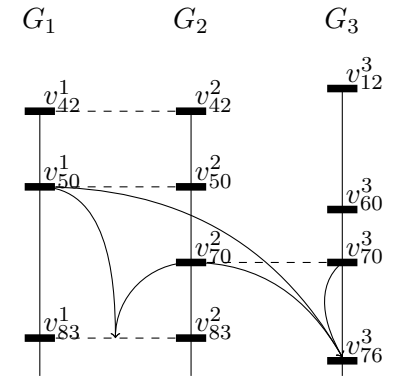
Figure 8.2. Decent STM: Constructing memory snapshots.



(a) Initial setting.



(b) Further updates.



(c) Reverting to older versions.

Algorithm 8.2 Decent STM: GAO execution loop - Read requests.

```
method INCONSISTENTWITH(GAOV gaov, List reads)
    checkSet ← gaov.checkSet
    for (gid,v) ∈ reads do
        if gaov.checkset.contains(gid) then
            if v is older than gaov.checkset.get(gid) then
                return true
            end if
        end if
    end for
    return false;
end

// execution loop of GAO
method RUN(GAO g)
    while true do
        if RECEIVE(t, FETCH, reads) then
            gaov ← g.mostRecent
            while INCONSISTENTWITH(gaov,reads) do
                gaov ← gaov.predecessor
                if gaov == null then
                    SEND(t, READFAIL)
                end if
            end while
            gaov.addToTIDList(t)
            SEND(t, DELIVER, gaov)
        else
            ... // continued in Algorithm 9.1
        end if
    end while
end
```

the form of messages. It answers the messages in a sequential way. In Section 9.2.2, details of the commit part can be found.

By construction, the read request sent by a transaction contains the current read set as tuples of GID and GVID. Starting from the most current version and following the predecessor chain, the GAO searches for a compatible version to deliver. To this end, it compares the versions read so far with the check set of the versions in its history.

If there is no compatible version available due to pruning on the history list, the GAO sends a read failure message to the transaction thereby, forcing the transaction to abort.

Further, the memory snapshot is linearizable since it contains for each GAO the most recent version which is consistent with the memory snapshot taken so far.

We are now going to develop a more formal description of the fetching operation. When a transaction T_j needs to fetch a version from a GAO's history list, say for GAO G_i, it picks one version, for example v_m^i. For the first read of a GAO by a transaction, the version must be the most recent one. But this does not necessarily hold for all following read accesses, as we have seen in the example.

Our algorithm establishes a partial order $\preceq$ on the versions of the GAOs according to their relative position in the version history list or the tentative version tree.

Definition 8.3.1. *A GAO version v_l^i is* dependent *on another GAO version v_m^i, $v_m^i \prec v_l^i$, if v_m^i is a predecessor of v_l^i in the version history list of GAO G_i.*

Now, let T_m be the transaction that committed v_m^i. Let R_m be the read set of T_m, and W_m its write set. We call the union of the read and write set the *check set* of version v_m^i and denote it by $C(v_m^i) = R_m \cup W_m$. Note that all versions written by the same transaction share the same check set. We further define for a set S of versions $C(S) = \bigcup_{v \in S} C(v)$ to be their check set.

The transitive closure $C^*(v)$ of a version's check set reflects all read and write dependencies of v, actually representing the memory snapshot of a transaction. It is formally defined as

$$C^*(v) = \bigcup_{k=1,\ldots} C^k(v)$$

where $C^1(v) = C(v)$ and $C^n(v) = C(C^{n-1}(v))$.

Transactions must not read GAO versions such that some versions date before and some after the elements of $C^*(v)$ because this would violate consistency of the memory snapshot taken. The option to access such GAO versions only arises if other transactions committed in the mean time and induced via the transitivity of check sets a dependency to the still running transaction.

To avoid this problem, we can use for all GAOs G corresponding to versions in $C^*(v)$ the order relation on GAO versions to check if $C^*(v)$ contains a more recent version of this GAO than the read set. This means we check that for all $v_n^h \in C^*(v_m^i)$, if there exists a $v_l^h \in R_j$, it holds that $v_n^h \preceq v_l^h$.

If the check holds, the fetch operation is successful and returns v_m^i. In this case, the operation also adds T_j to v_m^i's TID list.

Otherwise, the fetch operation fails for v_m^i. It retries and picks another version $v_{m'}^i \prec v_m^i$ from the GAO's history list. If no such earlier version exists, the fetch operation fails entirely and the transaction T_j must abort.

Theorem 8.3.1 (Correctness)**.** *The read operation of Decent STM constructs a consistent memory snapshot or aborts the transaction.*

Proof of 8.3.1: Following Definition 5.4.2, the memory snapshot is consistent under snapshot isolation if the algorithm does not admit any read skew.

Assume that the algorithm admits read skews. In the terminology of Decent STM, this means that a transaction T_k reads GAO versions v_n^i and later v_m^j such that there exists a version v_n^j and $v_n^j \prec v_m^j$, and there also exists a version v_m^i and $v_n^i \prec v_m^i$.

By construction of the check set, it holds that $v_m^i \in C(v_m^j) \subseteq C^*(v_m^j)$. Hence, the check at the fetch operation for a version of GAO j finds that there exists a version of GAO i such that $v_n^i \in R_k$ and $v_m^i \in C^*(v_m^j)$ with $v_n^i \prec v_m^i$. Therefore, the fetch operation reverts to a previous version, or issues an abort if no earlier version is available. □

Note that by construction it suffices to keep the most recent version of each GAO in the transaction's check set. Also, the check set does not change after committing the transaction. To obtain better performance, it can be cached so that the depth of the recursive calculation is small.

8.4. Limiting the Committed Version History List

When reading a GAO, going back in history can temporarily avoid a conflict. But it causes the reading transaction to depend on an older version. This potentially leads to a conflict at commit time as another transaction with an overlapping write set might have already updated variables.

Furthermore, the longer the version history, the more effort is required to iteratively validate the check sets in the transitive closure. This holds even in the case when the check sets are cached, because the number of involved GAOs can grow. For both reasons, the committed versions history list should be kept short.

The TID list in a GAOV contains all ongoing transactions which read this GAOV. A GAO version with an empty TID list is currently not used in any transaction. It is thus a candidate for pruning. However, we need a guarantee that the version is no longer reachable by any transaction for a read. The TID lists provide the system with the information that is necessary to decide when a transaction object and the related GAO versions may be discarded. We show its effect with the help of two lemmata.

Lemma 8.4.1 (Stability)**.** *Let v be some GAO version. If the TID lists of the predecessors of all $v' \in C^*(v)$ are empty, they will always remain empty.*

Proof of 8.4.1: A transaction only goes back in history when fetching a GAO version if it has a version in its read set that is older than any corresponding version in the transitive closure of the check set of v. Due to the algorithm, the TID list of this GAO's version in the read set is non-empty. Thus, if all predecessors of the $v' \in C^*(v)$ have empty TID lists, the transaction does not go back in history. Hence, the TID lists of these versions will remain empty as they will not be read by the transaction. □

Lemma 8.4.2 (Disposability)**.** *Let v be some GAO version. If the TID lists of the predecessors of all $v' \in C^*(v)$ are empty, v will not cause a (read) conflict.*

Proof of 8.4.2: Checking the transitive closure of the check set of a GAO version $v = v_m^i$ with respect to the current read set R_j of a transaction T_j fails if and only if $C^*(v_m^i)$ contains a GAO version v_n^h that is more recent than the respective version v_l^h in R_j, i. e. $\exists v_n^h \in C^*(v_m^i)$ with $v_l^h \prec v_n^h$. Then T_j would be registered in the TID list of v_l^h. This contradicts our assumption of empty TID lists for all predecessors of v_n^h as v_l^h is a predecessor of v_n^h. □

The disposability lemma gives a sufficient condition when a transaction object may be discarded, namely when all versions in the transitive closure of the transaction's check set have only predecessors with empty TID lists. The stability lemma shows that discarding transaction objects does not need to be synchronized because this condition is stable.

Finally, we can dispose of all GAO versions that are neither in the check set of any transaction nor the most recent version of a GAO. This entire process – i.e. checking the predecessors of the check set and disposing the transaction objects and GAO versions – is very similar to concurrent mark-and-sweep garbage collection. It can thus be implemented in a similar manner.

Clearly, a long lasting transaction may prevent the application of this rule, because the TID list of some GAO version might not become empty. In this case, the transaction is likely to fail regardless. It will only succeed if it did not read or write any conflicting GAO version. Therefore, the system clears the history list as described above. If the transaction then performs a conflicting read or commit, it fails immediately.

A practical evaluation of the effect on pruning the version history lists is done in Section 11.

The pruning of version histories does not take into account whether GAOs themselves are no longer reachable. For collecting the obsolete GAOs, a standard distributed garbage collector can be employed.

8.5. Committing a transaction

Besides the fetching of a GAOV for reading, the other global operation in Decent STM is the publishing of new versions to the distributed shared memory via the commit operation.

The implementation of an isolation level such as snapshot isolation or opacity requires that the commit must prevent lost updates. In Decent STM, this means that if for any GAO in $\{G_i \mid v_n^i \in W_n\}$ the corresponding read version $v_k^i \in R_n$ is not the latest version in the history list, the commit must fail. Otherwise, the new GAO version can be inserted into the tentative version tree.

The DecentSTM algorithm thus performs the following steps:

- For each GAO G_i which a transactions T_n wants to modify, it creates a new GAO version v_n^i in W_n. We call the corresponding version $v_k^i \in R_n$ its predecessor. Such a version exists because Decent STM requires that a transactions reads a GAO before writing to it.

- When issuing a commit, the version is inserted into the tentative version tree so that it becomes a child of its predecessor. If the previous version of any element in W_n is not the latest version in the respective history list, the transaction is notified that the commit failed and the transaction needs to be re-executed.

In a transaction's commit, checking for intermediate writes and adding the version to the tentative versions of the respective GAOs needs to be performed atomically for all $v_n^i \in W_n$. Thus, the transaction waits for the confirmation of all GAOs in its write set before it can proceed. The Decent STM commit phase resembles in this respect the classic two-phase commit.

If the current version has more than one tentative version pending, several transactions try to commit concurrently, thus creating write conflicts. In this case, a distributed consensus protocol gradually converges to one tentative version to be successful. This version is moved from the tentative version tree to the committed version history list.

For Decent STM, we developed a novel distributed consensus protocol that is tailored towards the commit process. This protocol is described in detail in the next chapter.

Chapter 9.

Distributed Commit Consensus

In the commit phase of the Decent STM algorithm, m globally accessible objects (GAOs) must decide in unison which of the n pending transactions that issued concurrently a commit request may finish the commit successfully. The standard approaches to solve this problem are based on phased commit protocols [32]. In the setting of these commit protocols, the transaction sends commit requests to the GAOs. If one of the GAOs issues a veto, the transaction has to withdraw its commit request and rollback. In implementations operating on one global address space, there is usually a fixed order in which the GAOs receive the commit requests. This ensures that in case of concurrent commits to the same memory locations, the first transaction to pass the consistency check is successful. In distributed settings, there is typically a central component that is responsible for serializing the commit requests in a globally consistent order.

In contrast to such a centralized approach, Decent STM uses a novel randomized distributed consensus protocol which leaves both the number of transactions and GAOs variable during the execution of the protocol. Instead of forwarding the requests to a centralized commit manager, the transactions communicate directly with the GAOs in question. The GAOs then need to reach a consensus on which transactions can proceed successfully, and which transactions have to abort. This *commit consensus* differs from the classic consensus problem as the number of participants for the GAOs and the transactions is not fixed and may even change during the execution of the protocol.

As in the previous chapter, we directly refer to single GAOs for now, though GAOs are usually grouped together and managed by a runtime instance to reduce the communication overhead (see Section 10.1).

For the protocol to work correctly, the commit consensus protocol needs to meet the following requirements:

Non-triviality If the entire network is non-faulty throughout the execution of the protocol, then, if a transaction is chosen by all GAOs in charge, it is successful.

Termination Every transaction reaches eventually either a successful or a failure state during commit.

Consistency It is impossible for a transaction to be in more than one state, i.e. a transaction is either successful, failed or pending. Once a transaction reaches a success or failure state, it remains in that state forever.

Stability Once a GAO version is published by a GAO, it cannot be withdrawn again.

We start this section with a formal definition of the problem that has to be solved for such a consensus protocol used in a transactional distributed commit.

9.1. Consensus and commit consensus in a distributed setting

In the classic consensus problem, a group of processes in a distributed system have to agree on one value. To this end, they propose each an input value from a fixed set of possible values. A consensus protocol is an algorithm for producing an agreement on one of the input values in a consistent way between all the participating processes.

Definition 9.1.1 (Consensus problem)**.** *Let $P_1, \ldots, P_m$ be m distributed processes. Each processor P_i proposes an input value $p_i \in V$, where $V = \{v_1, \ldots, v_n\}$ is the set of n possible input values. The consensus problem is solved if there exists a decision value v such that*

- *each non-faulty process eventually decides on v, and*
- *$v \in \{p_1, \ldots, p_m\}$.*

As has been shown by Fischer, Lynch, and Paterson [26], it is impossible to solve the consensus problem deterministically in a distributed system if even a single process can fail. However, if no process is assumed to fail, a simple consensus protocol is given by choosing a dedicated leader among the processes whose input is to be taken as the decision value.

In the setting of a distributed commit, even under the assumption of no failures, the distributed consensus problem cannot be solved by defining one GAO to decide on the outcome of the commit. Because transactions send commit requests to different subsets of GAO processes, the set of acceptable input values is possibly different for each GAO. We therefore define a new instance of the consensus problem, tailored towards the commit problem.

Definition 9.1.2 (Commit consensus problem)**.** *Let $P_1, \ldots, P_m$ be m distributed processes. Each processor P_i proposes an input value p_i from its set of input values $V_i \subseteq V$, where $V = \{v_1, \ldots, v_n\}$ is the set of all possible input values. The commit consensus problem is solved if there exists a non-empty set*

$$V' = \{v'_1, \ldots, v'_k\} \subseteq \bigcup_{i=1}^{m} V^i$$

of decision values such that for $i \in \{1, \ldots, m\}$ either

- *$V' \cap V^i = \{v\}$ is a singleton set and process P_i decides on v, or*

- $V' \cap V^i = \emptyset$ *and process* P_i *decides to decline all proposed inputs.*

By this definition, a process can decide on one value at maximum. This choice is uniformly taken by all processes that have this value in their set of input values. In contrast to the classic consensus problem, it is possible that a process dismisses all its input values, because processors are only allowed to decide on values in their input set V_i. Because we require that $V \neq \emptyset$, not all processes may back out of a decision.

In the setting of Decent STM, the GAOs take the role of the processes that have to decide which of the n transactions can commit in a consistent and conflict-free way. The set V_i corresponds to the set of transactions that try to commit a new version to a specific GAO. The decision set V' is then the set of transactions that can commit together in a conflict-free way such that each GAO is committed to by at most one transactions.

The commit consensus problem can also be examined from the perspective of a transaction. Let M_j be the set of GAO whose version are in the write set of transaction T_j for $j \in \{1, \ldots, n\}$. The set of all write sets is defined by $M = \{M_j \mid j \in \{1, \ldots, n\}\}$, and the set of GAOs which receive commit requests is given by $G = \bigcup_{j=1}^{n} M_j$. To solve a commit consensus problem, the protocol has to find a non-empty family $M' \subseteq M$ such that the sets in M' are pairwise disjoint.

The better the coverage of G by M', the more GAOs are updated by the transactions. Achieving a high coverage is desirable for an STM system as this indicates that progress is possible for either expensive transactions with many updates to the state, or many small independent transactions whose write sets span a large number of GAOs. However, getting an optimal coverage is nontrivial.

Theorem 9.1.1 (Set packing)**.** *Let S be a finite set and $\{S_i \mid i \in \{1, \ldots, n\}\}$ be a family of subsets of S. Then, finding the maximum number of pairwise disjoint subsets is an NP-complete problem.*

Proof of 9.1.1: A proof can be found in [43]. □

In a distributed setting, solving the set packing problem on the transactions' write sets requires to gather the information about the elements of the write sets at a central location, contrary to our design so far. As another complication in our setting, the number of transactions that compete for commit is initially unknown. Transactions enter the commit phase as soon as they finish executing the atomic block, and the commit protocol has to adapt to these dynamic updates.

For the commit consensus in Decent STM, we developed a novel decentralized and non-deterministic algorithm which we introduce in the next section.

9.2. Design of a randomized commit consensus protocol

To solve the distributed commit consensus problem from Definition 9.1.2, we propose a novel randomized algorithm. It does not rely on a central instance to decide, it can

deal with dynamic updates to the set of committing transactions, and it is adaptable to incorporate heuristics to increase the commit probability of transactions in favor. For example, a heuristic can be based on the number of objects in a transaction' write set.

In short, the GAOs decide on which transactions may commit based on probabilities. The protocol proceeds in rounds. In each round, every GAO chooses randomly a transaction from the set of transactions that issued a commit request to this GAO. Starting with an equal chance for all transactions, a transaction's probability to commit increases or decreases with the number of positive votes it received in the previous round. The protocol continues until one transaction has received only positives votes from all the GAOs in its write set. Transactions that have been competing with the winning transaction on these GAOs have to abort and restart. Those GAOs that were not part of the transaction's write set can continue with running their protocol.

GAOs and transactions communicate via messages. The types of messages are as follows:

Sent by transaction	Sent by GAO
COMMIT	COMMITFAIL
SUCCESS	POSVOTE
ABORT	NEGVOTE
CONTINUE	

Messages are sent asynchronously, but with respect to each GAO they are consumed in a serialized order. This order does not need to correspond to the absolute time when the messages were send as delays due to network traffic can occur. Also, the order of arrival may be different at each GAO node.

Figure 9.1 shows a transaction's state during the commit phase. After sending a commit request to all the GAOs it wants to commit on, it collects the votes or conflict messages it receives. When all answers have arrived, the transaction decides on success, failure, or continuation. It then communicates its decision to the GAOs. If it decides to continue for another round, it awaits again the votes from the GAOs.

Correspondingly, Figure 9.2 displays the state of a GAO with respect to the commit protocol. In the idle state, GAO is not participating in an instance of the commit protocol.[1] Once a commit request messages reaches the GAO, it checks whether the requested GAO version is eligible, i.e. no intermediate updates to the GAO has taken place since the transaction read it. If a GAO signals a write conflict, the transaction is doomed to fail and sends failure messages to all GAOs to cancel its commit request. If no write conflict has been detected, the transaction is notified with a vote message. The first transaction that announces a commit on a GAO receives a positive vote. While the GAO is then waiting for the transaction's

[1] Read requests are answered at any time, this is, both in the idle state and while the protocol is getting executed.

Figure 9.1. Decent STM: States of a transaction during the commit protocol.

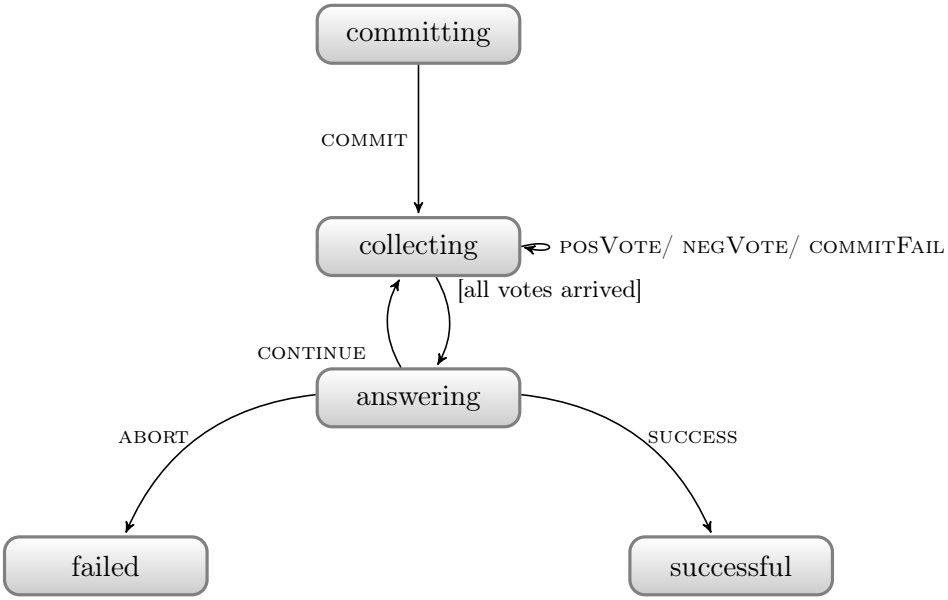
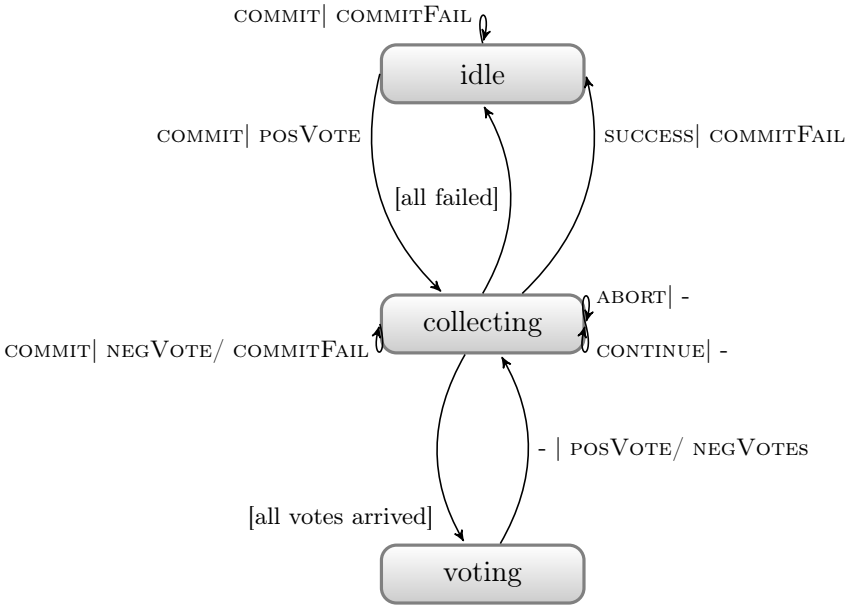


Figure 9.2. Decent STM: States of a GAO during the commit protocol.



reply, other commit requests that arrive at the GAO are similarly checked for write conflicts, but in the non-conflicting case they are answered with a negative vote.

Each transaction needs to answer the votes by sending either an ABORT, SUCCESS, or CONTINUE message.

- In case of ABORT, the transaction is removed from the set of participants. This is done by removing its GAO versions from the set of tentative versions. If the tentative version set is empty, the GAO falls back into the idle state.
- In case of SUCCESS, the commit protocol terminates. The GAO publishes the GAOV which has been committed by the successful transaction by setting it as the current version in the version history list, thus making it available for subsequent read requests. All other transactions participating in the commit consensus are informed about their failure. The GAO then returns to the idle state.
- All participants who have answered with a CONTINUEmessage or have sent a commit request in the meanwhile participate in the voting for the next round. Once all transactions have answered and no transaction issued a success, the GAO chooses one of the pending transactions randomly, based on the probabilities assigned the transactions. This winning transaction is notified with a positive vote, all other participants are notified with a negative vote. The GAO then returns into the collecting state.

A round of the commit protocol is finished when the GAO switches from the collecting state to the voting state, i.e., when all transactions that received a POSVOTE or a NEGVOTE have answered. An instance of the commit protocol is finished when the GAO switches from the collecting state to the idle state.

Next, we have a closer look at how the commit probabilities for transactions are calculated and adapted during the protocol's execution.

9.2.1. Example

Before we explain the randomization in the protocol in more detail, we first sketch its design rationale by means of an example. Consider a case where two transactions, T_1 and T_2, try to commit on overlapping sets of GAOs, namely on T_1 on G_1, G_2, G_3 and T_2 on G_2 and G_3. Figure 9.3 shows the GAOs' internal state together with the transactions' internal state as the protocol progresses round after round.

As before, v_j^i denotes the GAO version that is committed by transaction T_j on the GAO G_i. Assume that the commit requests of both transactions are issued at the same time. Further, assume that due to delays in the network the request for v_1^2 arrives before the one for v_2^2 at G_2, and the one for v_1^3 arrives after the commit request for v_2^3 at G_3. Each GAO answers the first incoming commit request with a positive vote, all following ones with a negative vote. Hence, both transactions receive positive and negative votes in the initial round: Transaction T_1 receives positive votes from G_1 and G_2 and one negative vote from G_3. Transaction T_2 gets

Figure 9.3. Example: Distributed randomized commit consensus protocol.

Probabilities and votes at the GAOs:

	G_1	G_2		G_3		GAOs
	v_1^1	v_1^2	v_2^2	v_1^3	v_2^3	tentative versions
Initial votes	+	+	-	-	+	votes $m_j^i(0)$
1st round	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{1}{2}$	$\frac{2}{3}$	$\frac{1}{2}$	$s_j(0)$
	1	$\frac{4}{7}$	$\frac{3}{7}$	$\frac{4}{7}$	$\frac{3}{7}$	$p_j^i(1)$
	+	-	+	+	-	votes $m_j^i(1)$
2nd round	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{1}{2}$	$\frac{2}{3}$	$\frac{1}{2}$	$s_j(1)$
	1	$\frac{4}{7}$	$\frac{3}{7}$	$\frac{4}{7}$	$\frac{3}{7}$	$p_j^i(2)$
	+	+	-	+	-	votes $m_j^i(2)$
3rd round	success	success	failed	success	failed	

Success probability of the transactions after each round:

	$s_j(0)$	$s_j(1)$	$s_j(2)$	
T_1	$\frac{2}{3}$	$\frac{2}{3}$	1	success
T_2	$\frac{1}{2}$	$\frac{1}{2}$	0	failed

one negative vote from G_2 and one positive vote from G_3. Hence, the success rate in the first round for T_1 is $\frac{2}{3}$, and for T_2 it is $\frac{1}{2}$.

The GAOs are then notified with the success rates ($s_j(0)$ in Figure 9.3). They take these values as success probabilities for the respective tentative versions, and normalize them within each GAO ($p_j(1)$ in the figure). Adapting these probabilities ensures that transactions that were successful in the previous round, are more likely to win the next round and the protocol converges towards a winning transaction.

Next, each GAO randomly chooses one of its tentative versions according to the given probabilities. Assume, GAO G_2 chooses (against the odds) v_2^2, GAO G_3 chooses v_1^3. As GAO G_1 still has only one tentative version, it has no choice and settles for v_1^1. The transactions are again notified with the results from this second round. In our example, their success rates and hence the respective probabilities do not change.

Only in the third round, when all GAOs happen to choose the tentative versions from transaction T_1, T_1 receives only positive votes, whereas T_2 receives only negative votes. After this round, the protocol finishes and discards the tentative GAO versions of T_2. The GAO versions of the successful transaction T_1 are then marked as the current versions of the corresponding GAOs.

From this example, we see that the initial vote has in general a strong influence on the final result. We exploit this fact in Decent STM and let subsequent transactions read the tentative version that has the highest probability at the time when the read request is issued. As we have explained, this does not violate consistency, because

subsequent transactions are only considered by the consensus protocol when all the versions in its read set have been committed successfully.

In practice, consensus is usually reached within the first two rounds. In our example, consensus is reached when G_2 and G_3 happen to vote for the same transaction. The probability for this is given by $(\frac{4}{7})^2 + (\frac{3}{7})^2 = (\frac{5}{7})^2$. Thus, in our example, we have a Bernoulli process with $p = (\frac{5}{7})^2$, which leads to a Poisson distribution with expectation value $1 + (\frac{5}{7})^2 \approx 1.71$ for the number of rounds.

9.2.2. Specification of the protocol

The algorithm for the transactions and GAOs is given in Figures 9.1 - 9.3.

Figure 9.1 shows the part of the execution loop for a GAO which is relevant for the commit phase. Whenever a GAO receives a message from a transaction, it dispatches on the message type and executes the respective actions. Messages are dealt with in a serial way, though the order in which they arrive at the single GAOs can differ between the GAO nodes due to network issues.

The corresponding code for committing transactions is given in Figure 9.2. While the transaction is in an open state, it collects the votes from the GAOs in its write set (see also Figure 9.3), dispatching on its state after all votes or possible failure message arrived.

For a formal discussion of the randomized commit consensus, we introduce the following definitions: As before, let T_j be a transaction for $j \in 1, .., n$ and G_i be a GAO for $i \in 1, .., m$. The write set of a transaction T_j, i.e. the set of versions it wants to commit, is denoted by

$$W_j = \{v_j^i \mid i \in M_j \subseteq \{1, .., m\}\}$$

where M_j is set of the GAOs to which the transactions sends a commit request.

Let $m_j^i(r)$ denote the vote that the GAO G_i is sending to transaction T_j in round r. For a positive vote, we write $m_j^i(r) = +$, and for a negative note, $m_j^i(r) = -$. Further, let $m_j = |M_j|$ be the total number of votes transaction T_j receives, and let

$$m_j^+(r) = |\{m_j^i(r) \mid m_j^i(r) = +, G_i \in M_j\}|$$

be the number of positive votes a transaction received in round t. Similarly, $m_j^-(r)$ denotes the number of negative votes.

The success rate of transaction T_j in round r is then given by

$$s_j(r) = \frac{m_j^+(r)}{m_j}.$$

A transaction decides on state (success, failure, or open) after it received all votes from the GAOs it has sent a commit request.

- A transaction commits successfully if all the votes it received are positive, i.e. $s_j(r) = 1$.

Algorithm 9.1 Decent STM: GAO execution loop - Commit requests.

```
method RUN(GAO g)
    if RECEIVE(t, FETCH, readset) then
        ... // continued from Algorithm 8.2
    else if RECEIVE(t, COMMIT, loc) then
        if loc.pred ≠ g.mostRecent then
            SEND(t, COMMITFAIL)
        else
            tentatives.add(gaov)
            if tentatives.size = 1 then
                SEND(t, POSVOTE)
            else
                SEND(t, NEGVOTE)
            end if
        end if
    else if RECEIVE(t, CONTINUE, gaov, p) then
        tentatives.adjustVotes(gaov,p)
        if txns for all gaov ∈ tentatives replied then
            loc ← choose from tentatives
            SEND(gaov.writtenBy, POSVOTE)
            for all gaov' ∈ tentatives, gaov' ≠ gaov do
                SEND(gaov'.writtenBy, NEGVOTE)
            end for
        end if
    else if RECEIVE(t, SUCCESS, gaov) then
        gaov.pred ← g.mostRecent
        g.mostRecent ← gaov
        for all gaov ∈ tentatives do
            SEND(t, COMMITFAIL)
        end for
    else if RECEIVE(t, ABORT, gaov) then
        tentatives.remove(gaov)
    end if
end
```

Algorithm 9.2 Decent STM: Commit of a transaction.

```
method STMCOMMIT()
    gaoset ← ∅
    for all (g,loc) ∈ writeset do
        gaov ← construct GAOV from loc
        gaoset.add(g,gaov)
        SEND(g, COMMIT, gaov)
    end for
    status ← open
    while status = open do
        posVotes ← COLLECTVOTES
        if status = abort then
            for all (g, gaov) ∈ gaoset do
                SEND(g, ABORT, gaov)
            end for
            abort transaction
        else if status = success then
            register create set
            for all (g, gaov) ∈ gaovset do
                SEND(g, SUCCESS, gaov)
            end for
        else
            for all (g,gaov) ∈ gaoset do
                SEND(g, CONTINUE, gaov, posVotes / writeset.size)
            end for
        end if
    end while
end
```

Algorithm 9.3 Decent STM: Collecting votes.

```
method COLLECTVOTES()
    pos, neg ← 0
    for all (g,loc) ∈ writeset do
        if RECEIVE(g, POSVOTE) then
            pos++
        else if RECEIVE(g, NEGVOTE) then
            neg++
        else if RECEIVE(g, COMMITFAIL) then
            status ← abort
        end if
    end for
    if pos = writeset.size then
        status ← success
    else if neg = writeset.size then
        status ← abort
    end if
    return pos
end
```

- A transaction aborts if it received negatives votes from all GAOs on which it tries to commit, i.e. $s_j(r) = 0$, or when one of its tentative GAO versions has become non-eligible because another conflicting transaction committed successfully at one of the GAOs. In this case, the GAO signals a write conflict.

- When a transaction receives both positive and negative votes, it sends a *continue* message to the GAOs to indicate its wish of participation in another round. This message contains the transaction's success rate $s_j(r)$ of the previous round.

After all transactions that have sent tentative versions for a GAO have replied with a continue message which includes their success rate probability $s_j(r)$ of the previous round, the GAO scales the probabilities for a version v_j^i in the next round $r + 1$ to be

$$p_j^i(r+1) = s_j(r) \left(\sum_{0 \leqslant k < n} s_k(r) \right)^{-1}$$

It then chooses a GAOV randomly according to the scaled probability.

As before, this winner is notified with a positive vote, all other tentative versions with a negative vote. A GAO stops its participation in an instance of a consensus protocol either when it gets notified with a SUCCESSmessage or when all involved transactions have withdrawn their tentative versions.

9.2.3. Correctness

It is trivial to see that the commit consensus fulfills the stability and consistency requirements we stated at the beginning of this chapter: Clearly, a transaction is either successful, pending, or failed, and once it reaches a success or fail state, it remains in this state.

For the non-triviality requirement, the protocol requires that a transaction is setting its state to success only if it received a positive vote from all the GAOs to which it had sent a commit request. On the other hand, each GAO randomly chooses exactly one version from the set of tentative versions in each round. Therefore, only one transaction can receive a positive vote in each round, so that conflicting transactions that try to commit on the same GAO cannot be successful together.

Finally, to see that the system eventually reaches consensus on which transactions have to commit, we have to show that every transaction reaches eventually either a final success or failure state. In a randomized system, an *eventual* consensus implies that each transaction reaches a final state asymptotically almost surely, i.e. with a nonzero probability. Because we assume that the number of transactions and GAOs has an upper bound, there is a nonzero probability that all GAOs from a transaction's write set happen to vote consistently in favor or against a transaction in each round. Therefore, the protocol terminates eventually.

Theorem 9.2.1 (Correctness)**.** *The randomized distributed consensus protocol is non-trivial, stable, consistent, and reaches consensus eventually.*

Further, the protocol part for submitting the initial commit request and the check for intermediate updates is wait-free. This allows a fast commit of updates to non-contended GAOs and a quick restart of transactions that are doomed to fail. In the following rounds, the staged answering of continue messages is lock-free.

The complexity of the algorithm renders a detailed analysis of the consensus protocol difficult. This is particularly due to the varying number of participants in the protocol as transactions can start or finish, as well as drop out or enter a consensus instance in each round. Another complication is given by the non-uniform association of transactions with the GAOs on which they try to commit, introducing complex dependencies between transactions and hence the adaptation of commit probabilities. For an analysis of a simple instance with two transactions and three GAOs, we refer to the end of Section 9.2.1.

9.3. Extensions to the protocol

Using a randomized consensus protocol allows for an easy integration of heuristics for selecting committing transactions. For example, for certain applications, the system might favor long running transactions or those using specific resources. By assigning higher commit probabilities to these favorable transactions, they are more likely to win the randomized selection process. In this thesis, we do not discuss such options, but assume equal chances for all transactions.

As stated before, read requests are answered at any time by the GAO, in particular, also when the GAO is active in some commit protocol. If a concurrently running transaction reads the GAO during such a commit phase, it is given the current version by the algorithm in Chapter 8.3. Another option would be to extend the Decent STM system such that read requests also deliver tentative GAO versions. As with the previously described fetch operation, the versions have to be delivered in a consistent manner for all read GAOs. When a transaction that read some tentative version commits while the election is still going on, its committed versions are inserted below the respective tentative versions (hence the term *tentative version tree*). These children of the tentative versions are handled after their parent version has been elected. If however such a parent version is not elected, all depending transactions must abort without being considered for consensus. To minimize the probability for a collective abort, the GAO answers read requests with the version which has the highest commit probability. If a GAO responds for each depth level always with the same tentative version, the tentative version tree has on each depth level exactly one branch with further branches.

Though we do not consider faulty or even malicious processes, it should be possible to integrate standard techniques such as timing assumptions or other failure detectors into the randomized commit consensus protocol to increase the robustness of the protocol. For example, if a failure detector identifies a failing node, the associated transaction is simply removed from the pool of participants. According to the impossibility result of Fischer, Lynch, and Paterson [26], such a modified protocol provides weaker synchronization semantics than consensus. For applications running on a multi-core processor, partitioning and failure is unlikely, and we therefore concentrate here on the setting without failure and perfect connectivity of nodes.

Chapter 10.

Implementation

In this chapter, we focus on the message-based communication layer and describe the implementations of the Decent STM components for this layer.

10.1. Components

Our implementations of Decent STM comprises components that represent shared distributed memory in the form of GAOs, and components that represent transactions which contain the program logic and operate on the shared memory. Both types of components may reside on different nodes in the distributed environment and exchange messages via dedicated communication channels. Depending on the actual architecture of the system, communication channels are for example implemented with the help of (a-)synchronous network protocols or simulated by shared memory on a multi-core architecture. We experimented with both of these channels as laid out in more detail in Chapter 11.

On a general account, we do not take failing of malicious nodes into account, although we conjecture that our implementation can be adapted with conventional methods to deal with such issues. Further, we assume that all communication operations are performed reliably, i. e. all requests are answered eventually. However, we do not assume a finite upper bound for the transmission delay. Thus, in practice, an underlying protocol can guarantee reliability with the help of retransmissions.

Threads and Transactions An application using the Decent STM environment is in general a composition of several threads running on different processing nodes. A *thread* operates both on local and globally shared data. The globally shared data is used for communication between the threads and is managed by the Decent STM library. The read and write accesses to the shared data therefore must be encapsulated in *transactions*. As described in previous chapters, transactions operate on consistent memory snapshots and perform roll-backs in case on conflicting accesses to globally accessible objects (GAOs).

Runtimes The distributed shared memory is hosted on *runtimes*. A runtime acts similarly to a small memory server. It accepts and answers the transactions' fetch and commit requests on behave of a specific GAO. To reduce the overhead in communication management, a runtime instance is responsible for a number of GAOs.

When sending a commit request, a transaction can thus merge the request messages or the messages with the commit probabilities for all GAOs that reside on the same runtime.

To simplify the protocol, each GAO is affiliated with only one runtime instance. In this thesis, the mapping from GAO to runtime is obtained by simply hashing the GID. More complex routing protocols can be applied to facilitate the migration of objects between different runtime instances. For increased reliability and faster read access, multiple nodes might handle additional copies of the same GAO. For a discussion on these extensions of Decent STM, we refer to Section 13.

System manager To run a program with Decent STM in distributed setting, it suffices to equip the application threads with information on how to connect to the runtimes, then start the runtimes and let the transactions interact with them. For a convenient test and benchmark environment, Decent STM additionally provides a system manager which helps in setting up the communication infrastructure.

The *system manager* is responsible for the set-up and tear-down phase of the Decent STM. It is the only central component in the whole design of Decent STM and could also be implemented in a decentralized way, though this is beyond the scope of the thesis. It synchronizes the initialization of the runtimes with the start of the program. Each thread that wants to obtain access to the globally shared data registers itself with the system manager. The application thread is then equipped with the details on the communication channels that are used by the runtimes. When all runtimes have informed the system manager about their successful initialization, the threads that execute transactions are allowed to proceed with the program logic.

When a thread has finished its execution, it unregisters with the system manager and shuts down its communication subsystem. Similarly, after all threads have unregistered with the system manager, it issues the shut-down of the runtimes and hence of the whole system.

Messages The communication between the components of Decent STM is based on messages. A *message* consists of a message header, containing the message type and information about sender and receiver, and the message body. Depending on the type of message, the message body contains data such as lists of version numbers and GIDs for the fetch request, serialized objects for obtaining or committing GAO versions, votes, or success probabilities.

10.2. Interface

The complex interaction between the components of a Decent STM system is largely hidden from the programmer. To program with atomic blocks and transactional data, Decent STM offers the programmer an interface for object-oriented STM as given in Listing 10.1.

Listing 10.1 Decent STM API for Java.

```
public interface Txn {
    public <T> GID<T> create(LOC<T> obj);
    public <T> LOC<T> read(GID<T> obj)
        throws StmReadConflictException;
    public <T> void write(LOC<T> obj);
    public void commit()
        throws StmWriteConflictException;
}
```

Further, the API enforces a distinction between local and globally accessible data through the type system. A GAO is accessible through its global identifier (GID). A transaction operates on local object copies (LOC) of the GAO, which are thread-local replica of some specific GAO version. GIDs and LOCs are parametrized with the type of object that they contain.

To create a GAO, a local object is promoted to become the initial version of the new GAO. The **create** method yields a new unique GID for this GAO.

For reading the object that is referenced by a GID, the STM system returns a local object copy of the GAO. This copy can then be modified by the transaction. With a call to the **write** method, the modified version is added to the write set of the transaction and later committed to the shared memory.

A transaction is initialized and started via a call to the constructor. Its scope is limited by a call to **commit**. When a read or write conflict is detected at run time, the transaction aborts by resetting its state and throwing the exception **StmReadConflictException** or **StmWriteConflictException** that both extend the **StmException** class. These exceptions are passed on to the application level where the program logic determines whether a transaction is restarted or aborted.

Listing 10.2 shows an example usage of the API. A globally shared integer object `counter` is read and incremented atomically. To restart the transaction in case of a conflict, a while loop surrounds the actual code of the transaction and is only left after a successful commit.

In addition to the API in Figure 10.1, the transaction class provides static methods for setting up and shutting down the STM layer. The **initialize** method sets up the communication infrastructure for the current application thread. Transactions that are executed by the same thread share the same communication channels. For example, in a distributed implementation of Decent STM it suffices to open the communication channels between transactions and runtimes once and later reuse these channels. The counterpart to the initialization is the **shutdown** method which closes all open channels and discards the transactional meta data.

Listing 10.2 DecentSTM: Explicit usage of the interface.

```
int increment(GID<Integer> counter) {
    Txn txn = new Txn();
    while (true) {
        try {
            int result = txn.read(counter);
            txn.write(counter,result + 1);
            txn.commit();
            break;
        } catch (StmException e) {}
    }
    return result;
}
```

10.3. Preprocessing the code

Using Decent STM in form of a Java library, the programmer needs to specify the begin and end of an atomic block by instantiating a transaction object and calling API methods, taking care that each atomic block is properly opened and closed, and that transactional exceptions are caught and failing transactions are restarted. Further, each read and write access to shared data must be wrapped into a method call which passes the request to the corresponding transaction object. This procedure is tedious and error-prone.

To simplify the development of STM applications, we developed a tool for static transformation of Java byte code, called JTransactifier. The implementation employs the ASM byte code framework [15]. With JTransactifier, the programmer declaratively specifies two annotations instead of manually transforming the code to incorporate the calls to the STM API:

- Each class whose instances are used as GAOs are annotated with **@GAO**.
- Each transaction is extracted into a method and annotated with **@Atomic**.

The remainder of this section explains the transformation, and shows how the Decent STM API can be used without the JTransactifier.

The JTransactifier tool takes a jar file with Java byte code containing the STM annotations and transforms the code with respect to the annotations. In a first phase, it retrieves all classes annotated with **@GAO**:

```
@GAO
class C {
    C (T1 v1,...,Tn vn) { ... }
    ...
}
```

Each of these classes is turned into a subclass of the **LOC** class, either directly or by transforming one of its super classes. A direct translation of the class yields:

```
class C extends LOC<C> {
    C (T1 v1,...,Tn vn) { super(); ... }
    ...
}
```

For transmissions of GAOV between nodes, all object instances that are used as GAOs need to be serializable. These classes therefore extend the abstract class **LOC**, which in turn implements the `Serializable` interface.

In addition to the class transformation, every access of fields of an objects of such an annotated class is rewritten to use the read and write method of the transactional interface.

- For an object `obj` of type `C extends LOC<C>`, a read access `obj.x` is transformed into `txn.read(obj).x`.
- Modifying a field of a GAO `obj.x = ...` requires access to the corresponding LOC:

  ```
  C loc = <C> txn.read(obj);
  loc.x = ...;
  txn.write(loc);
  ```

- Newly allocated instances of GAOs are also registered with the transaction by inserting `txn.create(obj);` after the constructor call of the `obj` object.

Next, all references to classes of global objects are replaced by GIDs. This decouples the dependencies between single LOCs, allowing the construction of an optimal snapshot of the shared memory. A GID is reference pointing to a GAO. It contains the information that is necessary to retrieve a local copy of the corresponding object.

Finally, the methods with **@Atomic** annotations are wrapped with an exception handler as shown in Figure 10.1, thus enforcing restarts of the transaction until it succeeds. The current transaction object is stored in some static field of the **Txn** class, using a thread-local variable, to make it available when calling atomic methods in a nested manner. As can be derived from the code snippet, the JTransactifier implements flat nesting. Calling an atomic method while being already in the scope of an transaction executes the nested method on the same transaction object. Only when the outermost transaction commits, all the changes are published to shared memory.

In combination with the library based Decent STM implementations, the JTransactifier tool alleviates the programmer from the work of manually rewriting every access to GAOs. Furthermore, it offers a declarative specification of what comprises an atomic block and allows code optimizations based on this information.

Figure 10.1. JTransactifier: Transformation of atomic methods.

Before transformation:

```
@Atomic
T m (T1 v1,...,Tn vn) {
    ... // method body
}
```

After transformation:

```
T m (T1 v1,...,Tn vn) {
    T result;
    Txn txn = Txn.getCurrentTxn();
    if (txn == null) {
        txn = new Txn();
        while (true) {
           try {
              result = m(txn,v1,...,vn);
              txn.commit();
              break;
           } catch (StmException e) {
             ...  // e.g. counting the aborts
           }
        }
    } else {
        result = m(txn,v1,...,vn);
    }
    return result;
}

@Atomic
T m (Txn txn,T1 v1,...,Tn vn) {
    ... //transformed method body
}
```

Our experiences with JTransactifier have been very positive. It considerably simplified the creation of benchmarks and case studies, in particular when parallelizing existing code bases with STM. For example, we transformed a Java implementation of the vacation benchmark from the STAMP benchmark suite. The code has been transactified by simply annotating two classes with **`@GAO`** and 16 methods with **`@Atomic`**.

The current limitations of JTransactifier lie in the static, class based separation between local and globally shared data. The type system does not allow to operate on local instances of GAOs outside of transactions, thus prohibiting popular programming schemes like privatization. A more sophisticated version of JTransactifier could leverage this limitation to some extend by employing data flow analyses in order to determine the set of objects that are actually shared.

So far, JTransactifier is not able to transform classes from the Java standard libraries as these do not carry the required annotations. A further complication here is given with native libraries. For installment in a production environment, the JTransactifier can further be extended, for example, to restrict transactions to include solely reversible actions on shared memory. Also, it can be optimized by incorporating data flow analyses to determine which objects are actually shared between threads and have to be transferred to the shared memory.

Chapter 11.

Evaluation

The distinguishing feature of Decent STM is its ability to operate in a fully decentralized setting. To assess this feature and evaluate the algorithm's performance, we created a simple in-memory reference implementation and a distributed version with communication based on TCP/IP. This chapter presents an evaluation of both versions discussing the impact of version history lists and the randomized commit protocol.

11.1. Decent STM with shared-memory synchronization

The Decent STM implementation with shared-memory synchronization simulates a distributed setting. Runtimes are running on separate threads, concurrently to the applications threads and transactions. Communication between the runtimes and transactions is done by message objects which are inserted by the sender into a synchronized message queue at the receiver's side. Using conventional shared-memory synchronization for information exchange does neither entail establishing dedicated channels nor serializing the messages to byte arrays.

To keep the overhead of operations in Decent STM low, the system implements snapshot isolation as default semantics for transactions. In contrast to serializable transactions, snapshot isolation requires consistency checks for intermediate updates for the elements in the write set, not for the read set. For compatibility with STM applications requiring serializable transactions, the implementation of Decent STM offers the possibility to switch to the stricter semantics of opacity. A straightforward modification to obtain opacity is to extend the commit-time check for intermediate updates to the elements in the read set.

For the in-memory implementation, we chose a different approach. For making the snapshot isolation protocol serializable, we applied the algorithm by Cahill, Roehm and Fekete [18]. It is based on some theoretical results of Fekete et al. [24] who have analyzed dependency graphs for multi-versioned transactional systems. A history is serializable if the corresponding graph, showing read-read, read-write, and write-read dependencies, is cycle-free. They further showed that any cycle in the graph due to snapshot isolation must have two read-write dependency edges that occur consecutively, and further, each of these edges is between two concurrent transactions. To detect these cycles, each transaction object is extended with two boolean values indicating the read-write dependencies to concurrently running

transactions. The algorithm is particularly suitable for the in-memory implementation because transactions set these bits in their siblings' meta data, thus requiring access to these fields. In practice, the algorithm yields only a small number of false positives while keeping the overhead considerably small.

We evaluated our implementation with several applications on a Dual-Quad-Core AMD Opteron running Java 1.6. The runtimes with the GAOs consisted of eight background threads. Each thread handled an equal set of GAOs. The assignment of the GAOs was random based on the objects' hash code and did not reflect the relations between the GAOs. Each of the figures shows the average of 10 runs.

Figures 11.1 and 11.2 present the results of micro benchmarks with a red-black tree and an AVL tree. In these benchmarks, the tree itself and each node is represented by a GAO. The tree is initially filled with 100 elements in a setup phase. (This phase is not represented in the figures.) We perform a total of 8000 operations on the tree, of which

- in Figure 11.1(a) and 11.2(a), 10% insert, 10% delete, and 80% look up an element;
- in Figure 11.1(b) and 11.2(b), 40% insert, 40% delete, and 20% look up an element.

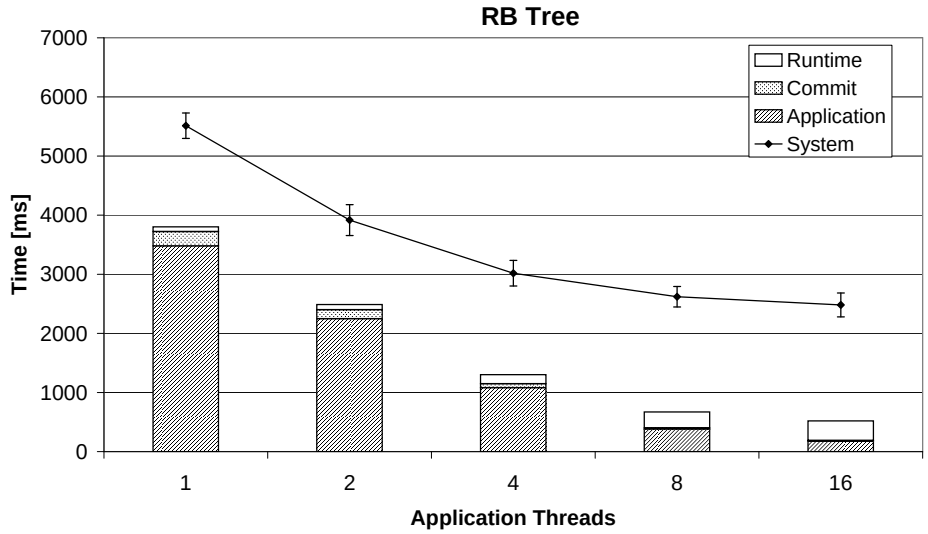
These elements were randomly chosen integers ranging from 0 to 1024, so that we created sufficient contention to stress our STM algorithm.

The figures show the total system time and the time spent in the benchmark itself. For the latter, we differentiate between the time spent in the application threads, the commit phase, and the runtime overhead:

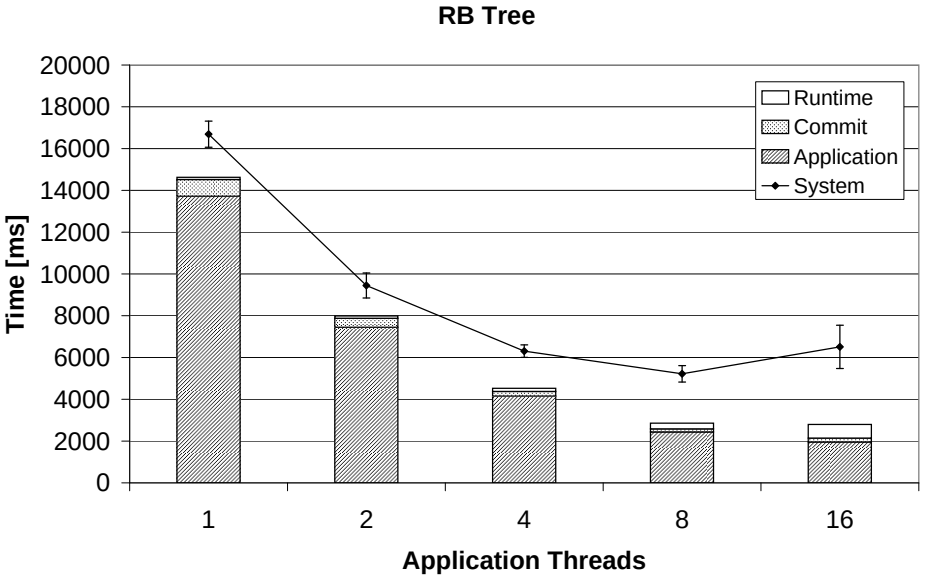
- *Runtime* is the absolute time spent in the GAO management for handling read requests, commit requests, and voting messages;
- *Commit* is the absolute time that the transactions spent in the commit protocol;
- *Application* is the absolute time spent in the application, including the overhead for transactional reads and writes at the transaction's side.
- *System* corresponds to the time that elapsed between start and end of the benchmark run.

For the benchmark of the RB Tree (Figure 11.1), increasing the number of application threads leads to a speed-up of 1.6 (2 threads), 2.9 (4 threads), and 5.6 (8 threads) with a low percentage of update operations, and 1.9 (2 threads), 2.8 (4 threads), and 5.0 (8 threads) for a high percentage of update operations. Similar results are given for the AVL tree benchmark (Figure 11.2), with a small improvement for the setup with 80% updates, where a speedup of 5.64 is obtained with 8 threads.

Figure 11.1. Decent STM: Micro benchmarks - RB Tree.

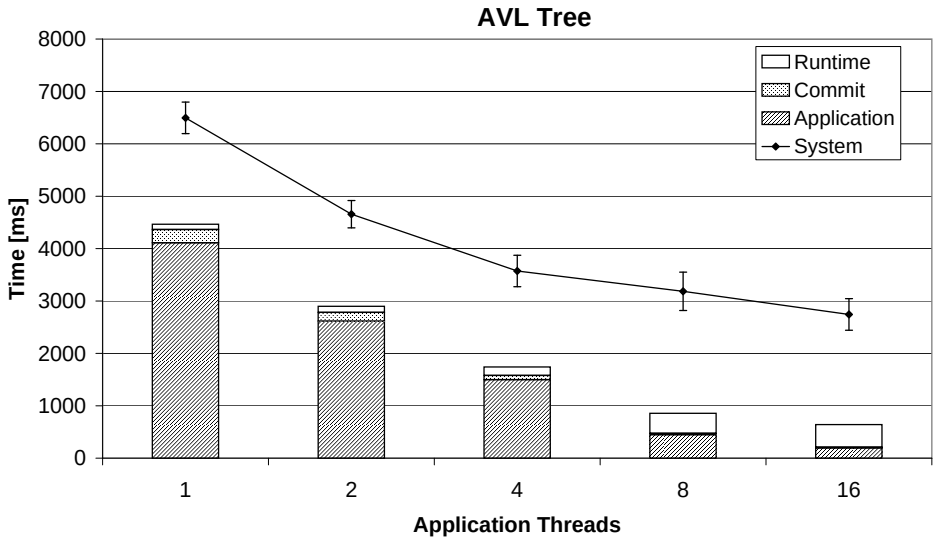


(a) RB tree, 20% update.

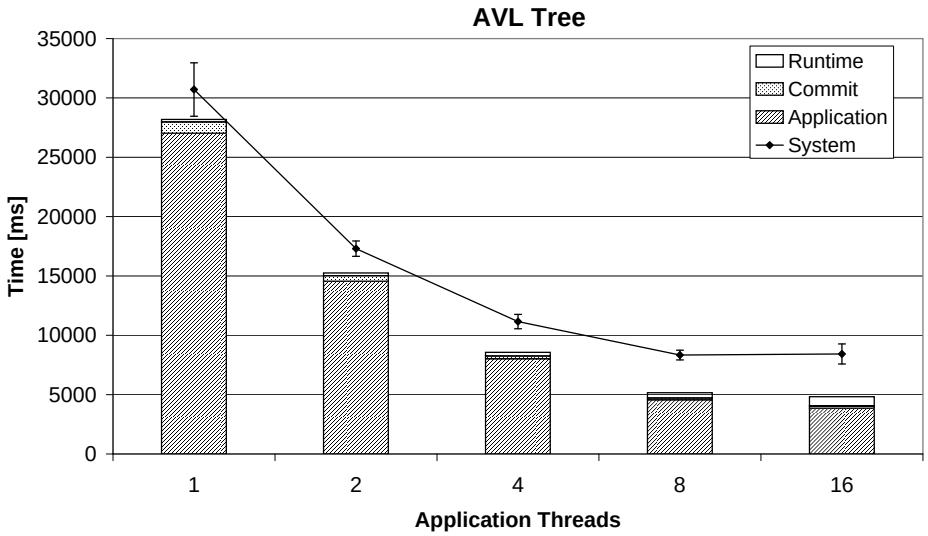


(b) RB tree, 80% update.

Figure 11.2. Decent STM: Micro benchmarks - AVL Tree.

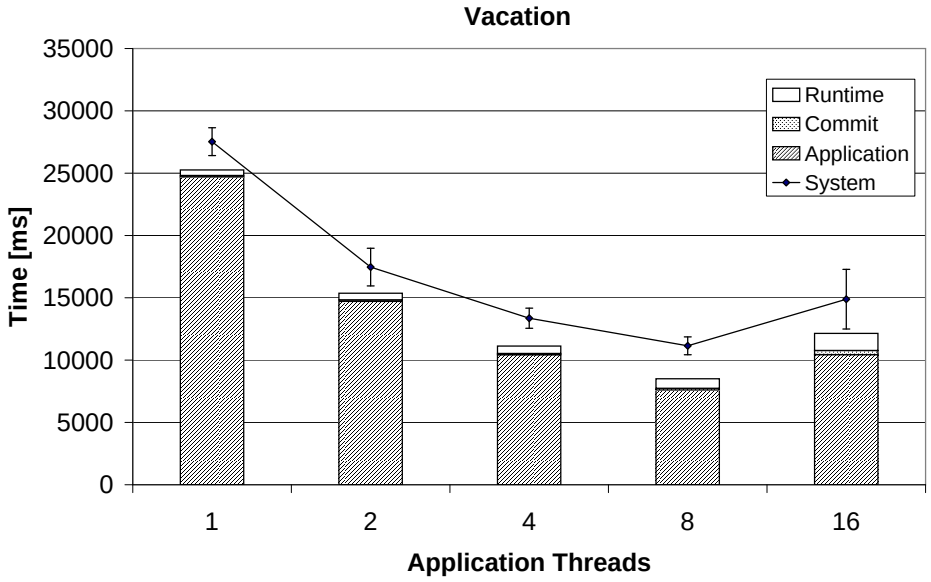


(a) AVL tree, 20% update.



(b) AVL tree, 80% update.

Figure 11.3. Decent STM: STAMP Benchmark - vacation.



The figures show a good scalability for the benchmarks on the tree data structures, in particular for the benchmarks with a high update rate. They underline that a substantial part in the execution of the infrastructure and the application can run in parallel, and that Decent STM offers support to utilize the available parallelism.

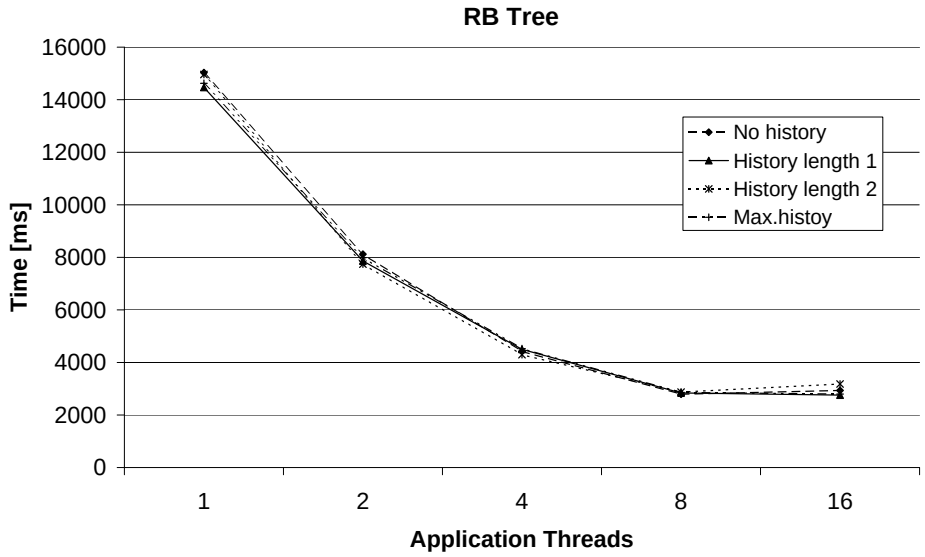
As expected, the performance decreases when we deploy more than eight threads, due to a disproportion between available processor cores and threads. The overhead is introduced by thread switches and rescheduling done by the operating system.

We attribute the difference between the system time and the benchmark time to the operating system overhead as well as the setup and tear down of the benchmark, including the IO activities that were caused by writing log files during the benchmark. In all applications and runs, the time spent in the application dominates by far the runtime and commit time overhead.

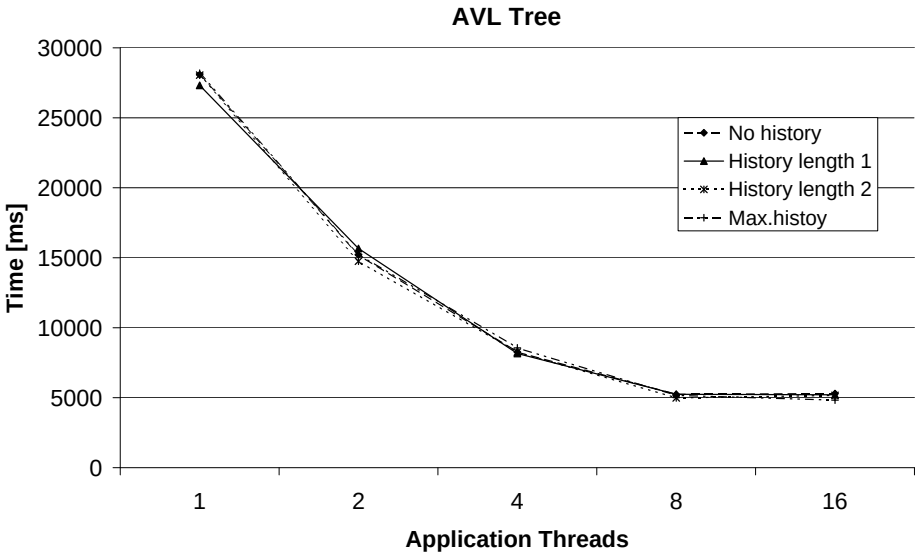
Figure 11.3 presents the results from running the vacation benchmark from the STAMP benchmark suite [19] with the parameters -n2 -q90 -u98 -r2024 -t4096. The customers and reservations are represented by GAOs, all other data is thread-local or read-only. Again, the figures show that Decent STM offers a performance gain when increasing the number of application threads. However, as the vacation benchmark induces a high likelihood of conflicts between concurrently running transactions, only a speedup of 3.1 is reached when distributing the total workload over 8 application threads.

Furthermore, we explored the influence of the length of the committed versions history list on the performance of the application. To this end, we analyzed the run times and conflicts for the tree benchmarks in the 80% update case (as described above). Figures 11.4(a) and 11.4(b) show the system times for version histories of

Figure 11.4. System time for micro benchmarks for different version history lengths.

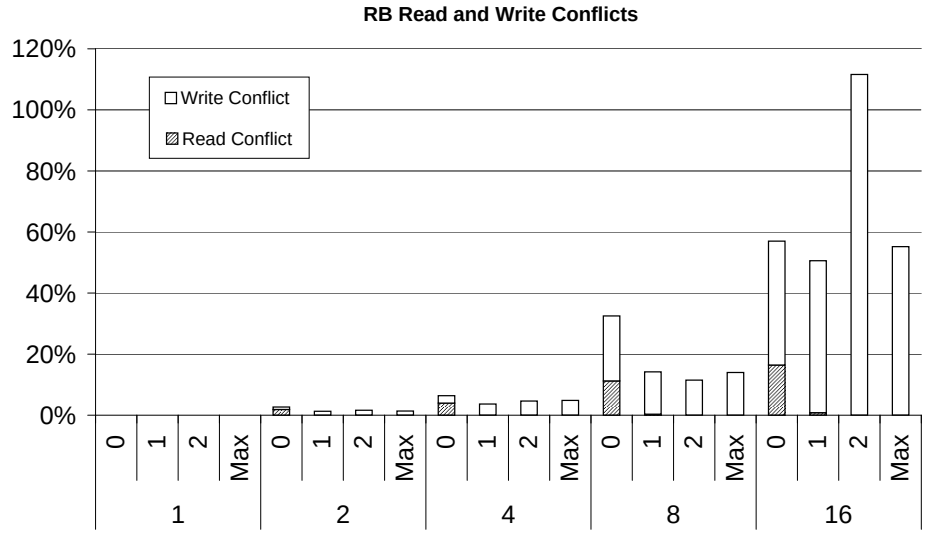


(a) RB tree.

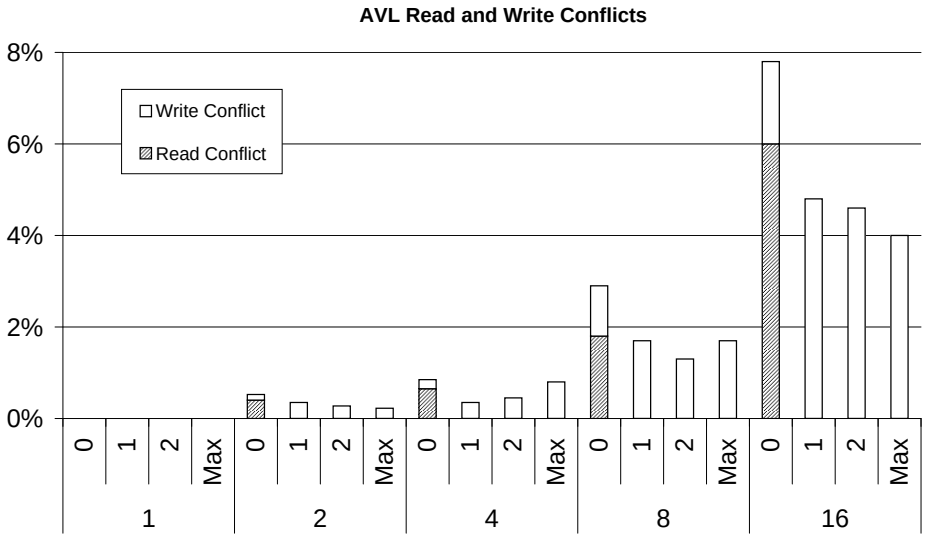


(b) AVL tree.

Figure 11.5. Conflicts for micro benchmarks.



(a) RB tree



(b) AVL tree.

length 0 (no version history), 1, 2, or max (all versions are used when resolving read requests). The system time differs only slightly between the different strategies. The reason becomes apparent when analyzing the number and kinds of aborts.

Figures 11.5(a) and 11.5(b) show the read and write conflicts for the respective benchmark. For the x-axis, we differentiate between different length of the version histories (0, 1, 2, all versions are kept). The figure shows then the percentage of transactions that aborted due to a read of write conflict when compared to the number of committing transactions. A percentage of more than 100% can be reached if transactions have to abort multiple times, and the total number of aborts is higher than the number of commits.

When using just one thread, unsurprisingly, no conflicts occur. When increasing the number of threads, the number of conflicts, especially write conflicts, increases significantly. Almost all read conflicts occur when no version history is provided. In most cases, the version history is only used up to a depth of one. All further entries are apparently not needed for responding to read requests.

The RB tree benchmark in Figure 11.5(a) shows a very high abort rate when compared to the AVL tree benchmark, although the RB tree application is in the presented case almost twice as fast. The high abort rate induced by write conflicts is mainly due to the RB tree being a faster data structure. A closer look at the data revealed that it suffers from multiple aborts in a row as the time needed for restart and re-calculation is less than the time for resolving write conflicts in this case.

This shared-memory version of Decent STM implements the garbage collection of no-longer accessible object versions from Section 8.4. The pruning of the history lists in done while the runtime is waiting for messages to arrive. For the workloads and benchmarks we investigated, neither the garbage collection process nor the reduced memory footprint had an influence on the performance.

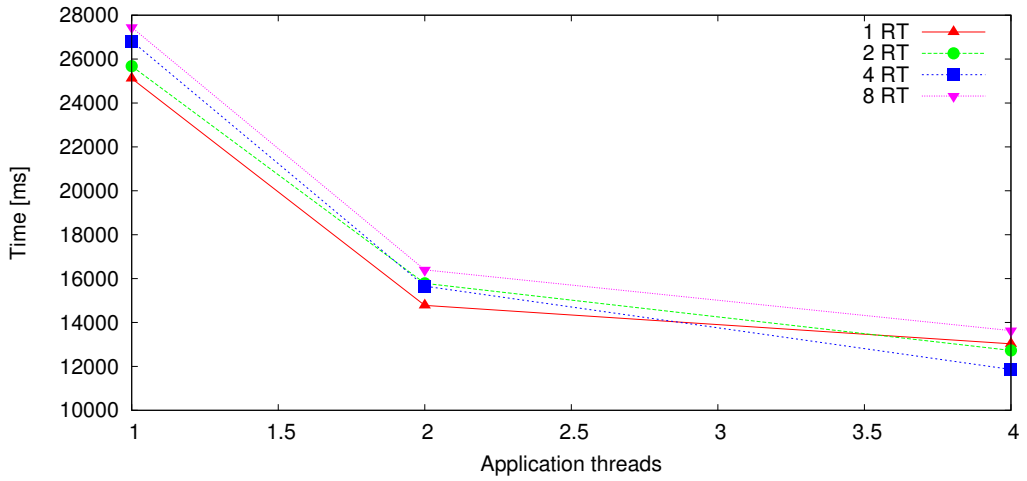
11.2. Decent STM in a distributed setting

The Decent STM version for distributed environments differs in several respects to the implementation described in the previous section.

To provide an easy setup and tear down, this Decent STM version provides a system manager which coordinates the connection to and disconnection from the system. The programmer has to specify in a configuration file where the system manager is hosted. The system manager sends the network addresses of the runtimes to the application threads who want access the shared memory. Each connection that is established between a transaction and a runtime is stored with the thread running the transaction for later reuse. Only when the thread disconnects from the shared memory, the connections are closed.

Messages are serialized into plain byte arrays and deserialized into objects at the receiver's side. This procedure adds a considerable overhead to applications, but cannot be avoided in the distributed setting. One optimization that we used to reduce the number of messages is to combine the messages to GAOs that reside

Figure 11.6. Decent STM: Varying the number of runtimes.



on the same runtime for the commit protocol. Further, each connection between threads and runtimes is initialized once and reused for all transactions running within the scope of a thread.

For the consistency check at the commit, we implemented the standard solution to obtain serializability of transactions, that is, all GAOs in the read set of the committing transactions are checked for intermediate updates. The messages for this consistency check are sent in the first phase of the commit and also combined for each GAO.

The number of runtimes has some influence on the performance of the full system. Figure 11.6 shows the micro benchmark featuring the AVL tree on the distributed version of Decent STM. The tree can contain up to 1000 elements and is initially filled with 500 elements. Then 1, 2, or 4 application threads concurrently insert (10%), remove (10%), or find (80%) keys in the tree, accessing on average approximately 8 GAOs in a transaction. The GAOs containing the tree and its nodes are equally distributed over 1, 2, 4, or 8 different runtime instances. The benchmark was carried out on a MacBook Pro, hosting an Intel Core i7, 2 GHz, with 4 processors. To reduce the influence on network noise, all Decent STM components resided on the same machine, communicating over different ports that are bound to the loop back device.

When having one application thread, maintaining multiple communication channels to different runtimes yields a performance penalty. Combining all messages to the GAOs into one, thus reducing the communication overhead, is faster than splitting the work load during commit over several runtimes to execute them in parallel. Only when increasing the number of threads which execute transactions, it pays off to partition the shared memory. In the case of 4 application threads, using also 4 runtimes is optimal. However, as the line for 8 runtimes indicates, the performance

suffers when starting too many runtimes. This holds especially in the setup we used for this micro benchmark because the components share the processing units.

Part III.

Related work and Conclusion

Chapter 12.

Related Work

Transactional Memory has a huge design space which is investigated by numerous researchers. In addition, research on transactions in data bases as well as concepts from operating systems provide a fertile ground from which many TM ideas have re-emerged.

12.1. STM and irrevocability

Most closely related to Twilight STM are STM implementations which provide irrevocable and inevitable transactions.

Welc et al. [78] propose an irrevocability mechanism to support flexible contention management and the execution of non-reversible actions (I/O, system calls, pre-compiled libraries, ...) within transactions. To ensure safety, they use a protocol with single-owner read locks. A transaction becomes irrevocable by executing a special statement that tries to acquire locks on the read set so far and all upcoming reads. To avoid deadlocks, this approach enforces that only a single irrevocable transaction can run at a time. The system is implemented as an extension of the Java-based McRT-STM and uses dynamic method dispatch to enforce the correct usage and interaction with other language constructs.

Similarly, Spear and coworkers [70] compare five mechanisms which all require that at most one irrevocable transaction runs at a time and that these transactions do not abort (they use the term inevitability instead of irrevocability).

Twilight STM introduces also a notion of irrevocability. I/O operations can be safely integrated into transactions after the programmer compensates for possible inconsistencies. The main difference to the aforementioned systems is that Twilight STM permits arbitrarily many transactions with possibly non-reversible actions to run concurrently.

Ziarek et al. [80] propose a framework where Java's synchronization monitors can be freely mixed and combined with atomic blocks. Their system optimistically tries to execute all concurrency primitives transactionally. In situations where such an execution is infeasible, the implementation irrevocably switches the execution of the concerned critical region to monitors as specified in the original program and to a global locking approach for user-defined transactions. Twilight STM does not allow an arbitrary mixing of synchronized and atomic blocks. It restricts the use of

non-transactional synchronization primitives to the twilight code and never reverts to global locking. The default case is transactional execution.

Welc and coworkers [77] propose a dual-mode implementation of monitors for Java which switches at run time between a lock-based implementation and a transactional one. The switch is based on the level of contention where high contention triggers the use of the transactional implementation. To integrate irrevocable actions, they rely on the same mechanism as in the previously described paper [78]. Here, mode switching is handled transparently to the programmer. Twilight does not switch modes but offers integration with lock-based code in the twilight code. It further reduces the conflict potential of transactions by allowing to inspect and repair read conflicts.

Harris [33] proposes a mechanism to integrate exceptions and side effects with transactions. The proposal relies on a transaction being able to register external actions that execute at commit-time of the transaction.

Smaragdakis and coworkers [68] integrate side effects in a transaction by essentially committing before and resuming the transaction afterwards. Volos and coworkers [75] propose a system call interface for use inside transactions. They perform error checking as early as possible and defer the main task of the system call to commit time. They rely on sentinels to manage concurrent access to resources.

12.2. HTM approaches for irrevocability

Although our work is a software-only approach, there are quite a few related HTM-based efforts that suggest mechanisms to execute irrevocable operations inside a transaction.

Blundell et al. [12] simulate a hardware TM design that supports I/O and system calls within transactions. They allow only one unrestricted transaction, which can perform I/O and system calls, running concurrently with multiple restricted transactions, which cannot perform I/O. This choice limits concurrency.

Moravan et al. [55] simulate a nested TM in hardware. Their transactions may contain "escape actions" that execute code outside the transactional scope. To retain the transactional appearance, escape actions can register actions to run on a successful commit or on restart (compensating actions). Such commit and compensation actions are also needed in their implementation of open nesting of transactions.

McDonald and coworkers [52] define an instruction set architecture for HTM. Their architecture includes commit handlers and violation handlers that can continue a transaction in an arbitrary way after a conflict. However, only the validated outcome of a transaction is visible. There is no provision to compare previously read values with the current ones.

Volos et al. [74] analyze the problems of using locking operations with TM. They implement transaction-safe locks on top of a HTM. Their design includes commit actions to obtain locks and compensating actions to release them. Further tasks can be scheduled at conflicts and escape actions are also supported.

12.3. Conflict avoidance

To increase performance of STM applications, there are several proposals to avoid conflicts between concurrently running transactions.

Herlihy et al. [39] introduced early release as a technique to avoid conflicts on non-significant memory locations [15]. It allows the programmer to remove elements from a transaction's read set if intermediate updates on these locations by other threads do not change the program's semantics. This decision is subtle, as later re-reading of a variable can lead the transaction to operate on an inconsistent memory snapshot. Twilight STM simplifies the reasoning about correctness by enforcing consistency for each transaction.

Harris and Stipić [37] implement the concept of an abstract nested transaction (ANT) in STM. Failure of an ANT does not cause failure of the enclosing transaction. Instead the ANTs are retried when the enclosing transaction is ready to commit. Side effects like I/O and system calls are disallowed inside ANTs. ANTs can be implemented with the Twilight API by performing potential re-execution of computations in the twilight code.

Ramadan and colleagues [60] enable conflicting transactions to commit. They introduce the notion of dependency-aware TM where a value written by one transaction is forwarded to another transaction reading the same variable before the actual commit, thus avoiding a conflict between the transactions and achieving significant speedups. Twilight STM can also deal with this kind of conflicts by first letting the writing transaction run to completion and then relying on twilight code to fix the conflicting read in the other transaction.

Shapiro et al. [66] investigate how to construct systematically replicated container data types that ensure eventual consistency. The convergence to a consistent state when merging the replicas is ensured by the commutativity of operations. In a similar fashion, Burckhardt, Baldassin, and Leijen [16] apply different merging strategies to incorporate modifications on local object copies to the global state. Depending on the kind of data, the changes are either accumulated or compete with each other, the last patch applied being the winning one. Twilight STM allows to define such strategies in the twilight zone. As the fine-granular conflict detection and repair on word level is rather tedious, it is an interesting open problem to extend the Twilight STM with high level representations of data modifications. For example, the actual operations on a container data type can be stored in an abstract way as part of the actual data structure.

12.4. Semantics of Transactional Memory

Weikum and Vossen [76] include a comprehensive overview on theory and practice of transactional systems. Although their work is based on databases, the presented results relate to all transaction-based execution environments. They differentiate in detail between several notions of serializability, and give soundness proofs for all

major commit protocols.

For STM, Single Global Lock (SGL) semantics [53] has been suggested to simplify reasoning about strong and weak atomicity. SGL provides an intuitive and simple STM semantics, though most STM algorithms do not implement this strong semantics as it impedes scalability.

Opacity was introduced by Guerraoui and Kapalka [31] as a correctness criterion for transactional memory. They also show how opacity can be efficiently implemented for different relaxed memory models [30].

Jagannathan and coworkers [41] specify a formal system for transactions with nesting implemented by a versioning and a locking algorithm. They do not model aborts, but stuck executions are implicitly rolled back. They show that the presented algorithms implement serializability.

Abadi and coworkers [1] formalize the semantics of the Automatic Mutual Exclusion (AME) programming model. Similarly, Moore and Grossman [54] provide a formal model with small-step operational semantics for an impure functional language. Both works focus on the treatment of memory locations inside and outside of transactions, and in which cases the notion of weak and strong atomicity coincide.

Doherty and coworkers [22] give a formalization for transactional memory in terms of an I/O automaton. Their specifications are of different granularities and aim to for machine-checked correctness proofs of implementations.

Our work is partially inspired by Lipton's work [50] on a reduction theory for proving properties of concurrent programs. His main idea was to identify certain statements that may be moved to the left or to the right in the trace of an interleaved execution. In particular, he establishes that lock acquisition can always be moved to the right over statements executing in another thread, whereas lock release is a left mover. The commonality is that we are also reordering traces to prove isolation properties, but the difference is that we consider a transactional framework which also includes explicit transaction aborts.

Riegel et al. [63] transfer snapshot isolation semantics from the database domain to STM. In their implementation, a transaction's read set is taken from a consistent memory snapshot based on a time stamp given a priori. In contrast, Decent STM incorporates updates into a transaction's memory snapshot if this does not violate the snapshot's consistency, i. e. Decent STM gives always the latest consistent versions.

Fekete and coworkers have investigated in a number of publications the particularities of multi-version transactions with snapshot isolation in databases. In a seminal theory paper [24], they identify characteristics of applications that preserve data integrity under different isolation levels by showing that these applications have cycle-free dependency graphs. Similar in idea, though different in technique, our work on opacity and snapshot isolation for singe-version STMs detects data dependencies and conflicts in effect traces. Building on the theoretical results [18], Fekete and co-workers devise a simple algorithm for serializing the transactions in a snapshot isolation system. It yields only a small number of wrongly identified conflicts and the implementation requires only small changes to the transactions.

The algorithm is particularly suitable for the in-memory implementation because it requires transactions to directly access the state of their sibling transactions. We implemented this algorithm for offering serializable transactions in a shared-memory version of Decent STM. It is still an open question to what extent the results on weaker isolation levels in databases apply to STM as workloads differ substantially between database applications and concurrent applications.

12.5. STM in Haskell

Harris and coworkers [35] give a report on an implementation of STM in Haskell. They introduce the STM monad, which is a special type of computation inside an atomic section. This design enables a clean separation between unrestricted computations outside the STM monad and which restricts operations in an atomic section to reads and writes to transactional variables. There is a special *orElse* operation that enables the specification of alternatives in case of failing transactions. But these alternatives cannot preform repairs as in the Twilight approach because all alternatives must be consistent with respect to the start of the original transaction.

This work has been extended to also cover data invariants [36]. These invariants impose conditions that delay the restart of a transaction until the conditions become true.

12.6. STM in distributed settings

Many popular STMs, such as TL2 [20], SkySTM [49], LSA [62], or McRT-STM [65], use blocking synchronization barriers; or they rely on centralized components such as version counters. Both make them unsuitable to be transferred to a (truly) distributed setting.

There is also a variety of non-blocking implementations, e. g. [39] or [34], which propose techniques to reduce contention and to avoid serialization due to bottlenecks. These systems provide higher scalability than the blocking systems but introduce a higher runtime overhead. Decent STM is also a non-blocking system. It reduces the entailed runtime overhead by offering non-conflicting reads. Moreover, its underlying data structures are specifically designed for a fully decentralized setting.

ClusterSTM [13] is an STM design for high performance computing on very large-scale commodity clusters. In contrast to our system, they provide a low-level API which is supposed to get integrated into some domain specific language for high productivity computer systems, and thus poses a great burden on the programmer. Following their reasoning about the design space, we also try to minimize communication overhead and aim for both appropriate placement of shared memory and execution frames on appropriate nodes in the future.

Manassiev et al. [51] apply STM to a distributed setting. Unlike our system, Distributed Multiversioning uses replicas of the shared memory on each network

node in combination with a distributed shared memory consistency protocol. In our system, we rely on one physical copy of each shared object, though caching and redundancy copies may lead to multiple instances. This data is immutable for the major parts, as described in Section 8.1.

Kotselidis et al. [46] have implemented an STM framework for clusters where a master node serves as a global data store. They evaluated several coherence protocols, one of them decentralized (Transactional Coherence and Consistency, TCC). Since all worker threads are involved in this protocol, it leads to substantial overhead. They also show that leases provide a bottleneck when the application entails high contention. Our protocol involves only objects in the write set. Thus non-interfering transactions can commit concurrently.

Riegel et al. [64] replace version counters with real-time clocks. Unfortunately, this requires both hardware support and a synchronization protocol to ensure bounded deviation of the timers.

Distributed systems like Telex [71] introduce the notion of optimistic execution and rollback to high-level software development. These systems often resemble or incorporate databases and offer application-specific support whereas we focus on a generic approach to be incorporated into the memory management of virtual machines, for example.

12.7. Multi-versioned STMs

Reed [61] proposes the use of multi-versioning for handling decentralized data. In line with Decent STM, he retrieves a version corresponding to the memory snapshot taken so far when reading a variable. In contrast to our approach, he utilizes synchronized timers to obtain a consistent memory snapshot.

Cachopo et al. [17] implement a multi-versioning scheme with versioned boxes to store the value history of a variable in combination with a global commit counter.

Aydonat et al. [6] incorporate multi-versioning for read-only access into an online schedule generation to reduce the number of conflicts. Ramadan et al. [60] show that their dependence-aware transactional memory system accepts all conflict-serializable schedules. We do not need to integrate such a scheduling system into Decent STM as transactions register their writes only at commit time. Similar to their approach, we do allow forwarding of tentative versions when reading transactional objects.

Perelman and co-workers [58] prove that no STM algorithm can be space optimal, that is, it cannot ensure that it always maintains the minimum number of object versions. As we have done for Decent STM, they define a garbage collection for versions of shared objects such that versions are kept only when they may be needed by some existing read-only transactions. They also demonstrate that read-only transactions must leave some trace in shared memory, even after they have committed.

12.8. Consensus and commit protocols in STM

Guerraoui et al. [32] compare different one-, two- and three-phase commit protocols in terms of underlying assumptions, given guarantees and message overhead. They also show the close relation to consensus protocols. Further, they propose a decentralized one-round three-phase commit protocol [2]. Contrary to their investigation, we propose for Decent STM a (potentially multi-round) randomized consensus protocol as we want the flexibility to introduce some bias towards certain transactions.

Gray and Lamport [28] propose a non-blocking, fault-tolerant distributed commit protocol, called Paxos Commit algorithm. It employs a fixed number of coordinators that are responsible to achieve an agreement for a decision taken by a dedicated leader, based on majority voting. In Decent STM, the runtimes take a similar role as the coordinators, without requiring a leader and allowing the number of coordinators to vary for each commit request.

Aspnes [5] gives a comprehensive survey of randomized consensus protocols for distributed and shared-memory settings. The commit consensus protocol we defined in Section 9 combines ideas from both branches of research. As in the classical distributed consensus protocols, messages are used to pass information between the processes involved in the protocol. Though we do not consider faulty or even malicious processes, it is possible to integrate standard techniques for failure detection into our commit consensus protocol. As in the shared memory versions, our protocol proceeds in rounds. The introduced randomization does not build upon one shared coin, yet, the probability distribution for the input set is similar adjusted on all nodes as the protocol proceeds.

12.9. Code instrumentation for STM

Deuce STM [45] dynamically instruments Java classes at load time with STM operations. Similar to the JTransactifier, it is uses `@atomic` annotations for methods to determine the scope of transactions. It implements weak atomicity, therefore it does not require a distinction between shared and non-shared objects, and needs to provide each method in a plain and a transactified flavor. Targeting a distributed environment, JTransactifier requires globally used objects to be annotated as they must be serializable. Several optimizations and analyses implemented by Deuce STM, such as transformation closed source libraries, detection of irreversible operations, or flow analysis for detection of transaction-local data, would make the JTransactifier a more powerful tool.

Felber and co-workers [25] present code instrumentation with STM for word-based C/C++ programs using the LLVM compiler framework or assembly code. Their results show that the automatically transformed code is competitive with a manually optimized version.

Chapter 13.

Conclusion

13.1. Summary

Software transactional memory provides a simple mechanism for thread-safe programming. Atomic blocks are a powerful concurrency primitive with strong isolation and consistency guarantees. Data races on shared memory that may arise from executing atomic blocks concurrently are resolved transparently by reverting conflicting operations. However, STM is impaired by severe restrictions, such as the irreversibility of side effects and the restriction to in-memory synchronization.

This thesis featured extensions for STM that circumvent these restrictions, while preserving transactional semantics.

Twilight STM This thesis presented Twilight STM, an STM system that extends the commit phase of a transaction with the execution of twilight code. Twilight code consists of arbitrary user code that runs with special concurrency privileges, such that this code can safely perform irrevocable I/O actions.

The twilight code runs after the transaction's main part, it is initialized by calling **`stm_prepare`** and ends with a call to **`stm_finalize`**. The twilight code may use the Twilight API to introspect and modify the transaction's state to implement sophisticated contention management. To this end, repair operations such as `stm_reread` and `stm_update` enable the inspection and modification of the transaction's read and write set. A reload operation obtains a current version of the memory snapshot obtained by the transaction. Furthermore, the marking of variables in the read and write set with tags offers an intuitive and practical mechanism to identify areas where conflicts arise.

Twilight STM serves as a platform for exploring relaxed transactional semantics. It is the first STM implementation that admits the definition of application-specific conflict schemes without changing the underlying STM system. Thus, Twilight STM is a conservative extension of STM and can be integrated into any STM system with lazy updates.

Decent STM This thesis further presented Decent STM, a fully decentralized STM algorithm. It is based on versioning of shared objects that are employed for creating consistent memory snapshots spanning several nodes. Versions from transactions that are trying to commit concurrently are kept in tentative version

trees. A randomized commit consensus protocol operates on these tentative version trees: It accepts non-conflicting versions by moving them to the committed versions history list. It also discards conflicting versions, thereby causing the respective transactions to roll-back. By adjusting commit probabilities for different kinds of transactions, it is possible to install application-level heuristics for the commit protocol.

The design of Decent STM is targeted for distributed environments. It decouples the process of obtaining shared-memory consistency from the application logic. Replication and migration of data is handled transparently to the programmer. Also, it allows non-blocking reads in the presence of ongoing commits. Thus, Decent STM yields scalability for large-scale distributed systems. The Decent STM algorithm also ensures that there is always a winning transaction in case of competing transactions, and thus guarantees progress of the whole system even in case of rollbacks.

Twilight STM and Decent STM investigate different aspects of STM systems. An interesting open problem is the effect of combining these two STMs. Because of its resemblance with two-phase commit, the Decent STM commit protocol can straightforwardly be extended with twilight zones, thus adding irrevocability to the system. Also, a multi-version scheme for transactional objects can guarantee that transactions always reach their twilight zone based on previous versions, thus allowing more opportunity for sophisticated repair actions that even take multiple versions into account.

13.2. Outlook

To increase the impact of STM on (commercial) software, it is necessary to provide a full and sound integration of STM into a programming language to facilitate the program development. The research results presented in this thesis build the foundation for creating a portable system running concurrent programs on heterogeneous architectures.

Decent STM serves as the consistency layer for a distributed Java virtual machine, called DecentVM. DecentVM has been designed as a platform for STM systems running on heterogenous multi-cores or in a cloud. It creates a single-system illusion by hiding all aspects related to the distributed platform, such as consistency and communication of data, from the programmer. For backwards compatibility with legacy Java applications, it also supports traditional lock-based consistency via monitors. They are emulated with the same VM instructions that implement the STM model: copying objects into the private memory area of a thread, and replacing existing objects with a new version once the commit has succeeded or a memory barrier is passed.

Furthermore, there are plans to incorporate fault-tolerance into the DecentVM system. In the setting of a versioning STM, check-pointing is a straightforward

choice for dealing with failure as memory snapshots are already retained by system. As the tentative version trees indicate, we further envision optimizations such as the execution of transactions on versions still in the commit process. The communication latency can effectively be hidden if application threads are allowed to progress in some kind of sandbox while the transactional memory system is resolving possible memory inconsistencies. This also increases the amount of parallelism and full utilization of the system resources.

DecentVM will further provide algorithms for distributed object instantiation and migration mechanisms to adapt the distributed memory to the changing workloads during the program's life cycle.

The multi-faceted research on STM in the last couple of years has revealed that transactions are an exciting concept that is applicable not only to databases. Though the transactional memory research focuses on high-performance computing and concurrency, the tentative nature of transactions makes them also attractive for other areas of computing, such as monitoring security in applications [42], speculative out-of-order execution [14], or restoring test fixtures [38].

Bibliography

Bibliography

[1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In Phil Wadler, editor, *Proceedings 35th Annual ACM Symposium on Principles of Programming Languages*, pages 63–74, San Francisco, CA, USA, January 2008. ACM Press.

[2] Maha Abdallah, Rachid Guerraoui, and Philippe Pucheral. One-phase commit: Does it make sense? In *Proceedings of the 1998 International Conference on Parallel and Distributed Systems (ICPADS'98)*, page 182, Washington, DC, USA, 1998. IEEE Computer Society.

[3] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* The Pragmatic Programmers, LLC, 2007.

[4] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang.* Prentice Hall, NY, 1993.

[5] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16:165–175, September 2003.

[6] Utku Aydonat and Tarek S. Abdelrahman. Serializability of transactions in software transactional memory. In *TRANSACT '08*, 2008.

[7] M. Ben-Ari. *Principles of concurrent and distributed programming.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[8] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of Data*, pages 1–10, San Jose, California, United States, 1995. ACM.

[9] Annette Bieniusa and Thomas Fuhrmann. Consistency in hindsight: A fully decentralized STM algorithm. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–12. IEEE, 2010.

[10] Annette Bieniusa, Arie Middelkoop, and Peter Thiemann. Brief announcement: Actions in the twilight - concurrent irrevocable transactions and inconsistency repair. In Andréa W. Richa and Rachid Guerraoui, editors, *Proceedings of the 29th ACM SIGPLAN Symposium on Principles of Distributed Computing*, pages 71–72, Zurich, Switzerland, 2010. ACM.

[11] Annette Bieniusa and Peter Thiemann. Proving isolation properties for software transactional memory. In Gilles Barthe, editor, *Proceedings of the 20th European Symposium on Programming*, volume 6602 of *Lecture Notes in Computer Science*, pages 38–56, Saarbrücken, Germany, March 2011. Springer-Verlag.

[12] Colin Blundell, Christopher Lewis, and Milo Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report TR-CIS-06-09, University of Pennsylvania, Philadelphia, PA, USA, May 2006.

[13] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In Siddhartha Chatterjee and Michael L. Scott, editors, *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 247–258, New York, NY, USA, 2008. ACM.

[14] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. Speculative out-of-order event processing with software transaction memory. In *Proceedings of the second international conference on Distributed Event-Based Systems*, DEBS '08, pages 265–275, New York, NY, USA, 2008. ACM.

[15] Eric Bruneton. ASM 3.0, a Java bytecode engineering library. `http://download.forge.objectweb.org/asm/asm-guide.pdf`, 2007.

[16] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In Shail Arora and Gary T. Leavens, editors, *Proceedings of the 25th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 691–707, Reno/Tahoe, Nevada, USA, 2010. ACM Press, New York.

[17] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

[18] Michael J. Cahill, Uwe Röhm, and Alan David Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4), 2009.

[19] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC '08: Proc. IEEE International Symposium on Workload Characterization*, pages 35–46, September 2008.

[20] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006*, LNCS 4167, pages 194–208. Springer, 2006.

[21] Anthony Discolo, Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Satnam Singh. Lock Free Data Structures using STMs in Haskell. In Philip Wadler

and Masami Hagiya, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming FLOPS 2006*, pages 65–80, Fuji Susono, Japan, April 2006. Springer-Verlag.

[22] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. In *Proceedings of the RefineNet Workshop 2009 (REFINE 2009)*. Electronic Notes in Theoretical Computer Science, 2009.

[23] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19:624–633, November 1976.

[24] Alan Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[25] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying Applications using an Open Compiler Framework. In *TRANSACT*, August 2007.

[26] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, April 1985.

[27] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[28] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31:133–160, March 2006.

[29] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI - The Complete Reference: Volume 2, The MPI-2 Extensions*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, 1998.

[30] Rachid Guerraoui, Thomas A. Henzinger, Michal Kapalka, and Vasu Singh. Transactions in the jungle. In Friedhelm Meyer auf der Heide and Cynthia A. Phillips, editors, *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 263–272, Thira, Santorini, Greece, 2010. ACM.

[31] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In Siddhartha Chatterjee and Michael L. Scott, editors, *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184, Salt Lake City, UT, USA, 2008. ACM.

[32] Rachid Guerraoui, Mikel Larrea, and André Schiper. Reducing the cost for non-blocking in atomic commitment. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS'96)*, page 692, Washington, DC, USA, 1996. IEEE Computer Society.

[33] Tim Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325–343, 2005.

[34] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 388–402, Anaheim, CA, USA, 2003. ACM Press, New York.

[35] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, Chicago, IL, USA, June 2005. ACM Press.

[36] Tim Harris and Simon Peyton Jones. Transactional memory with data invariants. In *TRANSACT '06*, June 2006.

[37] Tim Harris and Srđan Stipić. Abstract nested transactions. In *TRANSACT '07*, Portland, OR, USA, August 2007.

[38] Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. DOM transactions for testing JavaScript. In *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques*, TAIC PART'10, pages 211–214, Berlin, Heidelberg, 2010. Springer-Verlag.

[39] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles of Distributed Computing*, pages 92–101, Boston, Massachusetts, 2003. ACM Press, New York, NY, USA.

[40] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[41] Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony Hosking. A transactional object calculus. *Sci. Comput. Program.*, 57(2):164–186, 2005.

[42] Suman Jana, Donald E. Porter, and Vitaly Shmatikov. TxBox: Building Secure, Efficient Sandboxes with System Transactions. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2011.

[43] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[44] Oleg Kiselyov and Chung chieh Shan. Lightweight monadic regions. In Andy Gill, editor, *Haskell 2008*, pages 1–12, Victoria, BC, Canada, September 2008. ACM.

[45] Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive concurrency with java stm. In *Programmability Issues for Multi-Core Computer (MULTIPROG'10)*, 2010.

[46] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris C. Kirkham, and Ian Watson. DiSTM: A software transactional memory framework for clusters. In *Proceedings of the 37th International Conference on Parallel Processing*, pages 51–58. IEEE Computer Society, 2008.

[47] Jim Larus and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

[48] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[49] Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT '09*, 2009.

[50] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[51] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In Josep Torrellas and Siddhartha Chatterjee, editors, *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 198–208, New York, NY, 2006. ACM.

[52] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 53–65, Washington, DC, USA, 2006. IEEE Computer Society.

[53] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Single global lock semantics in a weakly atomic STM. *SIGPLAN Not.*, 43(5):15–26, 2008.

[54] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In Phil Wadler, editor, *Proceedings 35th Annual ACM Symposium on Principles of Programming Languages*, pages 51–62, San Francisco, California, USA, January 2008. ACM.

[55] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 359–370, San Jose, California, USA, 2006. ACM Press, New York, NY, USA.

[56] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

[57] J. Eliot B. Moss. Open nested transactions: Semantics and support. Poster presented at Workshop on Memory Performance Issues (WMPI 2006), Austin, TX, February 2006.

[58] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in STM. In Andréa W. Richa and Rachid Guerraoui, editors, *Proceedings of the 29th ACM SIGPLAN Symposium on Principles of Distributed Computing*, pages 16–25, New York, NY, USA, 2010. ACM.

[59] Cristian Perfumo, Nehir Sönmez, Srdjan Stipic, Osman Unsal, Adrián Cristal, Tim Harris, and Mateo Valero. The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, pages 67–78, Ischia, Italy, 2008. ACM.

[60] Hany E. Ramadan, Indrajit Roy, Maurice Herlihy, and Emmett Witchel. Committing conflicting transactions in an STM. In Daniel Reed and Vivek Sarkar, editors, *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 163–172, Raleigh, NC, USA, 2009. ACM.

[61] David P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1):3–23, 1983.

[62] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *20th International Symposium on Distributed Computing, DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 284–298, Stockholm, Sweden, Sep 2006. Springer.

[63] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *TRANSACT'06*, Jun 2006.

[64] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*, pages 221–228. ACM, 2007.

[65] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In Josep Torrellas and Siddhartha Chatterjee, editors, *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[66] Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, January 2011. `http://hal.archives-ouvertes.fr/inria-00555588/`.

[67] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles of Distributed Computing*, pages 204–213, Ottowa, Ontario, Canada, 1995. ACM Press, New York, NY, USA.

[68] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 191–210, Montreal, QC, CA, 2007. ACM Press, New York.

[69] Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[70] M. F. Spear, M. M. Michael, and M. L. Scott. Inevitability mechanisms for software transactional memory. In *TRANSACT '08*, 2008.

[71] Pierre Sutra and Marc Shapiro. Fault-tolerant partial replication in large-scale database systems. In *Proceedings of the 14th International Euro-Par Conference*, LNCS 5168, pages 404–413. Springer, 2008.

[72] Tim Sweeney. The next mainstream programming language: A game developer's perspective. In Simon Peyton Jones, editor, *Proceedings 33rd Annual ACM Symposium on Principles of Programming Languages*, page 269, Charleston, South Carolina, USA, January 2006. ACM Press.

[73] Gadi Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[74] Haris Volos, Neelam Goyal, and Michael M. Swift. Pathological interaction of locks with transactional memory. In *TRANSACT '08*, Salt Lake City, UT, USA, February 2008.

[75] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M. Swift, and Adam Welc. xCalls: Safe I/O in memory transactions. In *EuroSys '09: Proceedings*

of the fourth ACM European Conference on Computer systems, pages 247–260, Nuremberg, Germany, 2009. ACM.

[76] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[77] Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Transparently reconciling transactions with locking for Java synchronization. In *20th European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 148–173, Nantes, France, July 2006. Springer-Verlag.

[78] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In Friedhelm Meyer auf der Heide and Nir Shavit, editors, *Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 285–296, Munich, Germany, 2008. ACM.

[79] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[80] Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, and Suresh Jagannathan. A uniform transactional execution environment for Java. In Jan Vitek, editor, *22nd European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 129–154, Paphos, Cyprus, 2008. Springer-Verlag.

[81] Ferad Zyulkyarov, Vladimir Gajinov, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic Quake: using transactional memory in an interactive multiplayer game server. In Daniel Reed and Vivek Sarkar, editors, *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–34, Raleigh, NC, USA, 2009. ACM.