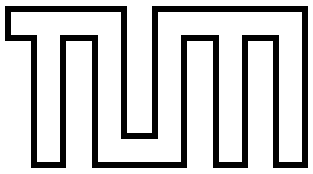


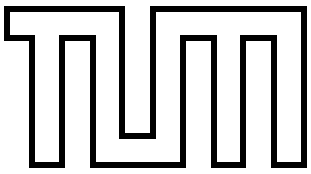
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Epistemic Characterization of Concurrent Computations

Master's Thesis in Informatik

Klaus Freiherr von Gleissenthall



FAKULTÄT FÜR INFORMATIK

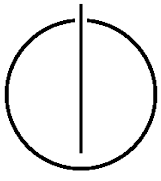
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Epistemic Characterization of Concurrent Computations

Wissensbasierte Charakterisierung nebenläufiger Berechnungen

Author:	Klaus Freiherr von Gleissenthall
Supervisor:	Prof. Dr. Andrey Rybalchenko
Advisor:	Prof. Dr. Andrey Rybalchenko
Submission Date:	14.06.2012



I assure the single handed composition of this master's thesis only supported by declared resources.

München, 14.06.2012 Klaus Freiherr von Gleissenthall

Abstract

Sequential consistency and linearizability are correctness conditions for libraries that are concurrently accessed by several threads. Correctness for such libraries is defined in terms of execution traces, i.e. sequences of events representing calls and returns that threads produce when they access library methods. Traditionally these correctness conditions are seen as a restriction on the order in which the library may process method calls. In this thesis, we approach these correctness conditions from a new perspective: that of the threads calling the library's methods. Threads have a local view on execution traces: they can only see the calls they make and the returns they receive. From their local views, the threads can distinguish some execution traces while others look the same to them. This change of perspective leads us to an interesting discovery: Sequential consistency and linearizability share a common interpretation. A trace is correct if a group of agents can not be sure that the trace does not meet the library's specification. For sequential consistency, this group consists of the threads; for linearizability, it consists of the threads plus another agent that we call the observer. We show that sequential consistency and linearizability can be described by an epistemic logic formula of the form: $\neg D_G \neg \text{correct}$. This means, a trace is correct if it is not distributed knowledge among the agents in group G that the trace does not meet the library's specification. Distributed knowledge is the knowledge of a group of agents that tell each other everything they know. We apply our approach to formalize TSO memory-consistency, showing that TSO is an instance of sequential consistency.

Contents

List of Tables

Chapter 1

Introduction

Concurrent correctness conditions like linearizability [11, 12] and sequential consistency [17] allow us to reason about the correctness of libraries which implement data structures that are accessed by several threads. Correctness for these libraries is specified in terms of execution traces, i.e. sequences of call- and return events that record the threads' invocations of the library's methods. In this thesis, we look at concurrent correctness from the perspective of the threads. That is, we treat the threads as agents with a local view of the computation. Threads can only see the method calls they make and the returns they receive. From this local view, the threads can distinguish some execution traces while others look the same to them. This leads us to a number of interesting insights:

- A trace is sequentially consistent if the threads cannot distinguish it from a correct trace.
- We can describe linearizability by introducing another agent that we call the observer. The observer monitors the order of non-overlapping method calls. An execution trace is linearizable if the threads together with the observer cannot distinguish it from a correct one.
- We can characterize the correctness of a memory trace under the TSO (total store order) weak memory model [26, 24] by the same reasoning. Under TSO, writes are stored in intermediate buffers before they are flushed to memory. To describe TSO, we apply a trick: we define correctness in terms of traces that contain events which mark the positions in which the memory-subsystem decided to flush values from the buffer to memory. These events are invisible to the threads/the programmer, however specifying the correctness of a trace that contains flush events is a lot easier than specifying the correctness of a trace that does not.

> This allows us to characterize TSO traces as follows: a trace is a valid TSO trace if the threads cannot distinguish it from a trace containing a number of unseen flush events that certify its correctness.

Interestingly, our view on concurrent correctness yields a logical characterization: We show that both sequential consistency and TSO can be described by a formula of the form $\neg D_{\text{THREADS}} \neg \text{correct}$. This means for a trace to be sequentially- or TSO-consistent, it must not be distributed knowledge among the threads that the trace is incorrect. Distributed knowledge is the knowledge of a group of agents that tell each other everything they know. For linearizability, we get the formula $\neg D_{\text{THREADS} \cup \{obs\}} \neg \text{correct}$.

Contributions

We make the following contributions:

- We provide logical characterizations of sequential consistency, linearizability, and the TSO memory relaxation, which we prove equivalent to standard definitions.
- We provide a unifying way to formalize concurrent correctness conditions. This is interesting, as they are usually expressed quite heterogeneously.
- We provide an interpretation as to what concurrent correctness conditions actually mean: a trace is correct if a group of agents cannot distinguish it from a trace that meets the library's specification.

Outline

This thesis is structured as follows: in chapter 2, we introduce our notion of views, present our logic and characterize sequential consistency and linearizability. In chapter 3, we extend our logic and characterize the TSO memory relaxation. We sum up our results in chapter 4.

Chapter 2

Epistemic Characterization of Concurrent Correctness

2.1 Introduction

Sequential consistency [17],[10, section 3.4] allows us to reason about the correctness of libraries implementing data-structures that are accessed by multiple threads. Such libraries offer methods that allow their clients to manipulate the data structures they implement (e.g. a library may implement a set, providing methods to add a value, remove a value, or check if the set contains a value). Correctness of such libraries is specified in terms of execution traces. Execution traces are sequences of events representing calls to and returns from the libraries' methods produced by a number of threads accessing the library. As the threads are not forced to synchronize, method calls by different threads may overlap. Sequential consistency allows reasoning about the correctness of execution traces containing overlapping method calls. Instead of having to specify the correctness of each possible interleaving of method calls and returns, the programmer only specifies the correctness of simple, one-after-the-other sequential executions. An execution trace is sequentially consistent if it can be rearranged to yield a sequential execution that meets the library's specification where rearrangement of method calls by the same thread is not allowed.

In this chapter, we look at sequential consistency from a different angle. We treat the threads accessing the library as agents with a local view of the computation, i.e. a thread cannot see the whole execution trace, but only her own method calls and returns. From her local view, a thread can tell the difference between some traces, while others look the same to her. This allows for an interesting new interpretation of sequential consistency: a trace is sequentially consistent if, from the threads' point of view, it is indistinguishable from a correct trace.

Sequential Consistency by Agent Views

We formalize our notion of views by defining projection functions that extract the part of an execution trace a given thread is aware of. That is, each thread only observes the method calls she makes and the returns she receives. Using the projections, we define agent specific indistinguishability relations. An agent's indistinguishability relation links pairs of traces that she cannot tell apart. We proceed to defining joint indistinguishability relations for groups of agents. Joint indistinguishability relations represent a joined forces approach to discriminate between traces – whenever an agent can tell the difference between two traces she tells the others so they can jointly distinguish them.

In this setting we can make our new interpretation of sequential consistency more precise: a trace is sequentially consistent if, from the threads' joint perspective, it is indistinguishable from a trace that meets the library's spec-

ification. That is, if the threads, by their joint indistinguishability relation, cannot spot the difference between the actual trace and a trace that meets the specification, the actual trace is considered to be correct. This provides an interesting intuition about the meaning of sequential consistency: a library is allowed to deviate from its specified behavior as long as the threads do not notice.

A Logical Characterization

Interestingly, given our notion of views, we can easily construct a logical characterization of sequential consistency. Indistinguishability relations yield a standard logic of knowledge: an agent a knows that a formula φ is true ($K_a\varphi$) if φ holds on all traces that she cannot distinguish from the actual trace. Joint indistinguishability among a group of agents G yields *distributed knowledge* ($D_G\varphi$). This is the knowledge of a group that tell each other everything they know. Translated into this logic, we can formulate sequential consistency as follows: a trace is sequentially consistent if it is not distributed knowledge among the threads that the trace is incorrect ($\neg D_{\text{THREADS}}\neg$ correct).

Linearizability

Linearizability [11, 12] extends sequential consistency by assuring that the order between non-overlapping method calls is kept. This makes sure that each method produces its entire visible effect at exactly one point between its invocation and its return. To formalize this constraint, we introduce an agent that we call the observer (obs). The observer's local view of the computation is the order of non-overlapping method calls. We show that a trace is linearizable if it is not distributed knowledge among the threads and the observer that the trace is incorrect ($\neg D_{Threads\cup\{obs\}}\neg$ correct).

Outline

The rest of this chapter follows the structure outlined above. In section 2.2, we introduce preliminary definitions. In section 2.3, we introduce views and indistinguishability relations. We use indistinguishability relations to define the agents' knowledge, and present our logic in section 2.4. In section 2.5, we characterize sequential consistency in our logic. We extend our logic by adding the observer, and define linearizability in section 2.6. In section 2.7, we present our logical characterization of linearizability and prove it equivalent to our definition. Finally, in sections 2.8 and 2.9 we discuss limitations of our characterizations and review related work.

2.2 Preliminaries

In this section we introduce some preliminary definitions. For $i \in \mathbb{N}$, we let $[i] = \{1, \ldots, i\}$. Let $\mathcal{E}$ be a set of events. We denote by $\mathcal{E}^*$ the set of finite-, and by $\mathcal{E}^\omega$ the set of infinite sequences over $\mathcal{E}$. We denote the empty sequence by ϵ. Let $E \in \mathcal{E}^\omega$. Then $E \upharpoonright i$ denotes the finite prefix up to- and including i. Now let $E, E' \in \mathcal{E}^*$. We let $E@i$ be the element of sequence E at position i. We define $len(E)$ to be the length of E, where $len(\epsilon) = 0$. By $E \cdot E'$, we denote the concatenation of E and E'. Sometimes, we omit the concatenation operator and write EE'. For $e \in \mathcal{E}$, we say that $pos(e, E) = j$, if $E@j = e$ and $pos(e, E) = \omega$ otherwise. We write $e \in E$ if $pos(e, E) < \omega$.

We define the set $\mathcal{E}$ of events as $\mathcal{E} := \textsc{Call} \uplus \textsc{Ret}$ with $\textsc{Call} \ni c :=$ (t, call m(v)) representing a thread $t \in \textsc{Threads}$ calling a method $m \in \textsc{Methods}$ with value $v \in \textsc{Values}$, and $\textsc{Ret} \ni r :=$ (t, ret m(v)) representing thread t returning from method m with value v. We make the assumption that each event occurs at most once in each trace.

2.3 Threads' View

In this section, we define the threads' views and their indistinguishability relations. An agent's view of a computation trace is the part of the trace she can observe. We define this part by a projection function that extracts the respective events. We use this projection function to define an indistinguishability relation for each agent. An agent's indistinguishability relation links traces that she cannot distinguish. Later, this indistinguishability relation will serve to define the agent's knowledge.

A thread t can observe events that represent its calling a library method or its returning from one. It can distinguish traces by this information only. Thus, if two traces share the events concerning thread t, t cannot distinguish them.

Definition 1 (Thread Indistinguishability Relation) *For a thread $t \in \textsc{Threads}$ the indistinguishability relation $\sim_t \subseteq (\mathcal{E}^\omega \times \mathbb{N})^2$ is defined such that:*

$$(E, i) \sim_t (E', i') \text{ :iff } (E \upharpoonright i) \downarrow t = (E' \upharpoonright i') \downarrow t$$

where $\downarrow\colon (\mathcal{E}^* \times \textsc{Threads}) \to \mathcal{E}^*$ designates a projection function onto t's local perspective, and we use ":iff" to abbreviate "by definition, if and only if". $E \downarrow t$ is defined such that:

$$\epsilon \downarrow t = \epsilon$$

Table 2.1: Satisfaction Relation for Concurrent Correctness

$(E,i) \vDash \text{correct}$:iff $E \upharpoonright i \in \textsc{Spec}$ where $\textsc{Spec} \subseteq \mathcal{E}^*$ is a specification of the library.
$(E,i) \vDash \neg\varphi$:iff not $(E,i) \vDash \varphi$
$(E,i) \vDash D_G\varphi$:iff for all (E',i'): if $(E,i) \sim_{\cap_G} (E',i')$ then $(E',i') \vDash \varphi$

$$(e \cdot E) \downarrow t = \begin{cases} e \cdot (E \downarrow t), & \text{if } e = (t, \_) \\ E \downarrow t, & \text{otherwise} \end{cases}$$

where "_" represents irrelevant, existential quantification.

Joint Indistinguishability Relations

Joint indistinguishability relations link pairs of traces that a group of agents can distinguish if they share their knowledge. Whenever an agent in the group can tell the difference between two traces, she tells the others, so they can jointly distinguish them.

Let $\mathcal{A} \coloneqq \textsc{Threads}$ be the set of agents and $G \subseteq \mathcal{A}$. We define the joint indistinguishability relation of a group G to be $\sim_{\cap_G} \coloneqq (\bigcap_{a \in G} \sim_a)$. This means whenever an agent can discriminate between the actual trace and another trace, the group can.

2.4 A Logic for Sequential Consistency

In this section, we use the agents' indistinguishability relations to describe their knowledge: having observed a trace, an agent's indistinguishability relation specifies a set of traces that the agent cannot distinguish from the observed trace. If a fact holds on all these traces, the agent knows this fact. Knowledge of an agent $a \in \mathcal{A}$ is represented by the modality $K_a\varphi$. The modal operator D_G represents distributed knowledge – the joint knowledge of a group G. It is defined by the joint indistinguishability relation of G. There is only one atomic proposition: *correct*, which holds on traces that meet the library's specification.

Definition 2 (Syntax) *A formula φ in our logic takes the following form:*

$$\varphi ::= \mathit{correct} \mid \neg\varphi \mid D_G\,\varphi$$

where $G \subseteq \mathcal{A}$. Let Φ denote the set of all formulae in our language.

The logic provides the following constructs:

- The proposition *correct*, signifying correctness with respect to the library's specification.
- The epistemic modality D_G representing *distributed knowledge* among a group of agents G. A fact φ is distributed knowledge among the agents in G if the agents know φ when they share everything they know.

We can macro-define the epistemic modality K_a representing the knowledge of an agent a as $K_a \coloneqq D_{\{a\}}$.

Definition 3 (Semantics) *We define the satisfaction relation $\vDash \subseteq (\mathcal{E}^\omega \times \mathbb{N}) \times \Phi$ in table 2.1.*

Knowledge

The definition for the knowledge modality D_G could be paraphrased as follows: whenever the agents in a group G observe a trace $E \upharpoonright i$ and there is a trace $E' \upharpoonright i'$ that they cannot distinguish from $E \upharpoonright i$, they cannot be sure whether it was $E \upharpoonright i$ or $E' \upharpoonright i'$ they saw. If, however, a formula φ holds for all the traces that they might have possibly observed, they know that φ holds. The dual of knowledge is epistemic possibility, denoted by $\neg D_G \neg \varphi$. If there is at least one trace which the group G cannot distinguish from the actual trace and on which φ holds, group G considers it possible that φ is true.

2.5 Sequential Consistency by Agent Views

Proposition 1 *A trace $E \upharpoonright i$ is sequentially consistent ($seqCons(E,i)$) if and only if there is a trace $E' \upharpoonright i'$ that meets the specification and that the threads cannot jointly distinguish from $E \upharpoonright i$. For all $(E,i) \in (\mathcal{E}^\omega \times \mathbb{N})$:*

> *seqCons (E,i) :iff there is (E',i') s.t.*
> *$(E,i) \sim_{\cap_{\text{THREADS}}} (E',i')$ and $(E',i') \vDash correct$.*

Sequential Consistency as Distributed Knowledge

We can rewrite this characterization using the definition of distributed knowledge to yield a formula for sequential consistency.

Proposition 2 *A trace is sequentially consistent if the threads consider it possible that the trace is correct. For all $(E,i) \in (\mathcal{E}^\omega \times \mathbb{N})$:*

$$\boxed{\textit{seqCons}\ (E,i)\ \textit{iff}\ (E,i) \vDash \neg D_{\textsc{Threads}} \neg \textit{correct}}$$

2.6 Adding Linearizability

Linearizability [11, 12],[10, section 3.5] extends sequential consistency by guaranteeing that each method call takes its effect at exactly one point between its invocation and its return. To characterize linearizability, we introduce another agent that we call the observer. The observer's view of a trace is the order of non-overlapping method calls. We show that a trace is linearizable if the threads together with the observer do not jointly know that the trace is incorrect.

Observer

The observer's view of a trace is the order of non-overlapping method calls. Keeping this order makes sure that each method takes its effect at some point between its invocation and its return. We extract this order with a projection function *obs*, that lists all pairs of events which record a method call returning before a second method call was made.

Definition 4 (Observer Indistinguishability Relation) *The indistinguishability relation of the observer* $\sim_{obs} \subseteq (\mathcal{E}^\omega \times \mathbb{N})^2$ *is given by:*

$$(E,i) \sim_{obs} (E',i') \textit{ :iff } obs(E,i) \subseteq obs(E',i')$$

where obs: $(\mathcal{E}^\omega \times \mathbb{N}) \to \mathcal{P}(\mathcal{E}^2)$ *designates a projection onto the observer's local view, such that:*

$$obs(E,i) = \{(r,c) \in \textsc{Ret} \times \textsc{Call} \mid pos(r,E) < pos(c,E) \leq i\}$$

Consider figure 2.1, showing two example traces, where the black lines represent the interval between a method's invocation and its return. In the left trace, the method calls of threads $t1$ and $t2$ overlap, whereas in the right trace, they are called sequentially, one-after-the-other.

In the right trace, the observer sees that m returned before m' was called, in the left trace, she sees nothing. Having seen a trace $E \restriction i$, the observer cannot distinguish it from any trace $E' \restriction i'$ that respects the order of non-overlapping method calls in $E \restriction i$ and possibly extends it by arranging further overlapping method calls into a sequential order. In our example, having seen the left trace, the observer would consider it possible to have seen the right trace, but not the other way round.

Effectively, the observer monitors linearizability's requirement that method calls must take effect at some point between their invocation and their return. Given this requirement, in the left trace it can either happen that m produces

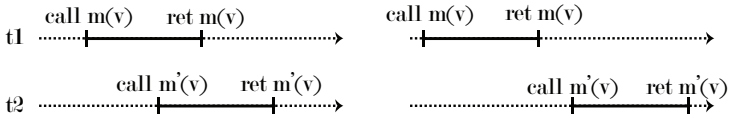


Figure 2.1: Two traces, where threads $t1$ and $t2$ call methods m and m'.

its effect before m', or vice versa. In the right trace, this is not possible. m takes effect before m', even if m takes effect at its latest possible point and m' at its earliest. In other words: the order of sequential method calls is fixed; the order of overlapping calls is not.

This is reflected by the observer's indistinguishability relation. In the left trace, the observer does not spot any ordering constraints, so she can neither distinguish the trace from a trace where m returns before m' is called, nor from one where m' returns before m is called. In the right trace, the order between the two calls is fixed. The observer can distinguish this trace from the left one.

Linearizability

To state our definition of linearizability, we need to define a "real-time" precedence relation. Its function corresponds to that of the observer.

Definition 5 ("Real-Time" Precedence Order) *We define* $\preceq_{real} \subseteq (\mathcal{E}^\omega \times \mathbb{N})^2$*:* $(E,i) \preceq_{real} (E',i')$ *:iff there is a bijection* $\pi : \{1,\dots,i\} \to \{1,\dots,i'\}$ *s.t for all* $j \in \mathbb{N}$ *such that* $j \leq i : E@j = E'@\pi(j)$ *and for all* $j,k \in \mathbb{N}$ *such that* $j < k \leq i$*: if* $E@j \in$ Ret *and* $E@k \in$ Call *then* $\pi(j) < \pi(k)$*.*

Definition 6 (Linearizability Relation) *We define the linearizability relation* $\preceq_{lin} \subseteq (\mathcal{E}^\omega \times \mathbb{N})^2$ *:* $\preceq_{lin} := \sim_{\cap_{\text{Threads}}} \cap \preceq_{real}$*.*

Definition 7 (Linearizability) *For all* $(E,i) \in \mathcal{E}^\omega \times \mathbb{N}$*: lin (E,i) :iff there is* (E',i') *such that* $(E,i) \preceq_{lin} (E',i')$ *and* $(E',i') \vDash$ *correct.*

We follow recent literature on linearizability [9, 25] and allow specifications and thread-local traces to be non-sequential. In the original definition of linearizability, pending method calls, i.e. method calls for which no matching[1] return event exists, can either be discarded or completed with an arbitrary response event [12, p.469]. We do not allow for this. As pointed out in [9,

[1]For an event (t, call m(_)), (t, ret m(_)) is a matching return.

p.459], completing or discarding pending method calls amounts to allowing methods to diverge before or after having produced their effect. This is just a special choice of liveness requirement that is by no means particular to linearizability. That is, we could reproduce the same treatment of pending calls for sequential consistency by choosing an appropriate specification. Thus, in an attempt to unify both conditions, we leave the treatment of pending calls to the specification.

To state our results, we formalize the assumption that each event only occurs once in each trace.

unique :=	for all E $\in \mathcal{E}^\omega$ and all $j, k \in \mathbb{N}$:
	if $E@j = E@k$ then $j = k$.

2.7 Linearizability by Agent Views

Theorem 1 *A trace $E \downharpoonright i$ is linearizable if and only if there is a trace $E' \downharpoonright i'$ that meets the specification and that the threads, together with the observer, cannot jointly distinguish from $E \downharpoonright i$. For all $(E, i) \in (\mathcal{E}^\omega \times \mathbb{N})$:*

if unique then
lin (E,i) iff there is (E', i') :
$(E, i) \sim_{\cap_{\mathcal{A}}} (E', i')$ *and* $(E', i') \vDash correct$.

where we let $\mathcal{A} :=$ Threads $\uplus \{obs\}$.

Proof 1 *We provide the proof in the Appendix A. Table A.1 shows the left-to-right-, table A.2 the right-to-left direction. The left-to-right direction mainly consists of unfolding definitions. For the right-to-left direction, we make use of lemma 3. It establishes that, given our uniqueness assumption, whenever two traces share their thread-local traces, they contain the same events, and thus a bijection between their indices can be found. Having established the existence of this bijection, we need to show that it preserves the order of sequential method calls. This is guaranteed by the observer's indistinguishability relation.*

Linearizability As Distributed Knowledge

By applying the definition of distributed knowledge, we can state our logical characterization:

if unique then
lin (E, i) iff $(E, i) \vDash \neg D_{\mathcal{A}} \neg correct$

2.8 Discussion and Limitations

In our framework we allow for only one library to be accessed. This can however easily be dealt with by annotating events with a library id. This extension does not affect our characterization of the correctness conditions. We could equally extend our framework to deal with atomic transactions as in *serializabilty* [20], by introducing events of the form atomic$(e_1 e_2 \dots e_n)$ with $e_1, e_2, \dots, e_n \in \mathcal{E}$, and adjusting projections and correctness accordingly. To characterize *quiescent consistency* [23], [10, pp.49-51], we would have to weaken the threads' view of the computation, i.e. threads can see their calls and returns but are oblivious about their order. Together with an observer that remembers the order of events which are separated by a period of quiescence, we could characterize quiescent consistency by $\neg D_{\mathcal{A}} \neg$ correct.

Our notion of knowledge corresponds to a standard epistemic logic framework with corresponding axioms for knowledge. We summarize them in Appendix B. Interestingly, introducing the observer comes at the cost of a weaker notion of knowledge, i.e. *S4* instead of *S5*. More precisely, the observer lacks the ability of negative introspection, i.e. $\neg K_{obs}\varphi \rightarrow K_{obs}\neg K_{obs}\varphi$ does not hold in general. This has the consequence that for linearizability, the agents do not generally know whether a trace is linearizable or not, as we show in Appendix C. Because the observer lacks the ability of negative introspection, it can happen that a trace is linearizable but the agents consider it possible that it is not. Sadly, this is of little practical relevance as the observers view is not realistic in the sense that it is beyond the perspective of the threads, i.e. the programmer accessing the library.

2.9 Related Work

Our epistemic framework closely follows [16] and [14]. Our logic is an instance of an *interpreted system*, as defined in [7, section 4.2], our agents have *perfect recall* and perform actions *asynchronously* [7, section 4.4] as they share no global clock. The only application of epistemic logic to concurrent correctness that we are aware of, is an axiom type for sequential memory consistency presented in Hirai's characterization of wait-free computations in an intuitionistic epistemic logic [13]. Our definition of linearizability closely follows [9].

Chapter 3

Epistemic Characterization of TSO

3.1 Introduction

Most programmers assume that memory behaves sequentially consistent, i.e. that the order of method calls issued by the same thread is kept even if multiple threads access the memory. On modern multi-processor architectures this is however no longer true. Performance optimizations, such as intermediate caches and buffers, weaken the guarantees memories provide on the order in which stores and loads are effected.

TSO

In this chapter, we extend our logic to describe TSO [26, 24] one of the most common memory relaxations [1] made among others by x86 processors [22]. Under TSO, values are not directly written to memory but stored in intermediate buffers. There is one such store buffer per (hardware-)thread. Whenever a thread t stores a value v on address a, this store is not directly processed, but saved in t's buffer. If t's next operation is a read on address a, instead of returning the value stored in memory, this read will yield v, the latest value for a in t's store buffer. Other threads, however, have no access to t's buffer. Thus, if another thread performs a read, this read will return the last value written to memory, or the latest value in the thread's own store buffer, if its buffer contains a value for address a. Eventually, however invisibly to the threads, the memory-system flushes v from t's store buffer and transmits it to memory, thus making it commonly available. This means, from the other threads point of view, the store of v to a does not take place until it is flushed from t's store buffer. This may cause store operations to be seemingly delayed past subsequent read operations of the same thread – a behavior often unexpected by the programmer [5].

TSO by Agent Views

Being a valid TSO memory trace essentially means being sequentially consistent with respect to a memory specification that allows store buffers. This means, following the last chapter, our logical characterization of TSO memory traces is of the form $\neg D_{\text{THREADS}} \neg \text{TSOCorrect}$, with an adequate specification TSOCorrect, however, there is a twist. Determining the correctness of a trace that does not contain flush events is highly non-straightforward; determining the correctness of a trace containing flush events is easy. We exploit this fact by defining TSOCorrect in terms of the simpler traces containing flush events. A trace without flush events is a valid TSO trace if we can insert suitable flush events such that the resulting trace is correct with respect to our specification.

This search for suitable flush events comes naturally in our knowledge framework: as flush events are invisible to the threads, the threads do not know when or whether flush events occur. For the threads to reject a trace, the threads must jointly know that the trace is incorrect ($D_{\text{THREADS}}\neg$TSOCorrect). But this means that there is no way to insert flush events into the trace such that the resulting trace meets our specification. If however there is a number of flushes that certify the correctness of the trace, the threads consider it possible that the trace is valid under TSO ($\neg D_{\text{THREADS}}\neg$TSOCorrect) as these flush events might just have taken place without the threads seeing them. In fact, this extends the reasoning presented in the last chapter – if a trace could possibly be correct, we accept it.

Correctness of Traces Containing Flush Events

We describe the correctness of a trace which contains flush events by the following formula:

$$\text{TSOCorrect} := \forall t \forall a \forall v(\boxminus(\text{load(t,a,v)} \rightarrow (\text{ldBuff}(t,a,v) \ \vee \text{ldMem}(t,a,v)))) \wedge \text{FlushOrder}$$

This means, a trace is correct if whenever a thread t loads a value v from address a:

- It loads v from its buffer, i.e. v is the latest value for a in t's store buffer (ldBuff) or
- It loads v from memory, in which case t's store buffer must not contain values for a, and v must be the latest value flushed to memory (ldMem).

Additionally, we get the constraint that for each thread, values written to the store buffer must be flushed to memory in order (FlushOrder).

With this correctness condition and our epistemic framework, our description of TSO traces follows naturally: we change the threads' projection function such that flush events are invisible to the threads. This simple change leads us to the following description: a trace is valid under TSO if the threads do not know that the trace is incorrect with respect to our specification ($\neg D_{\text{THREADS}}\neg$ TSOCorrect).

Outline

The rest of this chapter is structured as follows. In section 3.2, we extend our logic with temporal modalities and quantification. In section 3.3, we characterize the correctness of a trace containing flush events. We adapt the threads' views in section 3.4, and present our logical characterization of TSO in section

3.5. To justify our results, we give a standard operational characterization of TSO memory traces (see e.g. [4] or [6], or [2]) and prove our logical and operational characterizations equivalent, in section 3.6. Finally in sections 3.8 and 3.9, we discuss limitations of our characterization and review related work.

3.2 A Logic for TSO

In this section we present our logic. We extend the logic presented in the last chapter by temporal modalities and first order quantification.

Events

We record memory operations in events from the set $\mathcal{E} \ni e := st(t,a,v) \mid ld(t,a,v) \mid fl(t,a,v)$ representing $t \in$ Threads storing $v \in$ Values to $a \in$ Addresses, t loading v from a, or t flushing v to a, respectively. We no longer need to record call and return events separately, as we assume that method calls always terminate.

Definition 8 (Syntax) *A formula φ in our logic takes the form:*

$$\varphi ::= store(t,a,v) \mid load(t,a,v) \mid flush(t,a,v) \mid \varphi \wedge \varphi \mid \neg\varphi \mid \ominus\varphi \mid \varphi S \varphi \mid D_G\varphi \mid \forall x(\varphi)$$

with $G \subseteq$ Agents.

The logic provides the following additional constructs:

- Predicates $store(t,a,v)$, $load(t,a,v)$, $flush(t,a,v)$ representing t storing, loading or flushing v on a.
- The temporal modality $\ominus$, where $\ominus\varphi$ means: φ holds in the last state.
- The temporal modality S, where $\varphi S \psi$ means: since ψ occurred, φ holds.
- First order quantification.

Let Φ denote the set of all formulae in our logic.

Definition 9 (Semantics) *We define the satisfaction relation $\vDash \subseteq (\mathcal{E}^\omega \times \mathbb{N}) \times \Phi$ in table 3.1. We define the domain* $\mathrm{D} :=$ Threads $\uplus$ Values $\uplus$ Address. *As before, we let $\sim_{\cap_G} := (\bigcap_{a \in G} \sim_a)$. By $\varphi[d/x]$, we denote the term φ with all occurrences of x replaced by d.*

We macro-define the following standard operators: $\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi)$, $\varphi \rightarrow \psi := \neg\varphi \vee \psi$, $\top := \forall t \forall a \forall v(\mathrm{store}(t,a,v) \rightarrow \mathrm{store}(t,a,v))$, $\diamondminus\varphi := \top S \varphi$ ("once φ"), $\boxminus\varphi := \neg \diamondminus \neg\varphi$ ("so far φ"), and $\exists x(\varphi) := \neg\forall x(\neg\varphi)$.

Table 3.1: Satisfaction Relation for TSO

$(E,i) \vDash \text{store}(t,a,v)$:iff $E@i = st(t,a,v)$
$(E,i) \vDash \text{load}(t,a,v)$:iff $E@i = ld(t,a,v)$
$(E,i) \vDash \text{flush}(t,a,v)$:iff $E@i = fl(t,a,v)$
$(E,i) \vDash \varphi \wedge \psi$:iff $(E,i) \vDash \varphi$ and $(E,i) \vDash \psi$
$(E,i) \vDash \neg\varphi$:iff not $(E,i) \vDash \varphi$
$(E,i) \vDash \ominus\varphi$:iff $i > 0$ and $(E, i-1) \vDash \varphi$
$(E,i) \vDash \varphi S \psi$:iff there is $j \leq i$ s.t. $(E,j) \vDash \psi$ and for all k s.t. $j < k \leq i : (E,k) \vDash \varphi$
$(E,i) \vDash D_G\varphi$:iff for all (E',i'): if $(E,i) \sim_{\cap_G} (E',i')$ then $(E',i') \vDash \varphi$
$(E,i) \vDash \forall x(\varphi)$:iff for all $d \in \mathrm{D} : (E,i) \vDash \varphi[d/x]$

Table 3.2: Macro-definitions

$\text{stored}(t,a,v) :=$	$\diamondminus(\text{store}(t,a,v))$	"previously, t stored v to a"
$\text{flushed}(t,a,v) :=$	$\diamondminus(\text{flush}(t,a,v))$	"previously, t flushed v to a"
$\text{buffered}(t,a,v) :=$	$\text{stored}(t,a,v) \wedge \neg\text{flushed}(t,a,v)$	"t buffered v for a"

Table 3.2 macro-defines predicates *stored* and *flushed* that represent the existence of store and flush actions in the past. The predicate *buffered* reflects the fact that to be buffered, a value must have been stored and must not have been flushed yet.

3.3 Correctness of Traces Containing Flush Events

We now formalize the correctness of a memory trace containing flush events (TSOCorrect) in our logic. Each time a thread loads a value v from address a, one of the following two conditions must be true:

- It loads v from from the local store buffer and v is the latest value for a in the buffer (ldBuff).

- It loads v from memory, v is the latest value flushed to a, and the store buffer contains no value for a (ldMem).

We characterize these conditions in our logic using temporal modalities only. We then characterize the requirement that stores from the same thread must be flushed in order (FlushOrder), and combine the parts to yield the correctness condition TSOCorrect.

Load from the Store Buffer

If a thread t loads a value v for address a from its store buffer, v must be the latest value for a in t's buffer. We say that v locally is the latest value for t on a, if t stored v on a and t stored no v' on a later.

$$\boxed{\text{locallyLatest}(t, a, v) := \forall v'(\neg\text{store}(t, a, v')\ S\ \text{store}(t, a, v))}$$

If t loads v for a from its store buffer, v must be in the store buffer, i.e. t must not have flushed v yet, and v must be the locally latest value for t on a.

$$\boxed{\text{ldBuff}(t, a, v) := \neg\text{flushed}(t, a, v) \wedge \text{locallyLatest}(t, a, v)}$$

Load from Memory

If t loads v for a from memory, there must be no entry for a in t's store buffer, and v must be the latest value flushed to a. We say that v globally is the latest value on a if some thread t flushed v to a, and no thread t' flushed a value v' to a later:

$$\boxed{\text{globallyLatest}(a, v) := \forall v'(\neg(\exists t'\ \text{flush}(t', a, v'))\ \ S\ \exists t\ \text{flush}(t, a, v))}$$

If thread t loads v for a from memory, there must be no values for a in t's store buffer, and v must globally be the latest value on a:

$$\boxed{\text{ldMem}(t, a, v) := \forall v'(\neg\text{buffered}(t, a, v')) \wedge \text{globallyLatest}(a, v)}$$

Flush in Order

For each thread t, values are flushed in the order in which they were stored:

$$\boxed{\begin{aligned}\text{FlushOrder} := \forall t \forall a \forall v \forall a' \forall v' (\quad & \neg\ \text{store}(t, a, v)\ S\ \text{store}(t, a', v') \rightarrow \\ & \neg\ \text{flush}(t, a, v)\ S\ \text{flush}(t, a', v'))\end{aligned}}$$

TSOCorrect

We can now specify the correctness of a memory allowing store buffers:

$$\text{TSOCorrect} := \forall t \forall a \forall v (\boxminus(\text{load(t,a,v)} \rightarrow (\text{ldBuff}(t,a,v) \ \vee \text{ldMem}(t,a,v)))) \wedge \text{FlushOrder}$$

That is, a trace is correct if each value has either been read from the store buffer or the memory, and values stored by the same thread have been flushed in order.

3.4 Threads' View

We now adjust the threads' view. As flush operations are not visible to the threads, we need to adjust the threads' indistinguishability relations. We define:

$$(E,i) \sim_t (E',i') \text{ :iff } \mathit{visible}(E \restriction i) \downarrow t = \mathit{visible}(E' \restriction i') \downarrow t$$

where $\mathit{visible} : \mathcal{E}^* \rightarrow \mathcal{E}^*$ is defined such that: $\mathit{visible}(\epsilon) = \epsilon$, and

$$\mathit{visible}(e \cdot E) = \begin{cases} \mathit{visible}(E), & \mathit{if}\, e = \mathit{fl}(\_) \\ e \cdot \mathit{visible}(E), & \mathit{otherwise}. \end{cases}$$

We adjust $\downarrow t$ to match our new event structure:

$$\epsilon \downarrow t = \epsilon$$

$$(e \cdot E) \downarrow t = \begin{cases} e \cdot (E \downarrow t), & \text{if } e = \_(t,\_) \\ E \downarrow t, & \mathit{otherwise}. \end{cases}$$

3.5 TSO by Agent Views

To state our result, we need to make further assumptions about the traces a memory may produce. For a trace to be *balanced* (i.e. $(E,i) \vDash$ balanced), we require three conditions:

- Each value that is flushed has previously been stored by the same thread.
- Each value is stored only once.
- Each value is flushed only once.

We formalize these conditions in Appendix D.

Theorem 2 *A memory under TSO may produce all traces $E \restriction i$ such that:*

$$\boxed{(E, i) \vDash \neg D_{\text{THREADS}} \neg (\mathit{balanced} \wedge \mathit{TSOCorrect})}$$

To justify our result, in the next section, we define an operational semantics of a memory allowing store buffers and prove our logical characterization equivalent.

3.6 An Operational Characterization of TSO

In this section, we define an operational semantics for memories allowing store buffers.

Configurations

The set of memory configurations is defined in Table 3.3. A memory configuration $\sigma = (m, b, S) \in \text{CONFIG}$ consists of the memory state m, a function b mapping each thread t to a store buffer α, and a set S of already stored values. We represent the memory state as a finite partial map from memory addresses to values, and buffers as finite sequences of address-value pairs. We keep a set S of values that have already been stored.We make the assumption that each value is stored only once, and use S to enforce this constraint.

Transition Relation

The memory can execute actions from the set $\text{ACT} ::= st(t, a, v) \mid ld(t, a, v) \mid fl(t, a, v) \ni \gamma$, representing $t \in \text{THREADS}$ storing, loading, or flushing $v \in \text{VALUES}$ on $a \in \text{ADDRESSES}$, respectively. We define an operational semantics of TSO by a transition relation $\rightarrow_{TSO} : \text{CONFIG} \times \text{ACT} \times \text{CONFIG}$ on memory configurations. Transitions are labeled by actions from ACT. We state the inference rules defining $\rightarrow_{TSO}$ in Table 3.4, where $g[x := y]$ denotes the function that has the same value as the function g everywhere, except for x, where it has value y.

[ST] specifies the store operation. A thread t can store v to a, if v has not been stored before. Value v is not directly written to memory, but the address-value pair (a, v) is added to the front of t's store buffer. Value v is added to the set of already stored values.

Thread t can flush v from its store buffer α to address a if (a, v) is the last entry in α. The flush operation updates the memory state m, so that a contains v, and removes (a, v) from the store buffer.

[LD-BUFF] and [LD-MEM] specify the load operation. Thread t can load v for a from its store buffer α if the pair (a, v) is in α and α contains no newer entries for a. If the store buffer contains no value for a but a is in the domain of m, i.e. a value for a has been stored by some thread, then t loads the latest value flushed to a.

Table 3.3: TSO - configurations

MEM :=	ADDRESSES $\rightharpoonup_{fin}$ VALUES $\ni m$
BUFF :=	(ADDRESSES $\times$ VALUES)* $\ni \alpha$
STORED :=	$\mathcal{P}$(VALUES) $\ni S$
CONFIG :=	MEM $\times$ (THREADS $\to$ BUFF) $\times$ STORED $\ni \sigma, \sigma'$

Table 3.4: Inference rules for $\to_{TSO}$

$$\frac{v \notin S}{m, b\,[t := \alpha], S \xrightarrow{st(t,a,v)}_{TSO} m, b\,[t := (a,v)\alpha], S \uplus \{v\}}\ [\text{ST}]$$

$$\frac{}{m, b\,[t := \alpha(a,v)], S \xrightarrow{flush(t,a,v)}_{TSO} m\,[a := v], b\,[t := \alpha], S}\ [\text{FLUSH}]$$

$$\frac{\alpha = \alpha_1(a,v)\alpha_2 \text{ with } (a,\_) \notin \alpha_1}{m, b\,[t := \alpha], S \xrightarrow{ld(t,a,v)}_{TSO} m, b\,[t := \alpha], S}\ [\text{LD-BUFF}]$$

$$\frac{(a,\_) \notin \alpha \qquad a \in dom(m) \qquad m(a) = v}{m, b\,[t := \alpha], S \xrightarrow{ld(t,a,v)}_{TSO} m, b\,[t := \alpha], S}\ [\text{LD-MEM}]$$

Traces

A *finite run* ρ from σ to σ' is a sequence $\sigma_0\gamma_1\sigma_1\gamma_2\ldots\gamma_n\sigma_n$, where (1) $\sigma_0 = \sigma$, and $\sigma_n = \sigma'$ and (2) $\sigma_{i-1} \xrightarrow{\gamma_i}_{TSO} \sigma_i$ for $i \in [n]$. The *trace* of a finite run ρ is the projection of ρ to elements in ACT. We write $\sigma \xrightarrow{\tau}{}^*_{TSO} \sigma'$ if there is a finite run from σ to σ' with trace τ. Let $m_e : \varnothing \to$ VALUES and $b_e := \lambda t \in$ THREADS.ϵ, i.e. the function that assigns ϵ to all $t \in$ THREADS. The set of TSO-traces is defined as $\mathcal{T}_{TSO} = \{\tau \mid (m_e, b_e, \varnothing) \xrightarrow{\tau}{}^*_{TSO} (\_, b_e, \_)\}$.

3.7 Equivalence Operational and Logical Characterization

Equivalence

Theorem 3 *The operational and the logical description of TSO are equivalent.*

$$\boxed{E \downarrow i \in \mathcal{T}_{TSO} \text{ iff } (E, i) \vDash \mathit{TSOCorrect} \wedge \mathit{balanced}}$$

Proof 2 *We provide the proof in Appendix D. Essentially, for the left-to-right direction, we need to show that a trace produced by our operational semantics satisfies all the conditions our logical characterization imposes, i.e. it produces balanced traces, values are either read from buffer or from memory and values are flushed in the order in which they were stored. For the right-to-left direction, we show that whenever we apply an inference rule, its preconditions are satisfied.*

3.8 Discussion and Limitations

In its present form, our characterization does not include memory fences [22], i.e. operations that allow the user to flush the store buffer, and atomic read-write operations [2, 5]. This could easily be dealt with by extending our logic with atomic read-write operations $arw(t, a, v, v')$, signifying that t in one atomic step read v and wrote v' on a and adding the requirement that whenever we encounter $arw(t, a, v, v')$, t's store buffer must be empty. As pointed out in [5] and [2], a fence operation is equivalent to an atomic read-write to an irrelevant location.

We can change our characterization to describe the PSO-relaxation [26, 1, 24], by changing FlushOrder to:

$$\begin{aligned}\text{PSOFlushOrder} := \forall t \forall a \forall v \forall v' (\quad & \neg\, \text{store}(t, a, v)\ S\ \text{store}(t, a, v') \rightarrow \\ & \neg\, \text{flush}(t, a, v)\ S\ \text{flush}(t, a, v'))\end{aligned}$$

This means, only the order of stores by the same thread to the same address is preserved. This corresponds to an operational model, where each thread has a separate store buffer for each address [2, section 3.3]. Under RMO [26, 1], threads are allowed to buffer reads which amounts to guessing values the memory will contain in the future [2, section 8.1]. To allow this, we would have to introduce invisible validation events $val(t, a, v)$, representing thread t validating its guess that address a will contain value v.

3.9 Related Work

As in the last chapter, our logic follows [16] and [14], notably in the non-standard treatment of first order quantification, i.e. substituting domain elements instead of defining an assignment function. This means, except for our treatment of quantification, our logic is standard [7, chapter 4], [18].

We can now discuss the relation to Hirai's logic for wait-free communication [13] in more detail. Contrarily to our logic, Hirai's logic is non-classical. Once

an agent knows a fact φ, she knows it from then on. Thus the agents can be seen as passing proofs that, once known, assert the future knowledge of the proven fact. Consequently, their knowledge modality also differs in its axiomatic description. Most notably the axiom $\vee K$ states that, whenever a disjunction is known, one of the disjuncts has to be known. This is not generally true in our framework. Hirai introduces an axiom type describing sequential consistency. It essentially assures that memory states are totally ordered.

Another related logic is Ramanujam's locally linear time temporal logic [21]. In locally linear time temporal logic agents have local views but synchronize by executing common operations. This allows them to make assertions about other agents past future, or present state. By construction, all assertions are independent of the interleaving of agent actions. It would be interesting to see whether our results are reproducible in locally linear time temporal logic.

TSO is traditionally specified either axiomatically [26, 24, 5, 19] or operationally [22, 6]. As far as we are aware, apart from Hirai's axiom type for sequential memory consistency, our work constitutes the first logical description of memory consistency models.

Chapter 4

Conclusions and Future Work

In this thesis we present a look at concurrent correctness from the perspective of the threads accessing the library. By formalizing this perspective, we arrive at a logical description of sequential consistency and linearizability, which we extend to describe the TSO weak memory model.

Our logical description unifies the description of correctness conditions which are usually represented quite heterogeneously. Sequential consistency is usually stated in natural language [17, 10], linearizability in terms of ordering relations [12, 10, 11] or permutations [9], and TSO either axiomatically [26, 24, 5, 19] or operationally [22, 6].

Our characterization also provides an interesting interpretation of library correctness in a concurrent setting: an execution trace is considered correct if the agents cannot distinguish the trace from a trace that meets the specification. That means a trace is considered correct, if the agents cannot be sure that it is incorrect.

This reasoning extends readily to the more complicated case of TSO. We specify the correctness of TSO memory traces in terms of simpler traces, containing flush events. These flush events are invisible to the threads. If there exists a trace containing a number of flush events that certify its correctness, and the threads cannot distinguish the trace containing the flush events from the actual trace, the actual trace is a valid TSO trace.

We think that our logical framework, together with the idea that agents may reason about invisible operations, as in our description of TSO, will readily allow to describe other correctness conditions, such as the ones we hinted at in our discussions.

There are some interesting directions for future research. The traces augmented with flush operations in our description of TSO serve as a kind of certificate, that can easily be checked. This reminds of the certificates used to characterize the complexity class NP. Actually the problem of checking if a trace is sequentially consistent is known to be NP-complete [8]. It might thus be interesting to investigate the complexity of correctness conditions that can be described by our framework.

The threads' not knowing that a trace is incorrect could also be understood as the threads' not being able to prove the library's faultiness to one-another. It would thus be interesting to formalize concurrent correctness in a logic of proofs [15].

Acknowledgements

I would like to thank Prof. Dr. Andrey Rybalchenko for providing me with such an interesting and motivating subject. Thanks for being such a great advisor! Thanks to Anja, Basti, Jan and Katrin for proof-reading and the time spent in the library. Thanks to my parents for their support.

Appendices

Appendix A

Equivalence Proof for Linearizability

Definition 10 (Eventset) *Let us denote by* $[\![\cdot]\!] : \mathcal{E}^\omega \to \mathcal{P}(\mathcal{E})$ *a function that transforms a trace into the set of events it contains, that is:*
For all $E \in \mathcal{E}^\omega : [\![E]\!] = \{e \mid pos(e,E) \leq len(E)\}$.

Proposition 3 (Union of Thread-Eventsets) *For all* $E \in \mathcal{E}^*$: $[\![E]\!] = \uplus_t [\![E \downarrow t]\!]$.

Proof 3 *By induction on len(E).*
For len(E)=0, we have $E = \epsilon$ *and* $\varnothing = \varnothing$.
For len(E) = n+1, we have $E = e \cdot E'$ *for some* $e \in \mathcal{E}$, $E' \in \mathcal{E}^*$.
We get $[\![E']\!] \uplus \{e\} = \uplus_{t' \neq t} [\![E' \downarrow t']\!] \uplus [\![E' \downarrow t]\!] \uplus \{e\}$, *for some t, and by the induction hypothesis:* $[\![E']\!] \uplus \{e\} = [\![E']\!] \uplus \{e\}$.

Lemma 1 *For all* $(E,i),(E',i') \in \mathcal{E}^\omega \times \mathbb{N}$: *if unique and* $(E,i) \sim_{\cap_{\text{THREADS}}} (E',i')$ *then* $i = i'$.

Proof 4 *For a proof by contradiction we assume* $i > i'$ *without loss of generality. Then, by unique, there is an* $e \in \mathcal{E}$ *such that* $e \in [\![E \downharpoonright i]\!]$ *and* $e \notin [\![E' \downharpoonright i']\!]$. *By proposition 3:* $e \in [\![(E \downharpoonright i) \downarrow t]\!]$ *for some* $t \in$ THREADS *but* $e \notin [\![(E' \downharpoonright i') \downarrow t]\!]$. *But by* $(E,i) \sim_{\cap_{\text{THREADS}}} (E',i')$, *we have* $(E \downharpoonright i) \downarrow t = (E' \downharpoonright i') \downarrow t$ *and thus* $[\![(E \downharpoonright i) \downarrow t]\!] = [\![(E' \downharpoonright i') \downarrow t]\!]$, *from which we get the contradiction.*

Lemma 2 *For all* $(E,i),(E',i') \in \mathcal{E}^{\omega}\times\mathbb{N}$*: if* $(E,i) \sim_{\cap_{\text{THREADS}}} (E',i')$ *and* $E@j = e$ *for some* $j \in \mathbb{N}$ *such that* $j \leq i$ *then there is* $j' \in \mathbb{N}$ *such that* $1 \leq j' \leq i'$ *and* $E'@j' = e$*.*

Proof 5 *Suppose* $j \leq i$*,* $E@j = e$ *and* $(E,i) \sim_{\cap_{\text{THREADS}}} (E',i')$*. By proposition 3,* $e \in [\![(E \downharpoonright i) \downarrow t]\!]$ *for some* $t \in$ THREADS*. Then because* $(E \downharpoonright i) \downarrow t = (E' \downharpoonright i') \downarrow t$ *we have* $[\![(E \downharpoonright i) \downarrow t]\!] = [\![(E' \downharpoonright i') \downarrow t]\!]$ *and thus by proposition 3:* $e \in [\![(E' \downharpoonright i')]\!]$ *and thus by definition 10,* $E'@j' = e$ *for some j' with* $j' \leq i'$*.*

Lemma 3 (Existence of a Bijection) *For all* $(E,i),(E',i') \in \mathcal{E}^{\omega} \times \mathbb{N}$*: if unique and* $(E,i) \sim_{\cap_{\text{THREADS}}} (E',i')$ *then there exists a bijective function* $\pi : \{1,\ldots,i\} \to \{1,\ldots,i'\}$ *and for all* $j \in \mathbb{N}$ *such that* $j \leq i : E@j = E'@\pi(j)$*.*

Proof 6 *Let* $j \in \mathbb{N}$ *such that* $j \leq i$ *and* $E@j = e$*. By lemma 2 we know that* $E'@j' = e$ *for some* $j' \in \mathbb{N}$ *with* $j' \leq i'$*. We will now show that the mapping from j to j' is a function. Suppose there was* $k' \neq j'$ *with* $k' \in \mathbb{N}$ *such that* $E'@k' = e$ *and* $k' \leq i'$*. This cannot be, since by unique each event occurs at most once in each trace. Let us denote that mapping by* π*. We now need to show that* π *is a bijection. By lemma 1,* $i = i'$ *and we have* $\pi : \{1 \ldots i\} \to \{1 \ldots i\}$*. This means it suffices to show that* π *is injective. Now for a contradiction suppose that for* $j,k \in \mathbb{N}$ *with* $j,k \leq i$*:* $E@j = e$ *and* $E@k = e'$ *for some* $e,e' \in \mathcal{E}$ *with* $j \neq k$ *and thus by unique* $e \neq e'$*. Now let* $\pi(j) = j'$ *and* $\pi(k) = j'$ *for some* $j' \in \mathbb{N}$ *with* $1 \leq j' \leq i'$*. Then* $E'@j' = e = e'$*, contradicting* $e \neq e'$*.*

Table A.1: Equivalence proof for linearizability: ($\rightarrow$)

Show:	for all $(E, i) \in \mathcal{E}^\omega \times \mathbb{N}$:	
	if lin (E,i) then there is $(E', i') \in (\mathcal{E}^\omega \times \mathbb{N})$: s.t. $(E, i) \sim_{\cap_\mathcal{A}} (E', i')$ and $(E', i') \vDash$ *correct*.	
1.	$(E, i) \in \mathcal{E}^\omega \times \mathbb{N}$	hyp.
2.	lin (E,i)	hyp.
3.	$(E', i') \in \mathcal{E}^\omega \times \mathbb{N}$ and $(E, i) \preceq_{lin} (E', i')$ and $(E', i') \vDash$ correct	2,def. lin
4.	$(e, e') \in obs(E, i)$	hyp.
5.	$j, k \in \mathbb{N}$ and $E@j = e$ and $E@k = e'$ and $e \in \text{RET}$ and $e' \in \text{CALL}$ and $j < k \leq i$	4,def.obs,def.pos
6.	$\pi : \{1, \ldots, i\} \to \{1, \ldots, i'\}$ is a bijective function and $E'@\pi(j) = e$ and $E'@\pi(k) = e'$ and $\pi(j) < \pi(k)$	3,5,def. $\preceq_{lin}$,def. $\preceq_{real}$
7.	$\pi(j) < \pi(k) \leq i'$	5,6,def. π
8.	$(e, e') \in obs(E', i')$	5,6,7,def. obs
9.	$obs(E, i) \subseteq obs(E', i')$	4,8,def. $\subseteq$
10.	$(E, i) \sim_{obs} (E', i')$	9, def. $\sim_{obs}$
11.	$(E, i) \sim_{\cap_{\text{THREADS}}} (E', i')$	3, def. $\preceq_{lin}$
11.	$(E, i) \sim_{\cap_\text{A}} (E', i')$	10,11,def. $\sim_{\cap_\text{A}}$
12.	there is $(E', i') \in (\mathcal{E}^\omega \times \mathbb{N})$ s.t. $(E, i) \sim_{\cap_\mathcal{A}} (E', i')$ and $(E', i') \vDash$ *correct*	3,11
13.	for all $(E, i) \in \mathcal{E}^\omega \times \mathbb{N}$:	
	if lin (E,i) then there is $(E', i') \in (\mathcal{E}^\omega \times \mathbb{N})$: s.t. $(E, i) \sim_{\cap_\mathcal{A}} (E', i')$ and $(E', i') \vDash$ *correct*.	1,2,12

Table A.2: Equivalence proof for linearizability: ($\leftarrow$)

Show:	For all $(E,i) \in \mathcal{E}^\omega \times \mathbb{N}$: if unique then if there is $(E',i') \in \mathcal{E}^\omega \times \mathbb{N}$: s.t. $(E,i) \sim_{\cap_\mathcal{A}} (E',i')$ and $(E',i') \vDash \mathit{correct}$ then lin (E,i).	
1.	$(E,i) \in \mathcal{E}^\omega \times \mathbb{N}$	hyp.
2.	for all E $\in \mathcal{E}^\omega$ and all $j,k \in \mathbb{N}$ if $E@j = E@k$ then $j = k$.	hyp.
3.	$(E',i') \in \mathcal{E}^\omega \times \mathbb{N}$ and $(E,i) \sim_{\cap_\mathcal{A}} (E',i')$ and $(E',i') \vDash \mathit{correct}$	hyp.
4.	$(E,i) \sim_{\cap_{\text{THREADS}}} (E',i')$	3,def. $\sim_{\cap_\mathcal{A}}$
5.	$\pi : \{1,\dots,i\} \to \{1,\dots,i'\}$ is a bijective function and for all $j \in \mathbb{N}$ such that $1 \le j \le i : E@j = E'@\pi(j)$	1,3,4,lemma 3
6.	$j,k \in \mathbb{N}$ and $E@j \in$ RET and $E@k \in$ CALL and $j < k \le i$	hyp.
7.	$e \in$ RET and $e' \in$ CALL and pos(e,E) = j and pos(e',E) = k	6
8.	$(e,e') \in \mathit{obs}(E,i)$	6,7
9.	$\mathit{obs}(E,i) \subseteq \mathit{obs}(E',i')$	3, def. $\sim_{\cap_\mathcal{A}}$, def. $\sim_{obs}$
10.	$(e,e') \in \mathit{obs}(E',i')$	
11.	pos(e,E') < pos(e',E') $\le i'$	10
12.	for all $j \in \{1,\dots,i'\}$ there is $k \in \{1,\dots,i\}$: $j = \pi(k)$	5,π surjective
13.	$j',k' \in \{1,\dots,i\}$ and $E'@\pi(j') = e$ and $E'@\pi(k') = e'$ and $\pi(j') < \pi(k') \le i'$	11,12
14.	$E'@\pi(j) = e$ and $E'@\pi(k) = e'$	5,7
15.	$E'@\pi(j') = E'@\pi(j)$ and $E'@\pi(k') = E'@\pi(k)$	13,14
16.	$\pi(j') = \pi(j)$ and $\pi(k') = \pi(k)$	2,15
17.	$\pi(j) < \pi(k) \le i'$	
18.	for all $j,k \in \mathbb{N}$ if $E@j \in$ RET and $E@k \in$ CALL and $j < k$ then $\pi(j) < \pi(k)$	6,17
19.	$(E,i) \preceq_{real} (E',i')$	5,18
20.	For all $(E,i) \in \mathcal{E}^\omega \times \mathbb{N}$: if unique then if there is $(E',i') \in \mathcal{E}^\omega \times \mathbb{N}$: s.t. $(E,i) \sim_{\cap_\mathcal{A}} (E',i')$ and $(E',i') \vDash \mathit{correct}$ then lin (E,i).	1-4,19

Appendix B

Axiomatic Characterization

Proposition 4 *K_t for $t \in$ Threads, K_{obs} and D_G for $G \subseteq \mathcal{A}$ are valid modalities for knowledge, that is, in our framework, they are sound and complete with respect to the axioms that are standardly expected to hold for epistemic modalities.*
Consider the following axioms for a knowledge modality K:
***(K)**: $\vdash K(\varphi \to \psi) \to (K\varphi \to K\psi)$ (Kripke's Law)*
***(T)**: $\vdash K\varphi \to \varphi$ (Truth axiom)*
***(4)**: $\vdash K\varphi \to KK\varphi$ (positive introspection)*
***(5)**: $\vdash \neg K\varphi \to K\neg K\varphi$ (negative introspection)*
***(N)**: if $\vdash \varphi$ then $\vdash K\varphi$ (necessitation)*
where $\vdash$ designates the syntactic deduction relation.

- *K_t for $t \in$ Threads is sound and strongly complete with respect to the axiom system **S5** comprising axioms **(K)**, **(T)**, **(4)**, **(5)**, **(N)**, modus ponens and all instances of propositional tautologies.*

- *K_{obs} is sound and strongly complete with respect to the axiom system **S4** comprising axioms **(K)**, **(T)**, **(4)**, **(N)**, modus ponens and all instances of propositional tautologies. Both notions thus correspond to standard notions of knowledge.*

- *D_G inherits the properties of **S4**. Additionally, we get the following axioms:*
  For all $a \in \mathcal{A} : \vdash D_{\{a\}}\varphi \leftrightarrow K_a\varphi$ and for all $G, G' \subseteq \mathcal{A}$: $\vdash D_G\varphi \to D_{G'}\varphi$ if $G \subseteq G'$.

Proof 7 *Since $\sim_t$ for $t \in$ Threads is an equivalence relation and $\sim_{obs}$ is a partial order, we can apply the standard proof by constructing a canonical model*

(cf., [7, chapter 3.1] or [3, pp. 190-205]). For the properties of distributed knowledge see [7, pp. 73f].

Appendix C

S4 vs. S5

Under sequential consistency, agents can decide by communicating whether a trace is sequentially consistent or not. Under linearizability, they lose this ability. Whenever a trace is not linearizable, they jointly know it is not. For linearizable traces, they cannot be sure. There exist traces, that to the agents, might either be linearizable or not.

$\mathrm{T} := \textsc{Threads}$ $\qquad$ $\mathrm{seq} := \neg D_T \neg \text{ correct}$

$\mathcal{A} := \textsc{Threads} \cup \{\mathrm{obs}\}$ $\qquad$ $\mathrm{lin} := \neg D_{\mathcal{A}} \neg \text{ correct}$

Sequential Consistency

By (4):= $D\varphi \to DD\varphi$:

for all $(E,i) : (E,i) \vDash D_T \neg \text{ correct} \to D_T D_T \neg \text{ correct}$ iff

for all $(E,i) : (E,i) \vDash \neg \text{ seq} \to D_T \neg \text{ seq}$

By (T) := $D\varphi \to \varphi$, we get:

for all $(E,i) : (E,i) \vDash D_T \neg \text{ seq} \to \neg \text{ seq}$

$$\boxed{\text{for all } (E,i) : (E,i) \vDash \neg \text{ seq} \leftrightarrow D_T \neg \text{ seq}}$$

By (5) := $\neg D \neg \varphi \to D \neg D \neg \varphi$, we get:

for all $(E,i) : (E,i) \vDash \neg D_T \neg \text{ correct} \to D_T \neg D_T \neg \text{ correct}$ iff

for all $(E,i) : (E,i) \vDash \text{seq} \to D_T \text{ seq}$

By (T):

for all $(E,i) : (E,i) \vDash D_T \text{ seq} \to \text{seq}$

$$\boxed{\text{for all } (E,i) : (E,i) \vDash \text{seq} \leftrightarrow D_T \text{ seq}}$$

This means:

- By communicating what they observe, the threads can decide whether or not any given trace is sequentially consistent.

Linearizability

In linearizability we lose (5) because of the observer:
not for all $(E,i) : (E,i) \vDash \text{lin} \to D_{\mathcal{A}}\ \text{lin}$ iff

$$\boxed{\text{there is } (E,i) : (E,i) \vDash \text{lin} \wedge \neg D_{\mathcal{A}}\ \text{lin}}$$

by (4) and (T), just as above:

$$\boxed{\text{for all } (E,i) : (E,i) \vDash \neg\ \text{lin} \leftrightarrow D_{\mathcal{A}} \neg\ \text{lin}}$$

This means:

- The agents lose their ability to recognize consistent traces.
- That is, there are traces that are in fact linearizable but for which the agents cannot be sure that they are, even after having told each other everything they saw of the trace.
- If, however, a trace is not linearizable, the agents jointly know.
- In other words, for a non-linearizable trace, the agents spot that the trace is wrong; for some linearizable traces, the agents cannot be not sure. For all they know, these traces might either be right or wrong.

Appendix D

Equivalence Proof for TSO

Equivalence Proof

We formalize balancedness as:

$$
\begin{aligned}
\text{balanced} := \quad & \forall t \forall a \forall v \forall t' \forall a' (\\
& \text{stBeforeFl} := \quad \boxminus(\text{flush}(t,a,v) \rightarrow \text{stored}(t,a,v)) \\
& \text{stUnique} := \quad \boxminus\, \text{store}(t,a,v) \rightarrow \ominus\neg\text{stored}(t',a',v) \wedge \\
& \text{flUnique} := \quad \boxminus\, \text{flush}(t,a,v) \rightarrow \ominus\neg\text{flushed}(t'a'v) \\
&) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad
\end{aligned}
$$

Theorem 4 (Equivalence) *The operational TSO semantics and the logical description are equivalent:* $E \upharpoonright i \in \mathcal{T}_{TSO}$ *iff* $(E,i) \vDash \mathit{TSOCorrect} \wedge \mathit{balanced}$.

Proof 8 *"→":*
Assume $E \upharpoonright i \in \mathcal{T}_{TSO}$. *We show a series of lemmas from which the theorem follows. We first show that traces of the operational model are balanced, then that the trace is correct.*

Lemma 4 (stBeforeFl) *If a value has been flushed, it must have been stored before:* $(E,i) \vDash \forall t \forall a \forall v\ (\boxminus(\mathit{flush}(t,a,v) \rightarrow \mathit{stored}(t,a,v)))$.

Proof 9 *Assume* $(E,k) \vDash \mathit{flush}(t,a,v)$ *and* $k \leq i$. *For a contradiction, assume* $(E,k) \vDash \neg\ \mathit{stored}(t,a,v)$. *Since [Flush] was executable, before* $\mathit{flush}(t,a,v)$, *we must have had* $b(t) = \alpha = \alpha_1 \cdot (a,v)$. *But since only* $\mathit{st}(t,a,v)$ *can add* (a,v) *to* t*'s store buffer and store buffers are initially empty,* $\mathit{st}(t,a,v)$ *must have been executed before. But then there is* $j < k$ *s.t.* $(E,j) \vDash \mathit{store}(t,a,v)$, *contradicting our assumption.*

Lemma 5 (Store unique) *Each value is stored only once:* $(E,i) \vDash \forall t \forall a \forall v \forall t' \forall a' (\boxminus$ *store*$(t,a,v) \rightarrow \ominus \neg$*stored(t',a',v))*.

Proof 10 *Assume* $(E,k) \vDash$ *store*(t,a,v)*, with* $k \leq i$*. For a contradiction assume* $(E,j) \vDash$ *store*(t',a',v) *and* $j < k$*. After executing* *st*(t',a',v)*, we have* $v \in S$*. But then* *st*(t,a,v) *cannot have been executed.*

Lemma 6 (Flush unique) *Each value is only flushed once:* $(E,i) \vDash \forall t \forall a \forall v \forall t' \forall a' (\boxminus$ *flush*$(t,a,v) \rightarrow \ominus \neg$*flushed(t',a',v))*.

Proof 11 *Assume* $(E,k) \vDash$ *flush*(t,a,v) *and* $(E,j) \vDash$ *flush*(t',a',v) *with* $j < k \leq i$*. We distinguish two cases. Assume that* $t = t'$ *and* $a = a'$*. By Lemma 4, we get* $(E,k') \vDash$ *store*(t,a,v) *and* $k' < k \leq i$ *and by Lemma 5 there is no* $j' \neq k'$ *s.t.* $(E,j') \vDash$*store*(t,a,v) *and* $j' < k$*. Now after executing the first* *flush*(t,a,v)*,* (a,v) *is removed from the store buffer and since* *st*(t,a,v) *was not executed a second time, there can be no second instance of* (a,v) *in the buffer and* *flush*(t,a,v) *cannot be pulled.*
Now assume that $t \neq t'$ *or* $a \neq a'$*. Then by Lemma 4, we get* $(E,k') \vDash$ *store*(t,a,v) *and* $(E,j') \vDash$ *store*(t',a',v) *with* $j' \neq k'$ *and* $j',k' < k \leq i$*. But then, we get a contradiction by Lemma 5.*

Lemma 7 (Correctness) *Whenever a value* v *is read by a thread* t *from an address* a*,* v *is either the latest entry in* t*'s store buffer or the latest value written to the memory:* $\forall t \forall a \forall v (\boxminus($*load(t,a,v)* $\rightarrow$ *(ldBuff*$(t,a,v) \ \vee$ *ldMem*$(t,a,v))))$.

Proof 12 *Assume* $(E,k) \vDash$ *load*(t,a,v)*. The label* *ld*(t,a,v) *could have been caused either by a transition using [LD-Buff] or [LD-Mem].*

1. *Suppose it was caused by [LD-Buff]. We show that* $(E,k) \vDash$ *ldBuff*(t,a,v)*. We start by showing* $(E,k) \vDash \neg$ *flushed*(t,a,v)*. For a contradiction, assume* $(E,j) \vDash$ *flush*(t,a,v) *with* $j < k \leq i$*. Since* *ld*(t,a,v) *was caused by [LD-Buff], we have* $b(t) = \alpha$*, with* $(a,v) \in \alpha$ *before the execution. By lemma 4 and* $(E,j) \vDash$ *flush*(t,a,v)*, we get* $(E,j') \vDash$ *store*(t,a,v) *with* $j' < j$*. There can be no other* *st*(t,a,v)*, because after the first store, we would have* $v \in S$*, and the second one would not have been executed. After* *flush*(t,a,v)*,* (a,v) *is removed from* α*, so* $(a,v) \notin \alpha$*, a contradiction.*

   Now to show that $(E,k) \vDash \forall v' (\neg$*store*$(t,a,v')$ S *store*$(t,a,v)))$*, we show that* $(E,j) \vDash$ *store*(t,a,v) *with* $j < k$*. Suppose there was no such store. Then* (a,v) *could not be in* α*, since only* *st*$(t,a,\_)$ *can add values to* t*'s*

store buffer. But then we have $(a, v) \notin \alpha$*, contradicting our assumption that* $ld(t, a, v)$ *was caused by [LD-Buff].*
We now show that for all $j < j' \leq i$*:* $(E, j') \vDash \neg\ store(t, a, v')$*. Suppose that* $(E, j') \vDash store(t, a, v')$ *and* $j < j' \leq i$*. But since load was caused by [LD-Buff], before executing* $ld(t, a, v)$*, we must have* $b(t) = \alpha_1(a, v)\alpha_2$*, with* $(a, \_) \notin \alpha_1$*. But since* $st(t, a, v')$ *took place after* $st(t, a, v)$*, we have* $(a, v') \in \alpha_1$*, which leads us to a contradiction.*

2. *Suppose it was caused by [LD-Mem]. We show that* $(E, k) \vDash ldMem(t, a, v)$*.*
We start by showing $(E, k) \vDash \forall v'(\neg buffered(t, a, v'))$*. For a contradiction assume* $(E, k) \vDash stored(t, a, v') \wedge \neg\ flushed(t, a, v')$*. But then, before executing* $ld(t, a, v)$ *we have* $b(t) = \alpha$ *with* $(a, v') \in \alpha$*, because* (a, v') *was added to t's store buffer by* $st(t, a, v')$*, and the only way it could have been removed is by* $flush(t, a, v')$*, which we know didn't occur. But to execute [LD-Mem], we must have* $(a, v') \notin \alpha$*, from which we get a contradiction.*
To show $(E, k) \vDash \forall v'(\neg(\exists t'\ flush(t', a, v'))\ \ S\ \exists t\ flush(t, a, v))$*, we first show that* $(E, j) \vDash \exists t'\ flush(t', a, v)$*, with* $j < k$*. Since initially, the memory is empty, and [FLUSH] is the only rule that changes the memory state, we must have had* $flush(\_, a, v)$*, because* $a \in dom(m)$*. That is, we have* $(E, j) \vDash flush(t', a, v)$*, for some t' and* $j \leq k$*. Now for a contradiction suppose that* $(E, l) \vDash flush(t'', a, v')$ *and* $j < l \leq k$ *for some* t''*. Suppose* $v = v'$*, this cannot be by lemma 6. Now suppose* $v \neq v'$*. This means we would have* $m(a) = v'$*. But this contradicts* $m(a) = v$*.*

Lemma 8 (FlushOrder) *Stores are flushed in FIFO-order:* $(E, i) \vDash \forall t \forall a \forall v \forall a' \forall v'(\neg store(t, a, v)\ S\ store(t, a', v') \rightarrow \neg\ flush(t, a, v)\ S\ flush(t, a', v'))$*.*

Proof 13 *Assume* $(E, i) \vDash \neg store(t, a, v)\ S\ store(t, a', v')$*.*
We have $(E, j') \vDash store(t, a', v')$*. Since for all TSO traces the store buffer must finally be flushed, we have* $(E, k') \vDash flush(t, a', v')$*, with* $j' < k' \leq i$*. For a contradiction, assume that* $(E, k) \vDash flush(t, a, v)$*, with* $k' < k \leq i$*. We make a case distinction. Suppose that* $v = v'$*. This gives us a contradiction by Lemma 6. Now suppose that* $v \neq v'$*. Then we get* $(E, j) \vDash store(t, a, v)$*, with* $j < k \leq i$ *by Lemma 4, and* $j < j'$ *by our assumption. Now before* $flush(t, a', v')$ *is executed, we must have* $b(t) = \alpha_1(a', v')$*. But since* $j < j'$ *(i.e.* (a, v) *entered the store buffer before* (a', v')*) and* $j' < k' < k$ *(i.e. values have not been flushed), the store buffer must have the form* $b(t) = \alpha_1(a', v')\alpha_2(a, v)\alpha_3$ *– a contradiction.*

Proof 14 *"←": We first establish another lemma. We then show that we can justify each action by an inference rule where all preconditions are satisfied. We show that the computation is complete.*

Lemma 9 (Completeness) *Stores are flushed:* $(E,i) \vDash \mathit{FlushOrder} \wedge$ *balanced and* $(E,j) \vDash \mathit{store}(t,a,v)$ *with* $j \leq i$ *then there is* k *s.t.* $j < k \leq i$ *and* $(E,k) \vDash \mathit{flush}(t,a,v)$.

Proof 15 *We first show that* $(E,i) \vDash \neg \mathit{store}(t,a,v) S \mathit{store}(t,a,v)$. *We need to show that for all* j' : *if* $j < j' \leq i$ *then* $(E,j') \vDash \neg\ \mathit{store}(t,a,v)$. *Suppose there was* j' *s.t.* $(E,j') \vDash \mathit{store}(t,a,v)$ *and* $j < j' \leq i$. *Then we would get* $(E,j') \vDash \bar{\circ} \neg \mathit{stored}(t,a,v)$ *by* $(E,i) \vDash \mathit{stUnique}$ *and hence* $(E,j) \vDash \neg \mathit{stored}(t,a,v)$, *contradicting our assumption.*
We can now apply $(E,i) \vDash \mathit{FlushOrder}$ *to yield* $(E,i) \vDash \neg \mathit{flush}(t,a,v) S \mathit{flush}(t,a,v)$, *from which we get* $(E,k) \vDash \mathit{flush}(t,a,v)$, *with* $k \leq i$. *Now suppose that* $k < j$. *But then, by* $(E,i) \vDash \mathit{stBeforeFl}$, *we have* $(E,j') \vDash \mathit{store}(t,a,v)$, *with* $j' < k < j$. *But by* $(E,i) \vDash \mathit{stUnique}$ *and* $(E,j) \vDash \mathit{store}(t,a,v)$, *we get* $(E,j') \vDash \neg \mathit{stored}(t,a,v)$ *– a contradiction.*

- [LD-Buff]*: Assume* $(E,k) \vDash \mathit{load}(t,a,v)$, $(E,k) \vDash \mathit{ldBuff}(t,a,v)$, *and* $k \leq i$. *To show that [LD-Buff] is executable, we show that before* $\mathit{ld}(t,a,v)$, $b(t) = \alpha = \alpha_1(a,v)\alpha_2$, *where* $(a,\_) \notin \alpha_1$. *First, we show that* $(a,v) \in \alpha$. *From* $(E,k) \vDash \neg \mathit{store}(t,a,v) S \mathit{store}(t,a,v)$, *we get* $(E,j) \vDash \mathit{store}(t,a,v)$ *for some* $j < k \leq i$. *We also have* $(E,k) \vDash \neg\ \mathit{flushed}(t,a,v)$. *This means* (a,v) *was written to* α, *and since* $\mathit{flush}(t,a,v)$ *has not yet been executed, we have* $(a,v) \in \alpha$.
  We now show α_1 *does not contain* (a,v') *for some* v'. *Suppose it did, then we would have* $(E,j') \vDash \mathit{store}(t,a,v')$, *with* $j < j' < k$. *But from* $(E,k) \neg \mathit{store}(t,a,v) S \mathit{store}(t,a,v)$, *we get* $(E,j') \vDash \neg\ \mathit{store}(t,a,v')$ *from which we get a contradiction.*

- [LD-Mem]*: Assume* $(E,k) \vDash \mathit{load}(t,a,v)$, $(E,k) \vDash \mathit{ldMem}(t,a,v)$, *and* $k \leq i$. *We first show that* $b(t) = \alpha$ *does not contain* (a,v') *for some* v'. *Suppose it did, then there must be* $(E,j) \vDash \mathit{store}(t,a,v')$ *with* $j < k$. *But we can rewrite* $(E,k) \vDash \neg \mathit{buffered}(t,a,v')$ *as* $(E,k) \vDash \mathit{stored}(t,a,v') \rightarrow \mathit{flushed}(t,a,v')$, *so* (a,v') *was already flushed and no longer in* α.
  We show that $a \in \mathit{dom}(m)$ *and* $m(a) = v$. *By* $(E,k) \vDash \mathit{globallyLatest}(a,v)$, *we get* $(E,j) \vDash \mathit{flush}(t',a,v)$ *for some* t' *and* $j < k$, *giving us* $a \in \mathit{dom}(m)$. *Suppose that* $m(a) \neq v$, *then we must have* $(E,j') \vDash \mathit{flush}(t'',a,v')$, *with*

$v \neq v'$ *and* $j < j' < k$. *But by* $(E,k) \vDash$ *globallyLatest*(a,v), *we have* $(E,j') \vDash \neg$ *flush*(t'',a,v') – *a contradiction.*

- [ST]*: Assume* $(E,k) \vDash$ *store*(t,a,v), *with* $k \leq i$. *We show that* $v \notin S$. *For a contradiction, assume that* $v \in S$. *Then there must be* $(E,j) \vDash$ *store*(t',a',v), *with* $j < k \leq i$. *From* $(E,i) \vDash$ *stUnique, we get* $(E,j) \vDash \neg$ *store*(t',a',v) – *a contradiction.*

- [FLUSH] *Assume* $(E,k) \vDash$ *flush*(t,a,v) *with* $k \leq i$. *We show that* $b(t) = \alpha = \alpha_1 \cdot (a,v)$. *We first show that* $(a,v) \in \alpha$. *By* $(E,i) \vDash$ *stBeforeFl, we have* $(E,j) \vDash$ *store*(t,a,v) *with* $j < k \leq i$ *and* $(E,k) \vDash \overline{\circ}\neg$*flushed*$(t,a,v)$, *so* (a,v) *has been stored to* t*'s store buffer, but not flushed yet.*
  We show that $\alpha = \alpha_1 \cdot (a,v)$. *For a contradiction, assume that* $\alpha = \alpha_1 \cdot (a,v)\alpha_2(a',v')$. *Then we must have* $(E,j') \vDash$ *store*(t,a',v') *with* $j' < j < k \leq i$ *and* $(E,k) \vDash \neg$ *flushed*(t,a',v'). *Now by lemma 9, we get* $(E,k') \vDash$ *flush*(t,a',v'), *with* $j' < k' \leq i$. *We make a case distinction. Suppose that* $k' < k$. *We get a contradiction from* $(E,k) \vDash \neg$ *flushed*(t,a',v'). *Now suppose that* $k < k'$. *We show that we can derive a contradiction by* $(E,i \vDash)$ *FlushOrder.*

  We show that $(E,i) \vDash \neg$*store*$(t,a',v')S$ *store*(t,a,v). *Suppose there was* $(E,j'') \vDash$ *store*(t,a',v'), *with* $j < j'' \leq i$. *This leads to a contradiction by* $(E,i) \vDash$ *stUnique. Now by* $(E,i) \vDash$ *FlushOrder, and* $(E,i) \vDash \neg$*store*$(t,a',v')S$ *store*(t,a,v), *we get* $(E,i) \vDash \neg$ *flush*$(t,a',v')S$ *flush*(t,a,v). *But then by* $(E,i) \vDash$ *flUnique, there can be no* $l \neq k \leq i$ *s.t.* $(E,l) \vDash$ *flush*(t,a,v), *so we get* $(E,k') \vDash \neg$ *flush*(t,a',v') *and we are done.*

- Completeness*: We show that after the last action* $E@i$, $b(t) = \epsilon$, *for all* t. *For a contradiction, assume that* $(a,v) \in b(t)$, *for some* t. *Then we must have* $(E,i) \vDash$ *stored*(t,a,v), *and by lemma 9:* $(E,i) \vDash$ *flushed*(t,a,v), *so* $(a,v) \notin b(t)$.

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *POPL*, pages 7–18. ACM, 2010.

[3] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal logic*. Cambridge Univ. Press, 2001.

[4] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Concurrent library correctness on the TSO memory model. In *ESOP*, pages 87–107. Springer, 2012.

[5] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *CAV*, pages 107–120. Springer, 2008.

[6] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. Technical Report MSR-TR-2008-12, Microsoft Research, 2008.

[7] Ronald Fagin, Joseph Y. Halpern, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, 2003.

[8] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208 – 1244, 1997.

[9] Alexey Gotsman and Hongseok Yang. Liveness-preserving atomicity abstraction. In *ICALP*, pages 453–465. Springer, 2011.

[10] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[11] Maurice P. Herlihy and Jeannette M. Wing. Axioms for concurrent objects. In *POPL*, pages 13–26. ACM, 1987.

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

[13] Yoichi Hirai. An intuitionistic epistemic logic for sequential consistency on shared memory. In *LPAR*, pages 272–289. Springer, 2010.

[14] Simon Kramer. *Logical Concepts in Cryptography.* PhD thesis, EPFL, 2007.

[15] Simon Kramer. A logic of interactive proofs. Available online: `http://arxiv.org/pdf/1201.3667v2.pdf`, 2012.

[16] Simon Kramer and Andrey Rybalchenko. A multi-modal framework for achieving accountability in multi-agent systems. In *LIS*, pages 148–174, 2010.

[17] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[18] Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. The glory of the past. In *Conference on Logic of Programs*, pages 196–218. Springer, 1985.

[19] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso (extended version). Technical Report UCAM-CL-TR-745, University of Cambridge, 2009.

[20] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[21] R. Ramanujam. Locally linear time temporal logic. In *LICS*, pages 118–128. IEEE, 1996.

[22] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

[23] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, 1996.

[24] Pradeep S. Sindhu and Jean-Marc Frailong. *Formal specification of memory models*, pages 25–42. Kluwer, 1991.

[25] Viktor Vafeiadis. Automatically proving linearizability. In *CAV*, pages 450–464. Springer, 2010.

[26] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.