Specifying and Verifying Persistent Transactional Memory

by

Eleni Vafeiadi Bila

Submitted for the Degree of Doctor of Philosophy

Department of Computer Science
Faculty of Engineering and Physical Sciences
University of Surrey

Supervisor: Prof Brijesh Dongol
Co-Supervisor: Prof Gregory Chockler

16 June 2023

Declaration

This thesis and the work to which it refers are the results of my own efforts. Any ideas, data, images, or text resulting from the work of others (whether published or unpublished) are fully identified as such within the work and attributed to their originator in the text, bibliography, or in footnotes. This thesis has not been submitted in whole or in part for any other academic degree or professional qualification. I agree that the University has the right to submit my work to the plagiarism detection service TurnitinUK for originality checks. Whether or not drafts have been so-assessed, the University reserves the right to require an electronic version of the final document (as submitted) for assessment as above.

Signature: Eleni Vafeiadi Bila
Date: 16 June 2023

Acknowledgements

First and foremost, I would like to thank my supervisor, Brijesh Dongol, for his generous guidance and continuous support throughout my PhD. I am grateful for him introducing me to the world of formal methods, giving me the opportunity to work on very interesting research topics, and helping me evolve my research skills. Many of the ideas in this thesis are the product of our fruitful discussions, enriched by his insightful comments and expertise.

I am indebted to Heike Wehrheim, Gerhard Schellhorn, Simon Doherty, and John Derrick, with whom I collaborated in the early stages of my PhD. Our discussions have been particularly helpful and insightful. This initial collaboration provided the motivation to propel my research forward. I would also like to thank Ori Lahav, Azalea Raad, and John Wickerson with whom I have collaborated on the second project of my PhD. It was an enjoyable and productive experience, and I learned a great deal from our meetings. I would like to thank my co-supervisor Gregory Chockler for helping with reviewing my progress.

Thank you to Simon Doherty, Sadegh Dalvandi, and Mikhail Semenyuk for helping me familiarise myself with Isabelle/HOL and giving me useful advice on proof mechanization.

I would like to thank my examiners, Viktor Vafeiadis (MPI-SWS) and Sasa Radomirovic (University of Surrey), for their insightful corrections and suggestions.

This thesis would not have been possible without the funding provided by VeTSS.

I am immensely indebted to my parents, Vassilia and Stamatis, who have inspired me to pursue my goals since childhood and whose support has been invaluable. I am also indebted to my sister Victoria, who has been by my side through both the joyful and challenging moments of this endeavor. Finally, I would like to thank my partner Nick, whose enthusiasm and joy for his own research and hobbies has been an endless source of inspiration for me. Without him, my PhD journey would have been much different.

Abstract

This thesis sheds light on key aspects of Persistent Software Transactional Memory.

Persistent memory is a novel memory paradigm that retains its contents even in the event of power loss. It is widely expected to become ubiquitous, and hardware architectures are already providing support for persistent memory programming. However, writing persistent programs is extremely challenging, as it requires the programmer to keep track of which memory writes have become persistent and which have not. This is further complicated in a multi-threaded setting by the intricate interplay between the rules of memory persistency (which determine the order in which writes become persistent) and those of memory consistency (which determine what data can be observed by which threads).

Software Transactional Memory (STM) [132] has emerged as a highly promising solution for concurrency control in multi-threaded environments, allowing programmers to concentrate on algorithm design rather than intricate locking mechanisms. Recent research efforts focus on integrating durability into transactions. Persistent STMs aim to accomplish more than just thread synchronization; they also endeavor to maintain persistent memory in a consistent state. Implementing such mechanisms can greatly benefit developers, as they would no longer be burdened with the responsibility of persisting coherently memory locations. However, due to their innate complexity, persistent STMs are susceptible to errors, necessitating careful design and thorough testing. In this thesis, we are approaching the problem of persistent STMs correctness from a formal methods perspective.

At the core of this research two main topics are addressed. The first topic concerns the nature of software transactional memory correctness in the face of persistency. To this end, we present a novel definition of software transactional memory correctness, durable opacity [23], which adapts opacity to the persistent memory setting.

The second topic concerns the verification of persistent transactional memory algorithms. We aim to investigate how existing verification techniques designed for volatile memory algorithms can be adapted and applied in the context of persistent memory. In our initial work [23], we attempt to verify a durably opaque version of an STM algorithm, TML [38]. In this first endeavor, we assume a simplified model, Persistent Sequential Consistency (persistent SC), which lays the groundwork for understanding verification challenges unique to persistency. The proposed proof technique constitutes an adaptation to persistency of the verification method demonstrated in [44]. In our second work, we are focusing on developing an Owicki–Gries program logic (Pierogi) for reasoning about x86 code that uses low-level operations for controlling persistency such as fences and flushes. Our logic is able to accommodate the persistency features and weak behaviors induced by the Px86 model. We exemplify the utility of Pierogi by verifying a number of persistent x86 litmus tests. In our final work, we adapt our initial STM implementation to the realistic Px86 model and show that it is durably opaque. Our correctness proof is operational and builds on the Pierogi logic as well as the simulation-based proof technique demonstrated in our first work. As far as we are aware, this is the first application of simulation-based proofs for persistent x86 programs.

Our entire development has been mechanized in the Isabelle/HOL proof assistant.

List of publications

[23] E. Bila, S. Doherty, B. Dongol, J. Derrick, G. Schellhorn, and H.Wehrheim, "Defining and verifying durable opacity: Correctness for persistent software transactional memory," in Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings (A. Gotsman and A. Sokolova, eds.), vol. 12136 of Lecture Notes in Computer Science, pp. 39–58, Springer, 2020.

This work contributes to Chapter 3 of this thesis and received the Best Paper Award at FORTE'20.

[22] E. Bila, J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, and H. Wehrheim, "Modularising verification of durable opacity," Logical Methods in Computer Science, vol. 18, 2022.

This work is not part of my PhD thesis as my contribution was small, and I did not continue this line of research.

[24] E. V. Bila, B. Dongol, O. Lahav, A. Raad, and J. Wickerson, "View-based Owicki-Gries reasoning for persistent x86-TSO," in Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (I. Sergey, ed.), vol. 13240 of Lecture Notes in Computer Science, pp. 234–261, Springer, 2022.

This work contributes to Chapter 4 and Chapter 5 of this thesis and received the Distinguished Artifact Award at ESOP'22.

List of artifacts

- The artifact of the work presented in Chapter 4 (PIEROGI$_{\text{simp}}$) [24] and Chapter 5 [24] can be found at https://doi.org/10.6084/m9.figshare.18469103.v2.
- The artifact of the work presented in Chapter 4 (PIEROGI$_{\text{full}}$) and Chapter 6 can be found at https://doi.org/10.6084/m9.figshare.25037312.v2.

Contents

Chapter 1

Introduction

1.1 An Introduction to Persistent Memory

Persistent memory (*PMem*) promises the combination of the density and non-volatility of NAND Flash-based solid-state disks (SSDs) with performance comparable to volatile memory (DRAM). Persistent memory possesses aspects of both storage and memory. Instead, it can be viewed as a third tier that is to be used alongside memory and storage. Persistent memory is well-suited for applications that require frequent access and data retrieval from large, complex datasets as well as applications that are sensitive to the downtime caused by system crashes. Use cases include in-memory databases, virtualization, big data, high-performance scientific computing, cloud computing/IoT, and artificial intelligence applications.

The most commercialized persistent memory, until recently, was Optane DC Persistent Memory, formed by combining DDR DIMMs and non-volatile Optane DC persistent memory modules. The Optane DC persistent memory module is Intel and Micron's 3D XpointDIMM, implemented to coexist with DRAM on the memory bus. Companies such as Cypress/AgigA, Viking, Netlist, and Micron have previously paved the way for the adoption of the Optane DIMM by introducing technologies like the NVDIMM-N module. Additionally, Everspin has demonstrated the MRAM DIMM, which further supports the development of persistent memory solutions.

This thesis primarily uses the term persistent memory (*PMem*) to refer to Optane DC Persistent Memory, while the terms NVDIMM, Optane DIMM, and DC PMM are used interchangeably and refer to the DC persistent memory module. However, it's worth noting that the challenges presented by Optane DC Persistent Memory are likely to be similar across different implementations of persistent memory. Therefore, the methods developed in this thesis can be adapted to address other persistent memory implementations.

PMem provides fast access to critical data while allowing both *byte-addressability* and *persistency*. Byte-addressability enables applications to directly access data without having to issue I/O operations to retrieve data from a storage unit. Such operations are particularly expensive as they require data to be paged from the storage unit or streamed from the network before being copied to the main memory for further processing. Compared to DRAM, persistent memory modules are available in much larger capacities at a lower cost per GB. This means that applications can handle and store a much larger volume of data in place. Therefore, even when not used for its non-volatility feature, persistent memory can significantly decrease the number of disk I/O operations, leading to improved performance.

Persistent memory is placed in between DDR DRAM and NAND SSD in the memory/storage hierarchy in terms of latency cost and capacity. Fig. 1.1 provides estimated latency values for different memory/storage technologies. As depicted, persistent memory offers latency that is comparable to DRAM in terms of speed, measured in nanoseconds, and also provides persistency. On the other hand, block storage offers persistency but has higher latencies, which typically start in the microsecond range and vary depending on the type of technology used.

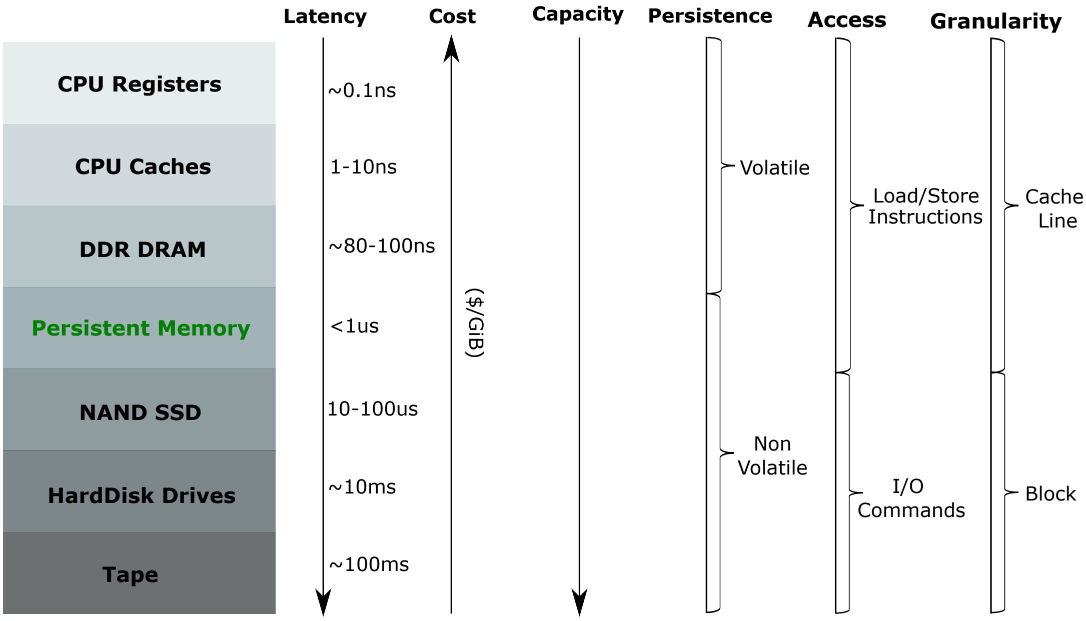


Figure 1.1: Estimated metrics for different memory/storage types as illustrated in [129].

1.2 Abstract Persistency Model

In the literature, numerous memory models have been proposed for persistent memory systems [24, 90, 120–123]. In order to describe the behaviors that can potentially arise within the persistent programming paradigm, we begin by introducing an abstract model of the memory architecture that we assume (Fig. 1.2). In this model, we depict persistent memory as an additional layer to the well-known total store order (TSO) model [135]. In the event of a crash, we expect any data that has not been stored in persistent memory, residing in the store buffers or volatile memory, to be lost.

The persistent memory model in Fig. 1.2 not only encompasses the inherent complexity of TSO, but also incorporates additional complexity pertaining to persistency. In order to understand Fig. 1.2, we begin by recapping the TSO model (§1.2.1). Later, in §1.2.2, we introduce the persistent memory layer, offering a glimpse into the program behaviors one can expect related to persistency.

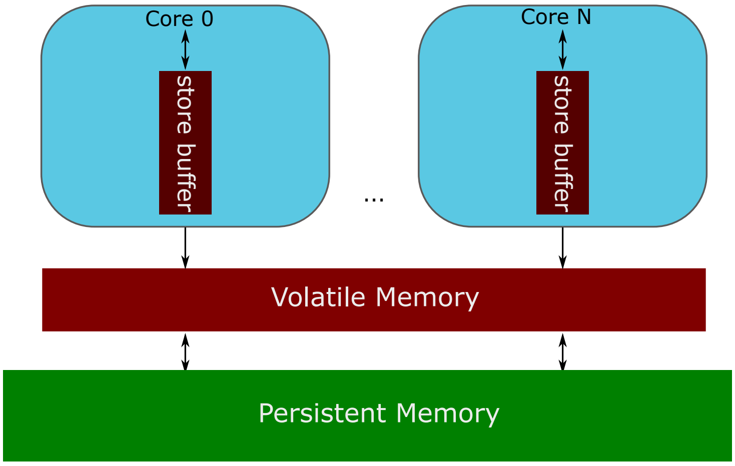


Figure 1.2: The TSO enhanced with persistency memory model. The volatile components are illustrated in red color (store buffers, volatile memory) and the non-volatile components are illustrated in green color (persistent memory).

1.2.1 The Total Store Order Model

TSO aims to reduce write latency by introducing, to each core/thread, a local store buffer. When a thread executes a write, the write is recorded solely in the thread's local buffer. While in the buffer, this write is only visible to the thread that issued it. The buffered writes are eventually debuffered and sent to the memory at a later point in time, following a FIFO order. On the other hand, reads are performed in real-time. When a thread requests to read a location, it first checks its own buffer. If there are recorded writes for this location in the buffer, the thread returns the value of the most recent write.

The above model enables a thread to avoid stalling its execution until a write operation is completed. Instead, it can proceed with subsequent read operations, resulting in a reduction of the overall program latency. However, this model comes with a potential drawback. Due to the fact that the order in which writes become visible to other threads differs from the order in which they are initially issued, it introduces counterintuitive behaviors to the program, which leads to unexpected outcomes and complicates reasoning about correctness. We illustrate some of the most typical patterns occurring in TSO (Fig. 1.3) by demonstrating three examples presented in [131]. In all the examples demonstrated in this thesis, we assume that all locations and registers are 0 in the initial state. Furthermore, we assume that the left thread of concurrent programs has id 1, and the right thread has id 2.

We start with the store buffer pattern (example: SB). In this example, after thread 1 issues **store** x 1, it caches the store to its local store buffer and then proceeds to execute the subsequent read. Similarly, after thread 2 issues **store** y 1, it caches the store to its local store buffer and then proceeds to execute the subsequent read. In this scenario, it is possible for thread 1 to read the value 0 for y while the write operation on y from thread 2 is still pending in its store buffer. Symmetrically, there is a chance that thread

$$\begin{array}{l||l} \textbf{store}\ x\ 1; & \textbf{store}\ y\ 1; \\ a := \textbf{load}\ y & b := \textbf{load}\ x; \\ \multicolumn{2}{c}{a, b \in \{0,1\}} \end{array} \quad (\textsc{sb}) \qquad \begin{array}{l||l} \textbf{store}\ x\ 1; & \textbf{store}\ y\ 1; \\ \textbf{mfence}; & \textbf{mfence}; \\ a := \textbf{load}\ y & b := \textbf{load}\ x; \\ \multicolumn{2}{c}{a = 1 \lor b = 1} \end{array} \quad (\textsc{sb} + \text{mfence})$$

$$\begin{array}{l||l} & a := \textbf{load}\ y; \\ \textbf{store}\ x\ 1; & \textbf{if}\ (a{=}1) \\ \textbf{store}\ y\ 1; & \quad b := \textbf{load}\ x \\ \multicolumn{2}{c}{a = 1 \Rightarrow b = 1} \end{array} \quad (\textsc{mp})$$

Figure 1.3: TSO examples presented in [131]. In all examples x, y, z are distinct locations with initial value 0, and α is a (thread-local) register

2 reads the value 0 for x while the write operation on x from thread 1 is still in its store buffer.

To accommodate this problem, one can control the debuffering process, by utilizing instructions that stall the execution until the buffer is empty. These instructions, such as RMW (Read-Modify-Write) and fences, provide strong ordering guarantees. For instance (example: SB + mfence), placing a memory fence (**mfence**) instruction after the **store** x 1 operation of thread 1 causes the program execution to pause until the preceding writes of thread 1 are fully debuffered. Consequently, after thread 1 executes the **mfence**, the value 1 becomes visible to thread 2 for the location x. Considering the symmetry of thread 2, we can ensure that by the end of the execution, at least one thread has executed its **mfence**, implying either $a = 1$ or $b = 1$.

Another typical pattern of TSO is message passing (example: MP). In this example, the load of value 1 for y in register a from the second thread indicates that the store of 1 at y from the first thread has already been evicted from its store buffer. Since the store of 1 at x precedes it, this write has also been evicted from the store buffer and therefore is visible to the second thread. Because of the message passing synchronization if the second thread reads the last written value at y it can only observe only the last written value at x.

1.2.2 The Total Store Order Model Enhanced with Persistency

In the context of TSO with persistent memory on top (Fig. 1.2), it is essential to consider not only the order in which writes are debuffered from the store buffers but also the order in which the writes transition from volatile to persistent memory. These orders may differ due to multiple layers of cache in volatile memory. The specific configuration of these caches, including factors such as block size, mapping policy, writing policy, and coherence protocol, significantly influences the order in which writes are evicted to persistent memory. In fact, according to Intel's reference manual [1] and as further elaborated by Raad *et al.* [122], when a thread performs two writes on distinct locations, the order in which these writes persist can vary, and there is even a possibility that they might not persist at all. To afford more control regarding persisting writes, Intel, ARM, and other vendors enhanced their instruction set architecture with instructions dedicated

```
store $x$ 1;
store $y$ 1;                (STORE)
↯ :$x, y \in \{0, 1\}$
```

```
store $x$ 1;
flush $x$;
store $y$ 1;                (FLUSH)
↯ :$y{=}1 \Rightarrow x{=}1$
```

```
store $x$ 1;
flush$_{\text{opt}}$ $x$;
store $y$ 1;
↯ : $x, y \in \{0, 1\}$
        (OPT. FLUSH)
```

```
                ||  $a$ := load $y$;
                ||  if ($a{=}1$)
store $x$ 1;    ||    flush$_{\text{opt}}$ $x$;        (FMP)
store $y$ 1;    ||    sfence;
                ||  store $z$ 1;
↯ : $z{=}1 \Rightarrow x{=}1 \wedge y{=}1$
```

Figure 1.4: Example Px86 programs by Raad *et al.* [122] where the assertion ↯ defines the possible persisted values during the execution. In all examples x, y, z are distinct locations with initial value 0, and a is a (thread-local) register.

to explicitly flush all the locations of a specified cache line to persistent memory. For instance, ARMv8 [9] offers the **DC CVAP** instruction, while Px86 [1] provides explicit persist instructions of varying strengths (**flush**, **flush**$_{\text{opt}}$). Despite these advancements, utilizing these instructions effectively requires careful consideration, due to their asynchronous nature and potential performance implications. Analogously to the store buffer level, it is possible to prevent the reordering of the explicit persist instructions' effect by combining them with instructions (persist barriers) that pause the program execution until the locations of the specified cache line have fully reached the persistent memory. For example, ARMv8 provides a *data synchronisation barrier* (**DSB**$_{\text{full}}$), which, unlike the less strong *data memory barrier* (**DMB**$_{\text{full}}$) instruction, ensures that no instruction in program order after this instruction executes until the earlier **DC CVAP** instructions are complete. Likewise, in the Px86 architecture, there are multiple types of persist barriers. These include the **mfence** and RMW instructions, as well as a weaker fence instruction, the store fence (**sfence**).

To provide a brief overview of the behaviors of Intel's different persistent memory instructions, we use four examples (Fig. 1.4) by Raad *et al.* [122]. The assertion at the end of each program (indicated by ↯) expresses the values of the corresponding locations in persistent memory immediately after the occurrence of a crash. In the context of this thesis, a crash is defined as an event that disrupts the normal operation of a computer system. Crashes can occur due to various factors, including hardware failures, software bugs, and power failures. Although the consequences of a crash can differ depending on its cause, our focus here is exclusively on its effects on the underlying memory model. Unless specified otherwise, we will assume that crashes, regardless of their origin, affect the entire system and result in the loss of context for all volatile system components. To this end, in this thesis, we are using the terms power failure, crash, and system crash interchangeably.

In the first example (STORE), the value 1 is first stored on location x, followed by the store of 1 on location y. In this case, there is no guarantee regarding the order in which these stores will be persisted, or if they will persist at all. Therefore, at any point of the execution, x and y can hold the previous or their new written values in persistent memory.

In example FLUSH, we include a **flush** x instruction immediately following the store operation of 1 into x. The purpose of the flush instruction on x is to ensure that all the locations of the cache line that x belongs to, are written back to persistent memory. Consequently, once the **flush** x instruction is executed, the value 1 for x is guaranteed to be stored in persistent memory. Assuming that the subsequent write operation (**store** y 1) also reaches persistent memory, we can conclude that the preceding **flush** x instruction occurred prior to a system crash. Hence, if y is 1 in persistent memory, we can infer that x is also 1.

Performing a $\textbf{flush}_{\text{opt}}$ x instruction (example: OPT. FLUSH) instead of **flush** x, weakens the persistent memory invariant. This is because the *optimized flush* instruction ($\textbf{flush}_{\text{opt}}$), flushes a single cache line but in an asynchronous manner (without blocking the execution of the corresponding thread). Consequently, its effect might take place after the last store (**store** y 1). As a result, even if a crash occurs after y persists, there is no guarantee that x will persist. To restrict the additional weak behaviors that $\textbf{flush}_{\text{opt}}$ introduces, one can use the **sfence** instruction that orders store instructions with $\textbf{flush}_{\text{opt}}$. The $\textbf{flush}_{\text{opt}}$ instruction is guaranteed to take effect (the contents of the given cache line reach the persistent memory) before the execution point of the following **sfence** instruction.

Example FMP constitutes a Px86 message passing example. As in TSO, the load of value 1 for y in register a from the second thread, indicates that the store of 1 at y from the first thread has been already evicted from its store buffer. Since the store of 1 at x precedes it, this write has also been evicted from the store buffer and therefore is visible to the second thread. Thanks to message passing if the second thread reads the last written value at y it is obliged to observe only the last written value at x. The $\textbf{flush}_{\text{opt}}$ x instruction that follows can not be reordered before the preceding load, thus it observes 1 at x. After the execution of the proceeding **sfence**, 1 reaches the persistent memory. If persistent memory obtains 1 for z, it means that no crash occurred until the execution point of **store** z 1 and thus $x = 1$ in persistent memory.

1.2.3 Persistent Memory Models Used in This Thesis

Utilizing persist instructions and persist barriers, when necessary, can help prevent reordering and strengthen persistency guarantees. However, the correct use of these instructions requires a deep understanding of their underlying memory semantics and potential interactions. In this work we approach the verification of durable (persistent) algorithms by considering two memory models of increasing complexity. In particular, we begin by considering a strict memory model, where all memory operations are appeared to be executed in a strictly sequential order. Subsequently, we delve into a weak memory model, characterized by the possibility that different threads may perceive operations occurring in different orders. Below, we provide a short description of the two memory models.

Persistent Sequential Consistency. We initially consider a simplified conceptual memory model (persistent SC) which involves the following considerations: **1)** It ignores the complexity induced by the total store buffers of TSO. This model assumes that all the writes are immediately becoming observable to all the threads after they are issued. **2)** It only supports a synchronous flush instruction. However, a write can still persist asynchronously due to a natural cache-line eviction. Nevertheless, we assume that the only means of controlling the persistence mechanism is through a synchronous flush instruction, which is semantically equivalent to the Px86 **flush** instruction. Consequently, once a location x is flushed, we consider its value to immediately reach the persistent memory without the need for persist barriers. The above model allows us to examine some of the fundamental problems associated with persistent memory verification, such as decoupling persistency from thread synchronization, determining the correct placement of flush instructions, and reasoning about recovery mechanisms. However, it is not a realistic model as it does not account for instruction reordering and the optimizations present in real-world persistent memory systems

Persistent Total Store Order. In this thesis, we will use the terms persistent TSO and Px86 interchangeably. The persistent TSO model (Fig. 1.2) reflects fully the Px86 complexity. The persistent TSO model addresses the limitations of the persistent SC model by considering the following aspects: **1)** Unlike persistent SC, the persistent TSO model takes into account the total store buffers of TSO. These buffers introduce reordering possibilities for writes and play a crucial role in the consistency of the memory system. **2)** The persistent TSO model recognizes that controlling the persistence mechanism goes beyond synchronous flush instructions. To elaborate, it considers additional (asynchronous) persist instructions and persist barriers that interact with the underlying memory model, affecting the order of persists and, thus, the expected content of persistent memory.

The persistent TSO model reflects the Px86 architecture. Our comprehension of this model was primarily obtained by studying the work of Raad *et al.* [122]. In this work, the state comprises three components: the thread-local store buffers, the global (volatile) persistent buffer, which represents volatile memory, and the global persistent memory. In the proposed semantics, the point at which a write exits the store buffer corresponds to when it becomes globally visible, whereas the point at which it exits the persistent buffer corresponds to when it becomes persistent. Although these semantics may align more closely with our intuition, our work builds upon the view-based operational semantics proposed by Cho *et al.* [32]. The Px86$_{\text{view}}$ model which has been shown to be equivalent to Px86 enables one to abstractly reason about underlying architectural complexities in terms of timestamps, resulting to a simpler state.

1.3 Thesis Objective

This thesis concerns the verification of persistent software transactional memory. *Software Transactional Memory* (STM) provides programmers with an easy-to-use synchronization mechanism for concurrent access to shared data. In brief, STM is a programming construct that allows one to specify blocks of code as *transactions*, with properties of database transactions (e.g., atomicity, consistency, and isolation) [70]. *Atomicity* ensures that all changes to data inside a transaction, are performed as they are a single operation - either all occur (commit), or none occur (aborted). *Consistency* ensures that a system's data remains in a valid state before and after the transaction. According to *isolation*, multiple transactions can execute concurrently without interfering with each other. In other words, each transaction must be executed as if it were the only transaction in the system. While locking-based algorithms can scale well, their design can often be challenging. In contrast, STM provides a better balance between scaling and implementation effort.

There are two main challenges when developing concurrent algorithms under persistent weak memory models such as Px86. **1)** The first challenge concerns thread *synchronization*. This difficulty is introduced by the fact that in the relaxed memory context, a read of a shared location may not return the location's last written value. **2)** The second challenge concerns *durability* (persistency). Without placing correctly explicit persist instructions in the algorithm and the careful design of a recovery mechanism, there is no guarantee on which values are visible in memory after a system crash. In this thesis, the term recovery mechanism/process refers to the set of actions taken following a crash to restore the computer system to a consistent state. While the specifics of the recovery mechanism can vary based on the nature of the crash, here we are primarily interested in describing the recovery mechanism in terms of its impact on the underlying memory model, without delving into any technical details.

Persistent STMs can help address both challenges by providing not only a way for thread synchronization, but also a high-level mechanism for managing *durability* and ensuring *failure atomicity*. In this manner, by utilizing persistent STMs, programmers can be liberated from the burden of comprehending and accurately placing low-level persist instructions within their programs in order to not compromise correctness. Despite the extensive literature on the implementation of persistent STMs [17, 83, 95, 106, 125, 141, 142,150], the verification aspect has received relatively little attention, with most relevant works primarily relying on informal correctness arguments e.g. [37,66,124]. Here, we are trying to narrow this research gap by formalizing persistent transactional memory and showcasing formal verification techniques for establishing the correctness of transactional memory implementations.

***Research question*:** To summarize, this thesis addresses the following research questions: Firstly, what does it mean for a persistent transactional memory algorithm to be correct, and how can this notion of correctness be formalized? Secondly, how can we effectively verify persistent transactional memory algorithms, and to what extent can existing verification techniques developed for volatile memory algorithms be applied in this context?

To this end, we have been focusing here on two goals.

I. *Defining correctness in the context of persistency*

One of the main challenges in verifying persistent transactional memory lies in defining correctness in a manner that encompasses the unique characteristics of persistent memory. In this work, we define correctness in terms of persistency at two different levels.

- We begin by defining correctness at the level of transactional memory. Our definition merges concepts from both the research areas of transactional memory correctness and concurrent object correctness in the context of persistency.

- We then define correctness at the language level. Our point of departure for this task is the theory of Owicki–Gries [114], which is widely used for reasoning about safety properties. We preferred the Owicki–Gries method over a concurrent separation logic [26, 58, 89, 138, 139] or a modal logic [3, 16, 33, 108] approach for two reasons. The first reason concerns its simplicity, which reduces the modeling and mechanization effort. The second reason is that it has been shown to work well with weak memory models [39, 96, 146].

Throughout this thesis, we use the Owicki–Gries method to reason about the safety of our programs' invariants (annotation) in terms of local correctness and interference freedom. Our perspective on safety requirements regarding persistent memory programs has evolved over time. In our initial work, we use the Owicki–Gries method in its original form to reason about our transactional memory implementation within the persistent SC model. However, in our subsequent work, we go beyond mere local correctness and interference-free properties. We also require our programs' invariants to preserve an additional property (*crash invariant*), which describes the state of the persistent memory *up to the first crash* of the program. Later we further refine the notion of the *crash invariant* by defining it as a collection of properties that are maintained by *all* program transitions, including those of the crash and the recovery operation.

II. *Applying formal verification techniques for showing correctness in the context of persistency*

Our next challenge entails employing formal verification techniques to show that our transactional memory implementations align with the notions of correctness specified above, ensuring, in this way, strong guarantees.

In both of our STM implementations, we formulate program invariants as Hoare triples [74]. While it is relatively straightforward to form assertions that describe the persistent SC state, forming meaningful assertions over the $\text{Px86}_{\text{view}}$ state requires careful handling. To this end, we have developed a Hoare logic comprising assertions specifically designed for the equivalent to the Px86, $\text{Px86}_{\text{view}}$ model. These assertions enable us to express relationships between different system components, including persistent memory. Our logic is largely inspired by prior work on the RC11 model [39].

To show that our STM implementations under the SC and the $\text{Px86}_{\text{view}}$ model are correct, we employ a methodology that has been used successfully to verify multiple concurrent programs [35, 44, 45, 47, 49, 50, 71]. The first step of the methodology involves modeling both the permissible behaviors (specification) according to our persistent STM correctness condition and the provided STM implementation as transition systems. The second step concerns demonstrating that the given implementation model refines the specification model via *simulation* proof techniques. The strength of this approach lies in its ability to enable hierarchical reasoning. Once we establish that our specification model implies our correctness condition, we reuse it to demonstrate correctness for both of our STM implementations.

1.4 Structure of the Thesis

Chapter 2 concerns background work. It first provides a high-level description of the persistent memory architecture. It then proceeds with presenting a set of definitions for formalizing Software Transactional Memory and an overview of the background work concerning correctness conditions for Software Transactional Memory and concurrent objects in the context of persistency.

In Chapter 3, we demonstrate a refinement proof that establishes the correctness of a persistent STM implementation under persistent SC. We begin by presenting a correctness condition, *durable opacity*, which adapts opacity [67] to the persistency setting. Afterward, we provide operational semantics for the persistent sequential consistency model. We then develop a persistent STM implementation, $\mathrm{dTML}_{\mathsf{SC}}$, which constitutes an adaptation of the *transactional mutex lock* (TML) algorithm [38] to handle system crashes. Subsequently, we introduce an operational characterization of durable opacity, dTMS2, which is based on the TMS2 specification that has previously been demonstrated to imply opacity [50]. Finally, we present a proof technique to show that our implementation is durably opaque. Our mechanized proof involves encoding dTMS2 and $\mathrm{dTML}_{\mathsf{SC}}$ as IO-automata within the Isabelle/HOL proof assistant. We then establish the existence of a forward simulation, which has been demonstrated to ensure trace refinement of IO-automata according to Lynch *et al.* [103]. This, in turn, guarantees the durable opacity of $\mathrm{dTML}_{\mathsf{SC}}$.

Chapter 4 presents a program logic for reasoning about x86 code that uses low-level operations such as memory accesses and fences, as well as persistency primitives such as flushes. Our logic benefits from a simple underlying operational semantics for the Px86 model based on views [32], and is mechanized in the Isabelle/HOL. We first present the equivalent to Px86, $\mathrm{Px86}_{\mathrm{view}}$ model, suggested in [32] and the corresponding operational semantics. Afterward, we detail our Owicki–Gries logic, Pierogi. We develop two versions of Pierogi, the first version ($\mathrm{Pierogi}_{\mathrm{simp}}$) supports reasoning up to the first crash of a program while the second version ($\mathrm{Pierogi}_{\mathrm{full}}$) supports reasoning beyond crash events. Next, we present the Pierogi proof rules and prove that they are sound.

In Chapter 5 we utilize the $\mathrm{Pierogi}_{\mathrm{simp}}$ logic to verify several litmus tests. In this way, we demonstrate how $\mathrm{Pierogi}_{\mathrm{simp}}$ can be used to reason about a range of challenging single- and multi-threaded persistent programs.

In Chapter 6, we demonstrate a refinement proof that establishes the correctness of a persistent STM implementation under the Px86 model. Firstly, we develop a persistent STM implementation, $\mathrm{dTML}_{\mathrm{Px86}}$, which is once again an adaptation of TML but with additional synchronization mechanisms to cope with Px86. Subsequently, we present a proof for showing that $\mathrm{dTML}_{\mathrm{Px86}}$ is durably opaque. Our correctness proof is operational and comprises two distinct types of proofs: (1) proofs of invariants of $\mathrm{dTML}_{\mathrm{Px86}}$ and (2) a proof of refinement against an operational specification that guarantees durable opacity. For (1), we build on the revised version of Pierogi, and for (2) we use the simulation-based technique presented in Chapter 3.

Chapter 7 includes a discussion of the preceding chapters and suggestions for future work.

1.5 Acknowledgements

Chapter 3 is based on work done in collaboration with Simon Doherty, Brijesh Dongol, John Derick, Gerhard Schellhorn and Heike Wehrheim [23]. In this project, I actively participated in discussions concerning the definition of durable opacity and the design of dTML$_{\mathsf{SC}}$. Additionally, I made contributions during the initial phases of mechanization, which involved encoding dTML$_{\mathsf{SC}}$ and dTMS2 in Isabelle/HOL and proving elementary lemmas. The presentation of this chapter revises the corresponding paper [23] to make it consistent with the remainder of the thesis. This work won the best paper award at FORTE 2022.

Chapter 4 and Chapter 5 are based on work done in collaboration with Brijesh Dongol, Ori Lahav, Azalea Raad, and John Wickerson [24] as well as work done with Brijesh Dongol (paper in progress). In [24], I actively participated in discussions regarding the formation of PIEROGI$_{\mathrm{simp}}$. Furthermore, I proposed several view-based assertions for facilitating the verification process of the programs discussed in Chapter 5. Finally, I was responsible for the mechanization of PIEROGI$_{\mathrm{simp}}$ in Isabelle/HOL, which involved encoding Px86$_{\mathrm{view}}$, establishing the proof rules of PIEROGI$_{\mathrm{simp}}$, and verifying the examples presented in Chapter 5. The presentation of these chapters revise the corresponding paper [24] to make it consistent with the remainder of the thesis. [24] won a Distinguished Artifact Award at ESOP 2022.

In a separate realm of work done with Brijesh Dongol, which is presented partially in Chapter 4, I actively participated in discussions concerning the extensions of the Px86$_{\mathrm{view}}$ model and PIEROGI$_{\mathrm{simp}}$ logic (PIEROGI$_{\mathrm{full}}$) to cover reasoning about programs beyond the occurrence crash events. Furthermore, I mechanized PIEROGI$_{\mathrm{full}}$ in Isabelle/HOL.

Chapter 6 is based on work done in collaboration with Brijesh Dongol. In this project, I was responsible for developing the dTML$_{\mathrm{Px86}}$ implementation and deriving the simulation relation along with the dTML$_{\mathsf{SC}}$ program annotation and crash invariant. Finally, I mechanized the simulation proof in Isabelle/HOL. The presentation of this chapter revises a paper in progress, to make it consistent with the remainder of the thesis.

Chapter 2

Preliminaries

This chapter presents preliminaries regarding the persistent memory architecture and software transactional memory.

2.1 Overview of Hardware

In this section, we begin by presenting a high-level overview of the hardware features of Intel's x86 persistency model. More specifically in §2.1.1, we provide a concise description of the modes in which persistent memory can be utilized. Following that, we discuss the concept of power-fail protected domains (§2.1.2) and analyze the different methods through which a write can be persisted (§2.1.3). Finally, in §2.2, we discuss a selection of persistency models.

2.1.1 Persistent Memory Modes

Persistent memory can be utilized in two memory modes, each with its unique architectural characteristics and operating system handling. Both modes comply with the SNIA NVM Programming Model, which is a guideline for how the operating system interacts with persistent memory. These modes offer a number of paths from the user space to the NVDIMMs with each path implying distinct persistent memory programming interface semantics. A brief summary of the modes can be found below.

2.1.1.1 Memory Mode

Under memory mode, persistent memory (e.g., Optane DC Persistent Memory) appears to the operating system as traditional volatile memory, allowing it to be used as the main memory for data that needs to be persisted. DRAM in this case is used as a direct-mapped write-back cache for persistent memory, meaning that it is not directly accessible from the user interface and cannot be explicitly controlled by the operating system. In this mode, writes are automatically buffered in DRAM which is essential for preventing performance degradation caused by a limited write bandwidth to NVDIMMs. Data placement on NVDIMMs is managed by the memory controller, which aims to

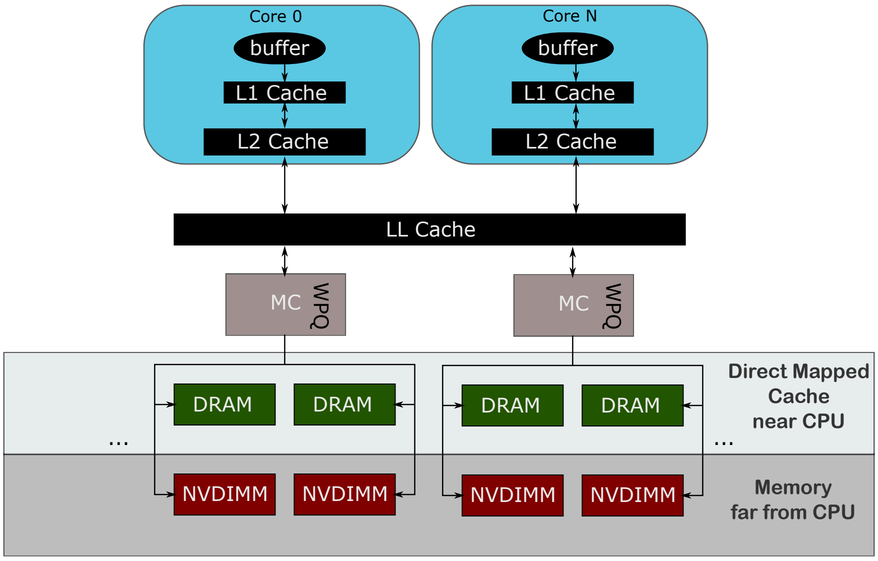


Figure 2.1: A high-level view of the hardware configuration of a system operating in Memory mode.

reduce the write latency gap between DRAM and NVDIMMs. One disadvantage of this design is that DRAM can only cache access to the NVDIMMs connected to the same memory controller [82], leading to non-uniform memory access (NUMA) effects [63]. Furthermore, the operating system needs to handle the data durability, as data in this mode do not automatically persist.

A simplified depiction of a system in Memory mode can be found in Fig. 2.1. As illustrated, typically, a system that displays Intel-x86 architecture can have one or more cores. Its core is connected to a local store buffer, which holds data before they are written to the cache RAMs. The store buffer is organized as a FIFO queue. While in the store buffer, data are only visible to the core that is connected to it. When data leave the store buffer, they enter the CPU cache which typically has three or more distinct levels. Usually, the caches that are nearer to the CPU cores are faster, smaller, and local to their core, while those that are farther away are larger in capacity and unified across all cores. The way data propagates through the CPU cache hierarchy is determined by several factors that may differ among systems, such as cache block size, associativity, replacement policy, writing policy, etc. The final level of the CPU cache is connected to one or more memory controllers, each of which has one or more memory channels. These memory channels are, in turn, connected to one or more DIMMs, which form a direct-mapped write-back cache. Additionally, each memory channel is also connected to one or more NVDIMMs where data reside after being evicted from the DRAM cache.

In the case of a read, the processor first searches for the desired data in the store buffer, and if the data are not found, it proceeds to search the next memory layers (i.e. CPU cache memory levels, DRAM cache, persistent memory) until it succeeds. In the case of a write, the desired data are first written to the store buffer. They are then propagated through the CPU cache hierarchy, and finally, they reach the DRAM cache. After being evicted from the DRAM cache, they are written to persistent memory.

When there is a need for cost-effective memory expansion, using NVDIMMs (in Memory

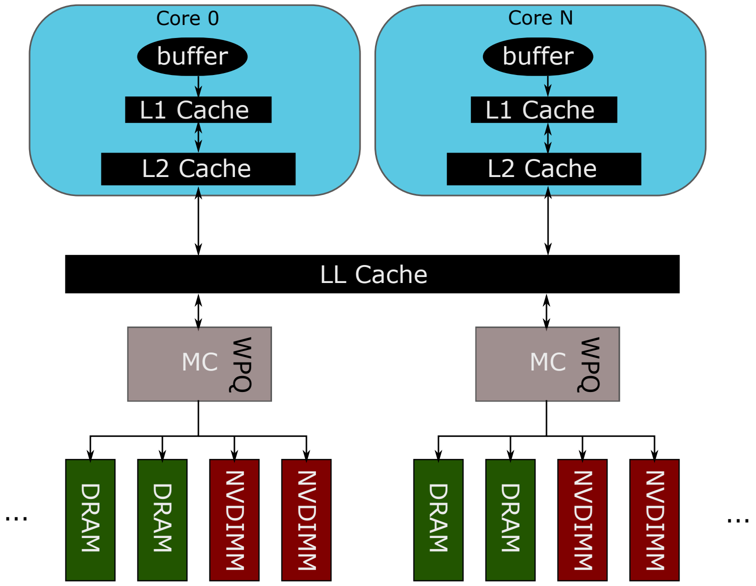


Figure 2.2: A high-level view of the hardware configuration of a system operating in App Direct mode.

mode) can be advantageous as they are cheaper than standard DIMMs and offer greater storage capacity. One additional benefit of this approach is its ease of integration with the system, requiring minimal effort due to its almost "Plug and Play" capability [59]. According to [149], a good indication that an application will perform well in memory mode is whether its working set (data that are used frequently during execution) fits inside the DRAM capacity. If so, the frequently used data can be accessed fast through the DRAM cache, while the remaining data reside in the persistent memory. Intel recommends a DRAM-to-persistent-memory ratio of 1:4.

2.1.1.2 App-Direct Mode

In App Direct mode, the Optane DC PMMs are exposed to the operating system as persistent block devices. Contrary to the Memory mode, there is no DRAM cache. By allowing the system to independently manage the DRAM and DC PMM resources, App Direct mode enables operations that require high speed to use DRAM, while less performance-demanding operations can utilize the DC PMMs. When a file system that supports *PMem* direct access (DAX) is available, file read/write operations can be translated into cache-line load/store instructions (64 bytes), enabling much more fine-grained data access than block-based access (usually 4k bytes), which is commonly used for storage [143, 148]. Moreover, the DAX feature enables accessing persistent memory directly from the user space.

Fig. 2.1 shows a high-level view of a system in App Direct mode. As illustrated, DIMMs (DRAM) and NVIDMMs (DC PMMs) are now at the same level and can both be accessed at the system's discretion. When a cache line is evicted from the CPU cache, its

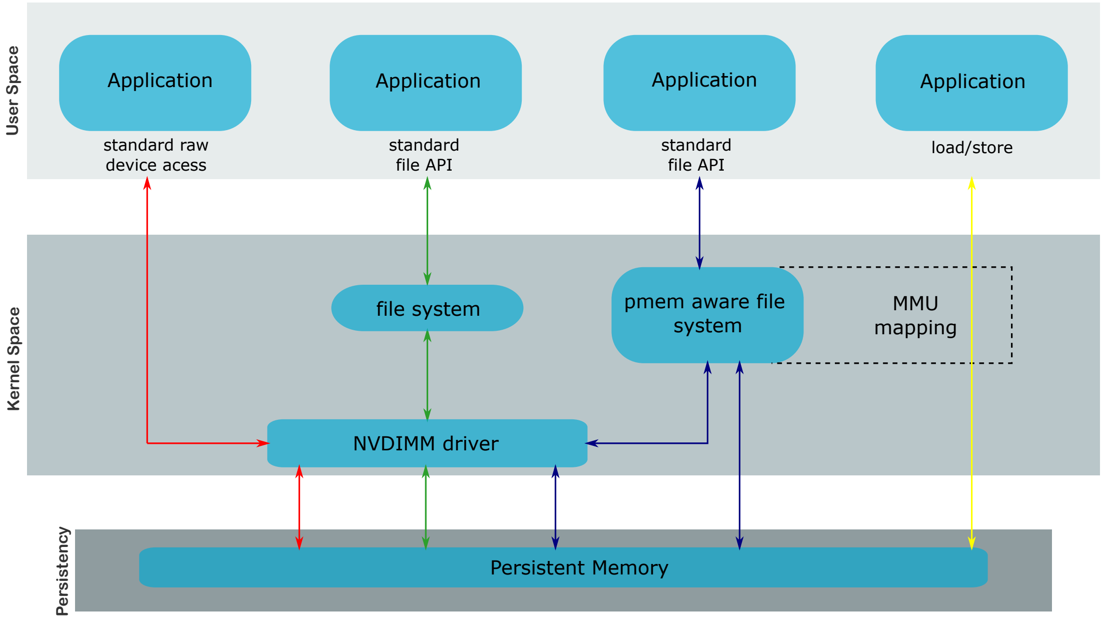


Figure 2.3: The software path in App Direct mode.

stored data enter the memory controller. The memory controller, among other features, maintains a write pending queue (WPQ), which temporarily holds the cache line stores. Stores are later pulled from the write pending queue and are sent to an NVDIMM in cache line (64-byte) granularity. Each NVDIMM has its own on-DIMM controller, which controls access to the Optane media by implementing an internal address translation. Upon the completion of the translation, stores access the Optane physical media in 256-byte (Optane block) granularity [147]. The disparity in access granularity (64-byte for the WPQ entry against 256-byte for the Optane block) causes write amplification where small stores become read-modify-write operations. The above, in turn, results in higher write latency compared to read latency. To mitigate this issue, the Optane controller maintains an on-DIMM write buffer that merges small adjacent writes [148].

Fig. 2.3 depicts a simplified version of the software stack as demonstrated in [129]. As illustrated, there are four different ways that persistent memory can be reached. The first way of using persistent memory was as a block storage device (red and green paths). To achieve this, the operating system was extended to detect the existence of persistent memory and support a persistent memory driver (NVDIMM Driver) which functions exactly as a storage device driver, with the addition that it can configure and monitor the state of persistent memory. As with traditional storage, applications can access persistent memory either directly or via the file system. The file system, in this case, continues to support standard file APIs, and the process of updating persistent memory through the file system remains the same as with block storage devices.

In brief, the most efficient way for an application to read from or write to a file is by memory mapping it. This is typically done with an `mmap()` system call in Linux or `MapViewOfFile()` in Windows. The `mmap()` / `MapViewOfFile()` call maps the file onto the virtual memory of the application's address space, enabling users to access the file directly in the same way as data on the memory. The virtual memory and main memory

(DRAM) are partitioned into virtual and physical pages of the same size, while the storage device is partitioned into blocks. Both load and store instructions operate on virtual addresses, which are translated to physical addresses on the fly by the Memory Management Unit (MMU).

The operating system is responsible for managing I/O operations by maintaining the page cache. In the case of a read, if the virtual address corresponds to a location in the program or to a data space that is mapped to a page in the main memory, the contents of the corresponding main memory location are immediately accessed. However, if the location is mapped to storage space, an exception is generated (page fault), and the operating system takes control to retrieve the requested word from the storage, which takes a significantly longer time. After fetching the corresponding storage block, it copies it to a page in the main memory (page cache). In this way, future references to the same address can be resolved in less time, as the requested data can be retrieved directly from the page cache. If all physical pages of the page cache are already occupied, the operating system is in charge of deciding which one to replace (LRU, FIFO, etc.). The low-level application programmer may not be aware of the limitations imposed by the limited physical memory available and how they are managed by the operating system.

In the case of a write, the operating system stores the new content in the corresponding page of the page cache. At some point, when the page cache is full, the page is written back into the storage device (lazy update). Only then is the new written content considered to be persistent. In order to ensure that data has been persisted before a system failure, an application can issue a system call (e.g. `fsync()/msync()` in Linux or `FlushFileBuffer()/FlushFileBuffers()` in Windows) that guarantees that the content of the provided file has been written back to the block storage.

This approach has two significant drawbacks: 1) updating the storage in page granularity makes both page faults and writing back to storage expensive, and 2) the main memory's available space is constantly reduced as it retains a copy of the files in the page cache. To avoid these drawbacks, the operating system was extended to support pmem-aware file systems. A pmem-aware file system has two ways of updating the persistent memory (blue paths). The first way is by using the NVDIMM driver and thus following the process described previously. The second way is by using the direct access (DAX) file system feature, which allows it to access the persistent memory directly. When DAX is supported, `mmap()` / `MapViewOfFile()` can map persistent memory directly into the user's address space. System calls such as `fsync()` work as expected, but the kernel handles persistent memory differently. Instead of writing back pages to storage, it utilizes newly supported explicit persist instructions that can update persistent memory at a cache line granularity. As before, when a file is mapped, its data are not necessarily persistent. On typical storage, a file becomes persistent when the relevant dirty pages are flushed out of the page cache to the block storage. In this case, in which direct access is possible, the newly written data, instead of being in the page cache, are located in the CPU cache. The new explicit persist instructions, instead of flushing pages from the page cache, flush CPU cache lines from the CPU cache to persistent memory. Bypassing the page cache boosts the overall performance significantly by taking advantage of the dual nature of persistent memory, which is accessed nearly as fast as DRAM memory but functions like storage.

The last extension of the operating system provides the option to the user to bypass almost completely the kernel code (file system) and handle persistency entirely from the user space (yellow path). Essentially, after exposing persistent memory to the application as a memory-mapped file (the initial mappings are still managed by MMU), a programmer can use the explicit persist instructions to flush cache lines directly into persistent memory. This alternative use is possibly a tradeoff between programming complexity and flexible, more efficient persistent memory handling.

Despite providing the shortest path to persistency, this option places the burden of determining which metadata to persist, in which way, and how frequently to do so in order to maintain consistency of data structures in the event of a crash (*failure atomicity*) on the programmer. Such a task is not trivial since algorithms that do not adhere to *failure atomicity* are indistinguishable from those that do during normal execution. This is due to the fact that in either scenario, the critical data are eventually made persistent through the applied cache coherence protocol. Therefore, without rigorous testing, failure atomicity is only detectable after a system crash, such as a power outage.

In summary, a programmer who accesses persistent memory from user space has to deal with the following challenges:

Out-of-order persisted stores: Dirty cache lines that are flushed by cache replacement mechanisms, do not ensure persistent memory consistency. Essentially, stores may reach the persistent memory in a different order than the order in which they were issued by the CPU. Persisting stores in arbitrary order leads to inconsistent persisted data. Store reorderings of similar nature have been studied in detail for preserving memory consistency under weak memory models (TSO, C11 etc). However, those reorderings concern the different to the issued order in which stores are made visible to other threads (exit the per-core store buffers), and not the order in which cache lines are evicted. Proposed *persistency models* [117, 121, 123], aim to describe the order in which stores persist, under different hardware assumptions, providing in this way persistent memory order guarantees to the programmer. Platform vendors such as Intel, ARM, and AMD support available from the user space explicit persist instructions, allowing in this way the users to afford greater control over the persistency order.

Non-atomic persistent updates: Even though stores are reaching persistent memory on a cache line granularity, on Intel hardware, the atomic store is 8 bytes. As a result, the size of stores that reach the persistent memory in an atomic manner is less than or equal to 8 bytes. In the case that a crash event interrupts a store with a size less than or equal to 8 bytes, persistent memory either contains the previously written or the newly written 8 bytes, but not a mix of both. Stores over 8 bytes, without special handling, might be torn after a system crash. However, most data structures utilize larger than 8-byte datatypes and thus require larger atomic updates. To prevent the possibility of an incomplete update caused by a system crash, the software cannot rely on a single instruction. Instead, the update must be made transactional by building on the 8-byte power-fail-atomic store in combination with the explicit persist and possibly persist barrier instructions provided by the hardware.

Asynchronous explicit persist instructions: The majority of the provided explicit persist instructions are asynchronous (i.e. they are not blocking the execution of following instructions until they are complete). Since the effect of those instructions might take place later than the time they were issued, they cannot provide any guarantee that the data of the address they flush reaches persistent memory unless they are used in conjunction with persist barrier instructions (in the case of Intel `sfence`, `mfence`, `CAS`). A persist barrier is a mechanism for enforcing ordering on the stores that reach the persistent memory by ensuring that the stores preceding it persist before those that follow it. Each explicit persist instruction has its own performance characteristics and should be preferred in different scenarios. For example in Intel-x86 architecture, the `clwb` instruction performs better than `clflushopt` in cases where future accesses to the flushed data are expected. As another example, according to [12], `clwb` performs better in redo log implementations while `clflushopt` performs better in undo log implementations. The programmer needs to decide carefully which instruction to use and place it in the program such that durability is not risked while performance is maximized.

The first requirement when handling persistent memory directly from the user space is the detection of the responsibilities of its entity towards flushing critical data to persistent memory media and recovering them after a power loss/crash. The concept of a *power-fail protected domain* (persistence domain) helps in the assignment of these responsibilities to the hardware platform or the application. A power-fail protected domain is a memory area whose stored data are assumed to be flushed by the hardware platform. As soon as the data reach a power-fail-protected domain, even if they haven't reached the persistent media before a system crash, they are assumed to be recoverable. Persistence domains may vary from system to system. In general, larger power-fail protected domains suggest less responsibility for applications regarding flushing. We briefly analyze Intel's power-fail-protected domains and flush instructions in the next sections.

2.1.2 Power-Fail Protected Domains in App Direct Mode

Intel supports two DRAM refresh features: `ADR` (Asynchronous DRAM refresh) and `eADR` (enhanced Asynchronous DRAM refresh). Each of those suggests a different persistence domain. Fig. 2.4 provides an illustration of the persistence domain for the two DRAM refresh modes.

The persistence domain for `ADR` includes only the (volatile) controller's write pending queue (WPQ) and the NVDIMMs (represented by the deep grey square of Fig. 2.4). The persistence domain for `eADR` also includes the CPU cache (represented by the light grey square of Fig. 2.4). In both cases, once a store reaches the boundary of the specified persistence domain (either the memory controller for `ADR` or the CPU cache for `eADR`), it is guaranteed to persist even in the event of a system crash. This is because the `ADR\eADR` feature introduces to the system an amount of reserve power, usually in the form of an external battery, which is sufficient to flush the *in-flight* stores to persistent memory in the case of a power outage.

The persistence domain has a significant impact on the complexity of the programming model. As shown in Fig. 2.4 when `eADR` is supported, the point at which a store becomes visible to other threads (PoV) coincides with the point at which it is assumed to be persistent (PoP). This implies that the consistency model specifies both the volatile and persistent memory orders. As a result, the application does not need to handle explicit flushes and only needs to maintain memory consistency. On the other hand, when `ADR` is supported, the point of visibility for a store differs from the point of persistency, making the programming model harder, as in this case failure atomicity is decoupled from

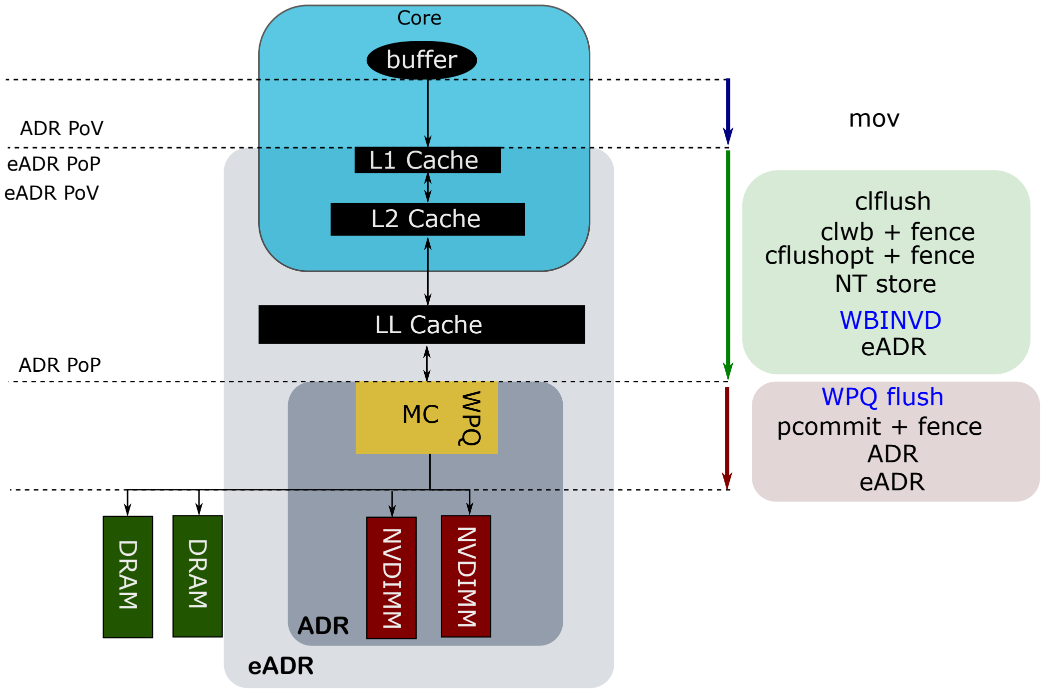


Figure 2.4: A high-level view of Intel's Power-Fail Protected Domains, and the flush instructions supported for each domain. "PoP" stands for "Point of Persistency," while "PoV" stands for "Point of Visibility". Kernel-only instructions are depicted in blue colour.

memory consistency. The main challenge with `eADR` lies in the large size of CPU caches, which require more energy (e.g., larger batteries) to be sustained. Larger batteries result in more expensive and less environmentally friendly designs. A possible alternative that provides the same durability guarantees as `eADR` is demonstrated in [6]. The proposed design, instead of requiring a battery-backed cache, only requires a smaller battery-backed buffer. The stores in this design are duplicated. The duplicate stores enter the buffer (upper boundary of the persistence domain) and the CPU cache simultaneously, making their points of persistency and visibility aligned. This thesis focuses on systems that support the `ADR` feature, as `ADR` is more widely supported compared to `eADR`.

2.1.3 Flushing in App Direct Mode

Fig. 2.4 illustrates the possible ways in which a store can pass through the levels of the memory hierarchy. The instructions analyzed here are specific to the Intel-x86 architecture.

Stores in Intel-x86 are implemented with the `mov` instruction (`mov` *dest src*). A store is typically first added to the core's FIFO store buffer. By default, when the store exits the store buffer, it enters the CPU cache. There are many ways in which the stores contained in a cache line can reach the memory controller.

1) ***Cache replacement mechanism:*** This method relies on the natural occurrence of a cache line eviction due to the cache coherence protocol. However, it does not provide any guarantee regarding whether or when a cache line will eventually make its way to the memory controller.

2) ***Explicit persist instructions:*** The explicit persist instructions (`clflush` *addr*, `clflushopt` *addr*, `clwb` *addr*), can be called from the user space, and their function is to flush from the CPU cache all the addresses of the cache line of the provided as argument address *addr*. As mentioned previously, the explicit persist instructions are executed in an asynchronous manner, permitting their effects to be reordered to a later point in the program execution. However, the behavior of `clflush` is identical to that of a synchronous instruction (with the exception of subsequent `clflushopt` instructions on different cache lines and `load` instructions to which it is not ordered) due to the greater number of ordering constraints that it imposes [122]. The `clflushopt` and `clwb` instructions present equivalent behavior and impose the same ordering constraints. Their main difference is that while `clflushopt` invalidates the cache line that persists (similar to `clflush`), the `clwb` instruction does not. The above makes `clwb` preferable in scenarios where reads exceed writes. In contrast to `clflush`, both `clflushopt` and `clwb` do not provide any durability guarantee without being combined with a persist barrier (in the case of Intel-x86, `sfence`, `mfence`, `CAS`). The store fence (`sfence`) instruction guarantees that at the end of its execution, the pending flushes of previously issued `clflushopt` and `clwb` instructions reach the persistence domain (as specified by `ADR`). The compare and swap (`CAS`) and memory fence (`mfence`) instructions are stronger than the (`sfence`) and thus provide the same durability guarantee compared to the store fence instruction.

3) ***NT stores:*** A non-temporal store is a type of memory access operation that does not require the use of the CPU cache, which is typically designed to optimize

mov x 1;
clflush x;
mov y 1;
clflush y;
pcommit;
sfence;
mov z 1;
(a)
	mov x 1;
	clflush x;
	mov y 1;
	clflush y;
	mov z 1;
	(b)
		mov x 1;
		clflushopt x;
		mov y 1;
		clflushopt y;
		sfence;
		mov z 1;
		(c)
			mov x 1;
			mov y 1;
			mov z 1;
			(d)

None	ADR	ADR	eADR

Figure 2.5: Four programs with equivalent behavior under different hardware assumptions (persistence domains).

access times based on the temporal locality. As a non-temporal store bypasses the CPU cache, it directly reaches the persistence domain (as specified by `ADR`). Non-temporal stores are often used in applications that involve large amounts of data that are accessed in a sequential or non-repetitive manner. Non-temporal stores can improve the overall performance by reducing cache pollution.

4) ***WBINVD:*** The WBINVD instruction flushes the entire CPU cache and invalidates every cache line. This instruction is only accessible to the kernel and is typically used by the operating system during a system shutdown to ensure that all stores to persistent memory have reached the persistence domain and any stale data is cleared from the cache before the system loses power.

5) ***eADR:*** As mentioned before, when `eADR` is supported, any store after exiting the store buffer is assumed to be persistent. Therefore, there is no need for explicit flushing of cache lines.

Typically, `ADR` is the minimum platform requirement for systems that support persistent memory; thus, in most cases, there is no need for explicit flushing of the memory controller's write pending queue (WPQ). However, a platform still has a way to flush the WPQ using the `wpq flush` kernel-only feature. Before `ADR` became standard, the persistence domain included only the NVDIMMS. The flushing of WPQ was performed by the `pcommit` instruction. The `pcommit` instruction is asynchronous, so its completion is guaranteed only after the execution of an `sfence`. Since the `ADR` feature became prevalent on Intel's platforms, the `pcommit` instruction has been deprecated [128].

Fig 2.5 demonstrates four programs that yield the same durability guarantee under different persistence domains. Specifically, if z is equal to 1 in persistent memory, then x also equals 1, and y equals 1 in persistent memory ($z = 1 \Rightarrow x = 1 \wedge y = 1$). Example 2.5a assumes a platform with neither `ADR` nor `eADR` supported. Firstly, the value 1 is written to address x. The `clflush` x instruction afterward flushes (evicts) the cache line of address x from the cache to the memory controller. Since `clflush` is ordered with respect to subsequent writes, its effect takes place before the write of 1 to address y. Afterward, the write of 1 to address y is executed, and the cache line that it belongs to is flushed (`clflush` y) to the memory controller. The `pcommit` instruction flushes the accepted to persistent memories stores (i.e. the stores that are in the write pending queue (WPQ) of the memory controller) to the NVDIMM. The effect of `pcommit` will surely take place by the completion of the `sfence` execution. Finally, 1 is written to address z. If `mov` z 1 has executed and persisted before a crash, then so must all the preceding instructions. Thus, if a crash occurs during the execution of this program and upon recovery $z = 1$ then $x = 1$ and $y = 1$.

The next two examples assume a platform with `ADR` supported. Here the programming model is simpler, as `ADR` relieves the application from flushing the controller's write pending queue. Due to this, there is no need for the `pcommit` instruction. The difference between examples 2.5b and 2.5c is that while the first uses the strongly ordered `clflush` , the second uses the less strongly ordered optimized flush instruction. To elaborate, in 2.5c the effect of `clflushopt` x might take place after the subsequent `mov` y 1 and `clflushopt` y if the cache line for address x is not the same as the cache line for address y. However, the effect of both optimized flushes takes place not later than the completion of the `sfence` instruction. As before, if the `mov` z 1 instruction has executed and persisted before a crash, then $x = 1$ and $y = 1$ in persistent memory upon recovery.

Assuming an `eADR` supported platform, example 2.5d follows a simpler programming model as explicit cache line flushing is not necessary. The self-ordering of the `mov` instruction ensures that the stores in 2.5d are executed and persisted in the order they are issued. When `eADR` is supported, the persistency order is determined by the volatile (visibility) order.

The scope of this thesis pertains to systems in App Direct mode (Fig. 2.2) that support `ADR`. We are mostly concerned with the verification of programs that handle persistency from the user space (yellow path of Fig. 2.3). The semantics and program logic given in later chapters concern only a subset of the flushing mechanisms demonstrated here. Specifically, we are concerned with programs in which flushing is either occurring through the cache replacement mechanism or with the use of explicit persist instructions combined, when necessary, with persist barriers.

Pelley *et al.* [117] proposed the use of persistency models to establish the persistency semantics of programs. These models dictate the acceptable behaviors of programs upon recovery by specifying the sequence in which writes should be persisted in memory. The next section provides a description of the most prevalent persistency models.

2.2 Background work on Persistency Models

This section starts by summarizing the classification of persistent memory models and describing certain models as proposed in [117] (2.2.1). The memory models discussed here are not used in the remainder of the thesis. However, we still think they are important for comparative analysis with Intel's persistency model (Px86 model), which we formalize and verify in later chapters. In §2.2.2, we introduce the Px86 persistency model and summarise some alternatives to ours, formalizations of Px86.

2.2.1 Persistency Models proposed in [117]

In [117], the problem of ordering persists is parallelized with the problem of ordering volatile memory accesses. The proposed persistent memory models leverage existing memory consistency models such as Total Store Order (TSO) and Relaxed Memory Order (RMO).

As discussed in [122], persistency models can be classified based on two criteria: (1) strict versus relaxed, and (2) buffered versus unbuffered. Under strict persistency, the order in which instruction effects become globally visible coincides with the order in which they persist. Relaxed persistency allows the above orders to differ. The second classification pertains to the timing of persists. In unbuffered persistency, persists occur synchronously, meaning that the effects of an instruction are immediately committed to persistent memory during its execution. On the other hand, buffered persistency enhances performance as it allows instructions to persist asynchronously without stalling the execution. In this approach, persists take place after their corresponding stores, so the execution may continue ahead of persists. As a result, in the event of a crash and subsequent recovery, it is possible that only a prefix of the persistent memory order has been successfully persisted. The Intel-x86 architecture follows a buffered persistency model.

The below presentation of Pelley's persistency models follows the formalization suggested in [93]. The term "persist" is used to describe the durable writing of a store to persistent memory, which we assume to occur atomically (with respect to failures) at an 8-byte granularity. The term "thread" refers to a core or a hardware thread. The formalization involves order relations between memory events, namely loads and stores, which are collectively referred to as memory accesses. The notation B_l^i denotes a memory access from a thread i to the location l. The volatile memory order (VMO) describes the ordering relation between memory accesses as imposed by the consistency model. The persistent memory order (PMO) includes the same events, though the order is determined by the constraints introduced by the persistency model. We denote as $B_{l1}^i \leq_{VM} B_{l2}^j$ the ordering relation in which the memory access B_{l1}^i happens before or at the same time as the memory access B_{l2}^j in VMO. Similarly, $B_{l1}^i \leq_{PM} B_{l2}^j$ denotes the ordering relation in which the memory access B_{l1}^i happens before or at the same time as the memory access B_{l2}^j in PMO.

Strict persistency. Under strict persistency, the order of the events in persistent memory is the same as the order of those in the volatile memory. Persistent models that follow strict persistency rely entirely on the volatile memory consistency model to set the order of the persistent events.

$$B_{l1}^i \leq_{PM} B_{l2}^j \Leftrightarrow B_{l1}^i \leq_{VM} B_{l2}^j \tag{2.1}$$

The volatile memory order can be either a conservative consistency model such as sequential consistency (SC) or a relaxed consistency model (RMO, TSO, etc.).

A program satisfies SC [98] if its execution result can be obtained by maintaining the instructions' order within each thread while interleaving instructions from different threads in any arbitrary but global manner across all the threads. On the contrary, relaxed consistency models, allow loads/stores inside a thread program to be reordered. Strict persistency simplifies the problem of persistent memory ordering, as reasoning about the volatile memory order tackles both problems.

Implementing strict persistency can lead to frequent execution stalls. This is because in the case of SC every time a store is executed, the CPU should wait for the counterpart NVDIMM write to complete. Flushing stores in persistent memory in the order they are issued, conflicts with two hardware optimizations. Firstly, stores in a cache line are coalesced and, when not explicitly flushed, reach the persistent memory on a cache line replacement. Inserting a `clflush` instruction after every store prevents coalescence, resulting in unnecessary write-backs. Secondly, processors rearrange the persistence of cache lines to enhance performance by leveraging temporal and spatial locality. This rearrangement takes place in both caches and memory controllers. Nevertheless, in strict pesistency, cache lines must be flushed in program order which eliminates any potential performance benefits from reordering writes in memory. Respectively, under relaxed memory consistency models every time a memory barrier is encountered, imposing an ordering constraint on the visibility of stores, the CPU should also stall until the stores prior to the memory barrier have reached the NVDIMMs. Strict persistency can be either implemented by software (putting persist barriers and explicit persist instructions in place) or hardware (e.g. `eADR`, [55, 84])

An optimization of strict persistency is ***buffered strict persistency*** in which the program execution can be ahead of the persistent state. As far as the visibility of the recovery process is concerned, it is allowed to view any past memory state, given that it is consistent. An instance of a hardware strict persistency implementation is DPO [94] which expands the cache-coherence protocol by integrating persist-buffers. The persistent buffers keep track of persistent requests and fences from their corresponding cores in order to identify persistent dependencies within a thread. They also monitor cache coherence traffic to identify persistent dependencies between threads.

Relaxed persistency. According to relaxed persistency, the order of the events in persistent memory may differ from the order of those in volatile memory. In this case, memory, and persistency barriers should be handled separately.

Memory barriers ensure that all the threads view the same ordering of memory operations, while persistency barriers specify the acceptable persists orders that are only visible from the recovery process. Relaxing persistency may complicate reasoning for the correctness of persists' ordering. However, it might improve performance due to fewer execution stalls. Below is a brief description of some relaxed persistency models. Both models outlined, respect *strict persist atomicity*, which builds on *store atomicity.* Store atomicity ensures that stores on the same memory address can be serialized. This property can be induced by cache coherence. Similarly, persist atomicity guarantees that persists in the same address can be serialized.

- **Epoch Persistency:** Under epoch persistency, volatile memory satisfies sequential consistency. Each thread can issue persist barriers. Persist barriers divide the thread's execution to persist epochs. Memory accesses that belong to different epochs in volatile memory order are ordered in persistent memory. In the following relation, PB_i denotes a persist barrier of the thread i.

$$B^i_{l1} \ \leq_{VM} PB_i \leq_{VM} \ B^i_{l2} \Leftrightarrow \ B^i_{l1} \ \leq_{PM} B^i_{l2} \tag{2.2}$$

  Persist barriers ensure that the persists of the epoch before the barrier never happen after the persists of the following epoch. Persists inside each epoch can happen concurrently and do not need to be serializable. However, this approach introduces complexity in determining the sequence of events, particularly when different threads access the same memory address. To maintain coherence persists on the same address must adhere to their execution order, thereby ensuring strict persist atomicity. The feasibility of epoch persistency has been explored in a number of works, such as [36, 85, 94]. These articles have investigated different implementations of persist barriers and have shown that they can provide performance improvements over alternative models.

A variation of epoch persistency is ***buffered epoch persistency*** according to which execution is allowed to continue beyond an epoch until encountering an epoch conflict. An epoch conflict arises when a location is updated in an epoch without its previous update in a preceding epoch being persisted. An epoch conflict triggers the explicit flush of all the locations that have not been persisted before the conflict.

- **Strand Persistency:** According to this model, program execution is divided into strands. Strands are logically independent segments of the execution of the same thread. Since there is no dependency between them, strands of the same thread behave as different threads. Strands are separated by strand barriers. In the following relation, SB_i denotes a strand barrier of thread i. Persists order within a strand is imposed by persist barriers. Strand persistency obliges that the persistent memory access order of any thread should respect Eq. (2.2) if it is not separated by a strand barrier. In the following relation, SB_i denotes a strand barrier of a thread i.

$$(B^i_{l1} \leq_{VM} PB_i \leq_{VM} B^i_{l2}) \wedge (\nexists SB_i : B^i_{l1} \leq_{VM} SB_i \leq_{VM} B^i_{l2}) \Rightarrow B^i_{l1} \leq_{PM} B^i_{l2} \tag{2.3}$$

Following the representation style of [85], Fig. 2.6 shows three execution timelines under three different persistency models. The first execution represents a sequence of writes under strict persistency. A depicted, when a write becomes visible to other threads (visibility line), it should immediately persist. In this scenario, the system effectively uses the CPU cache in a write-through mode to immediately flush all writes issued by a thread as soon as they become visible to other threads. The second timeline concerns a sequence of writes under epoch persistency. In this case, writes within an epoch can persist in a different order from which they were issued. The program execution stalls when it reaches an epoch boundary (persist barrier) until all the writes within the epoch are persisted. Modified cache lines that persisted naturally via cache replacement, are not explicitly flushed. This model leverages cache coalescence. As an example, there is no requirement for persisting both stores on address "a" at the end of epoch 1 since they both belong to the same cache line. As a result, the cache line of address "a" is only persisted once. Finally, the third timeline concerns a sequence of writes under buffered epoch persistency. In this case, the execution can continue beyond epochs if there is no epoch conflict. An epoch conflict arises when an epoch attempts to execute a store on an address without the previous store on the same address being persisted. In such scenario, the epoch must explicitly persist all the stores that precede the conflicting store before continuing its execution. As seen, the store on address "g" by epoch 3, is conflicting with the previous store in the same address by epoch 2. Epoch 3 flushes all the previous, not yet persisted, stores before completing the conflicting write.

2.2.2 The Px86 Model

The Px86 model describes the semantics of programs running in systems that display Intel-x86 hardware including an NVM technology such as Optane DC. In particular, this

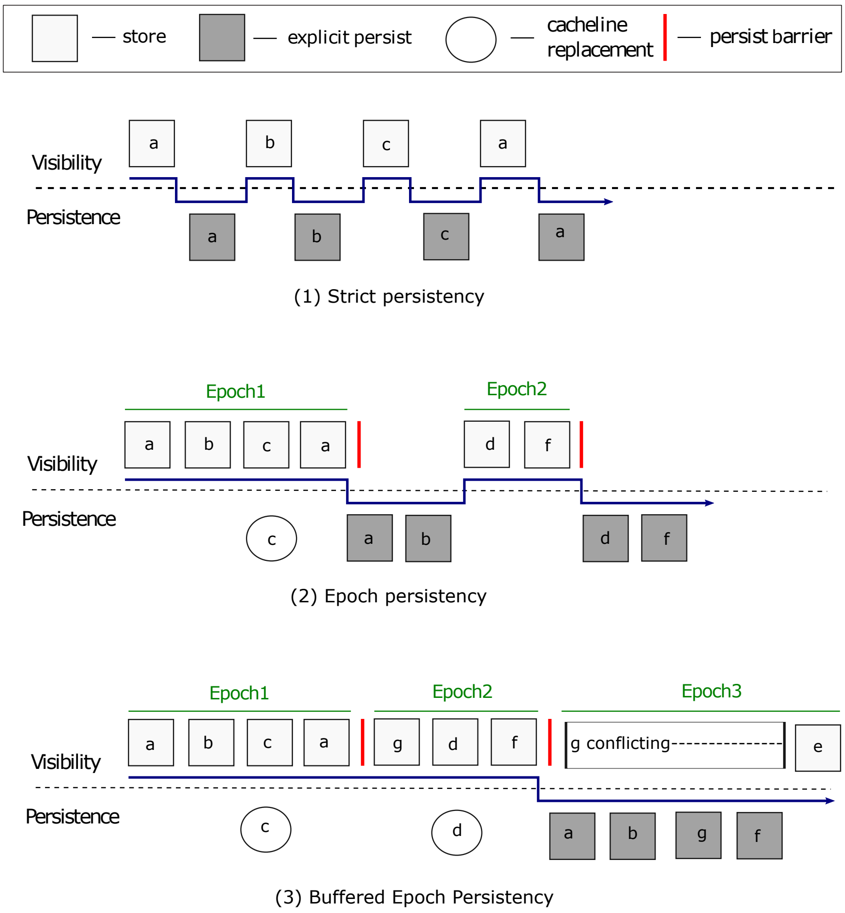


Figure 2.6: Sequences of stores to different cache lines under different persistency models as presented in [85].

model describes the behaviour of programs that run on App Direct mode platforms that support `ADR`.

X86 persistency follows a buffered relaxed persistency model which can be seen as an extension of the total store order consistency model. As mentioned in §2.1.3, one of the main challenges that the explicit persist instructions introduce is that there effect is asynchronous. This way persists occur after their corresponding stores and as prescribed by the persistency semantics while allowing the execution to proceed ahead of persists. As such, after recovering from a crash, only a *prefix* of the persistent memory order may have persisted.

Under relaxed persistency (§2.2.1), the volatile (VMO) and persistent memory (PMO) orders may disagree: the order in which the writes are made visible to other threads may differ from the order in which they are persisted. To distinguish between the two, in this discussion, memory *stores* are differentiated from memory *persists*: the former denotes the process of making a write visible to other threads, whilst the latter denotes the process of committing writes to persistent memory (durably).

For the rest of this section, Intel's `mov` instruction is denoted as **store**, the `ld` instruction is denoted as **load**, the `clflush` instruction is denoted as **flush** and the `clflushopt` instruction is denoted as $\textbf{flush}_{\text{opt}}$. Furthermore, the **mfence** and **sfence** instructions refer to Intel's memory and store fence instructions, respectively, and finally, **CAS**, as before, refers to the RMW compare-and-swap instruction. In order to illustrate the consistency and persistency model of Px86, we provide a recap of examples presented in Chapter 1 and introduce additional examples.

Px86 Consistency.

The consistency semantics of Px86 is that of the well-known TSO (total store ordering) [131] model, where later (in program order) reads can be reordered before earlier writes on different locations. This is illustrated in the *store buffering* (SB) example below (left):

$$\begin{array}{l||l} \textbf{store}\ x\ 1; & \textbf{store}\ y\ 1; \\ a := \textbf{load}\, y & b := \textbf{load}\, x; \end{array} \qquad\qquad \begin{array}{l||l} \textbf{store}\ x\ 42; & a := \textbf{load}\, y; \\ \textbf{store}\ y\ 7 & b := \textbf{load}\, x; \end{array}$$

$$a = 0 \land b = 0 : \checkmark \quad (\text{SB}) \qquad\qquad a = 7 \land b = 0 : \times \quad (\text{MP})$$

Specifically, assuming $x = y = 0$ initially, since $a := \textbf{load}\, y$ (resp. $b := \textbf{load}\, x$) can be reordered before **store** x 1 (resp. **store** y 1), it is possible to observe the weak behaviour $a = 0 \land b = 0$. A well-known way of modeling such reorderings in TSO is through *store buffers*: when a thread t executes a write **store** x v, its effects are not immediately made visible to other threads; rather they are delayed in a thread-local (store) buffer only visible to t and propagated to the memory at a later time, whereby they become visible to other threads. For instance, when **store** x 1 and **store** y 1 are delayed in the respective thread buffers (and thus not visible to one another), then $a := \textbf{load}\, y$ and $b := \textbf{load}\, x$ may both read 0.

After SC (sequential consistency) [98], TSO is one of the strongest consistency models and supports synchronization patterns such as *message passing*, as shown in MP above (right), where $a = 7 \land b = 0$ cannot be observed. Specifically, (assuming $x = y = 0$ initially) if the right thread reads 7 from y (written by the left thread), then the left thread passes a message to the right. Under TSO, message passing ensures that the instruction writing the message and all those ordered before it (e.g. **store** x 42; **store** y 7) are executed (ordered) before the instruction reading it (e.g. $a := \textbf{load}\, y$). As such, since $b := \textbf{load}\, x$ is executed after $a := \textbf{load}\, y$, if $a = 7$ (i.e. **store** x 42 is executed before $a := \textbf{load}\, y$), then $b = 42$.

Px86 Persistency.

The relaxed and buffered persistency of Px86 is shown in Fig. 2.7a. If a crash occurs during (or after) the execution of Fig. 2.7a, at crash time either write may have persisted and thus $x, y \in \{0, 1\}$ upon recovery. Note that the two writes cannot be reordered under Intel-x86 (TSO) consistency and thus at no point during the normal (non-crashing) execution of Fig. 2.7a is $x{=}0, y{=}1$ observable. Nevertheless, in case of a crash it is possible to observe $x{=}0, y{=}1$ after recovery. That is, due to the relaxed persistency of Px86, the store order (x before y) is separate from the persist order (y before x). More concretely, under Px86 the writes may persist 1) in any order, when they are on distinct locations; or 2) in the volatile memory order, when they are on the same location.[1]

[1] Given a *cache line* (a set of locations), writes on distinct cache lines may persist in any order, while writes on the same cache line persist in the volatile memory order. As in Chapter 3, we assume that each cache line contains a single location, thus forgoing the need for cache lines. However, it is straightforward

store x 1; **store** y 1	**store** x 1; **flush** x; **store** y 1	**store** x 1; **flush**$_{\text{opt}}$ x; **store** y 1	**store** x 1; **flush**$_{\text{opt}}$ x; **sfence**; **store** y 1	**store** x 1; **flush** x; **store** y 1	$a :=$ **load** y; **if** $(a{=}1)$ **store** z 1
(a)	(b)	(c)	(d)	(e)	
$\lightning : x, y \in \{0,1\}$	$\lightning : y{=}1 \Rightarrow x{=}1$	$\lightning : x, y \in \{0,1\}$	$\lightning : y{=}1 \Rightarrow x{=}1$	$\lightning :\ z{=}1 \Rightarrow x{=}1$	

Figure 2.7: Example Px86 programs and possible values after recovery from a crash ($\lightning$). In all examples x, y, z are distinct locations in persistent memory such that $x{=}y{=}z{=}0$ initially, and a is a (thread-local) register.

Instructions such as **flush** x and **flush**$_{\text{opt}}$ x can be used for controlling when pending writes are persisted.[2] This is illustrated in Fig. 2.7b: executing **flush** x persists the earlier write on x (i.e. **store** x 1) to memory. As such, if the execution of Fig. 2.7b crashes and upon recovery $y{=}1$, then $x{=}1$. That is, if **store** y 1 has executed and persisted before the crash, then so must the earlier **store** x 1; **flush** x. Note that $y{=}1 \Rightarrow x{=}1$ describes a *crash invariant*, in that it holds upon crash recovery *regardless* of when (i.e. at which program point) the crash may have occurred. Observe that this crash invariant is guaranteed thanks to the ordering constraints on **flush** instructions. Specifically, **flush** instructions are ordered with respect to all writes; as such, **flush** x in Fig. 2.7b cannot be reordered with respect to either write, and thus upon recovery $y{=}1 \Rightarrow x{=}1$.

However, instruction reordering means that persist instructions may not execute at the intended program point and thus not guarantee the intended persist ordering. Specifically, **flush**$_{\text{opt}}$ x is only ordered with respect to earlier writes on x, and may be reordered with respect to later writes, as well as earlier writes on different locations. Example Fig. 2.7c presents the same pattern as the example Fig. 2.5c and highlights the ordering constraints of the **flush**$_{\text{opt}}$ instruction. Specifically, **flush**$_{\text{opt}}$ x is not ordered with respect to **store** y 1 and may be reordered after it. Therefore, if a crash occurs after **store** y ,1 has executed and persisted, but before **flush**$_{\text{opt}}$ x has executed, then it is possible to observe $y{=}1, x{=}0$ on recovery. That is, there is no guarantee that **store** x 1 persists before **store** y 1, *despite* the intervening **flush**$_{\text{opt}}$ x.

In order to prevent such reorderings and to strengthen the ordering constraints between **flush**$_{\text{opt}}$ and later instructions, one can use either *fence* instructions, namely **sfence** (store fence) and **mfence** (memory fence), or atomic *read-modify-write* (RMW) instructions such as compare-and-swap (**CAS**) and fetch-and-add (**FAA**). More concretely, **sfence**, **mfence** and RMW instructions are ordered with respect to all (both earlier and later) **flush**$_{\text{opt}}$, **flush** and write instructions, and can be used to prevent reorderings such as that in Fig. 2.7c. This is illustrated in Fig. 2.7d. Unlike in Fig. 2.7c, the intervening **sfence** ensures that **flush**$_{\text{opt}}$ in Fig. 2.7d is ordered with respect to **store** y 1 and cannot be reordered after it, ensuring that **store** x 1 persists before **store** y 1 (i.e. $y{=}1 \Rightarrow x{=}1$ upon recovery), as in Fig. 2.7b. Note that replacing **sfence** in Fig. 2.7d with **mfence** or an RMW yields the same result. However, upon executing a barrier instruction (i.e. **mfence**, **sfence** or an RMW), execution is blocked until the effect of earlier **flush**$_{\text{opt}}$ instructions take place; that is, executing such barrier instructions ensures that earlier **flush**$_{\text{opt}}$ behave *synchronously* (like **flush**). Alternatively, one can think of **flush**$_{\text{opt}}$ x executing *asynchronously*, in that its effect (persisting x) does not take place immediately

to lift this assumption.

[2] Executing **flush** x or **flush**$_{\text{opt}}$ x persists the pending writes on *all locations in the cache line of* x. However, as discussed, we assume cache lines contain single locations.

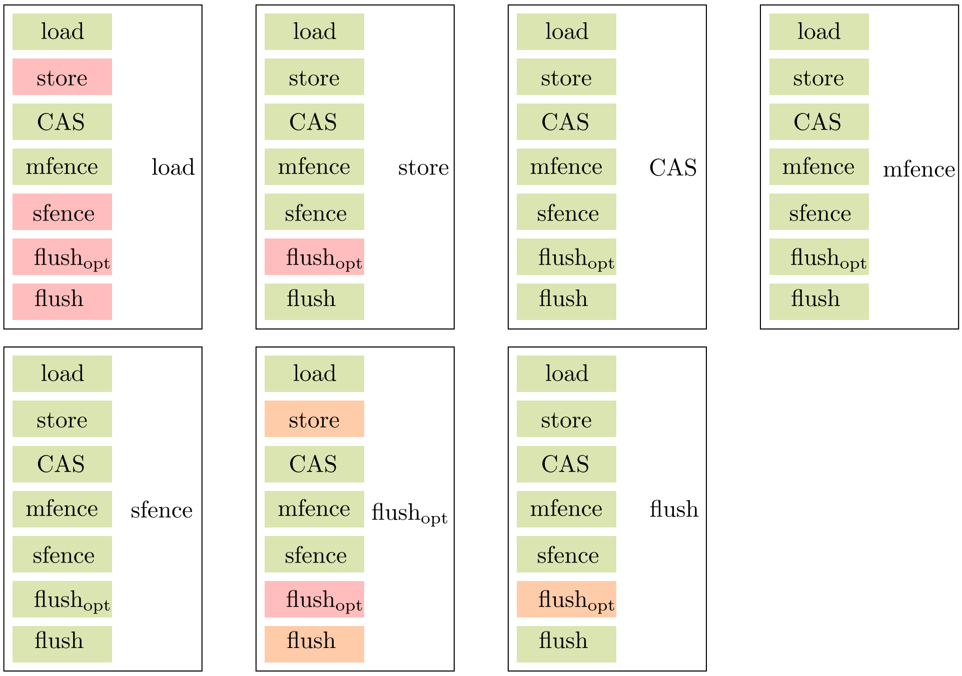


Figure 2.8: This diagram illustrates the ordering constraints of a selection of Px86 instructions as demonstrated in [122]. Each rectangle depicts how the instruction placed on the right side of the rectangle is ordered with respect to the instructions placed on the left side of the rectangle. The ordering here does not refer to the order in which the instructions appear to the program but the order in which the effect of their execution takes place. We assume that each instruction on the left side precedes in program order the instruction on the right side. The green-colored instructions cannot be reordered after the execution of the right instruction, while the red-colored instructions can be reordered. For example, consider the program **store** x 1; **load**y. Address x might obtain the value 1 after address y is read. The orange-colored instructions are only ordered with respect to addresses of the same cache line. For example, consider the program **flush**$_{\text{opt}}$ x; **flush** y. If x and y belong to the same cache line, the address x will always be persisted before the address y.

upon execution, but rather at a later time.

The example in Fig. 2.7e illustrates how message passing can impose persist orderings on the writes of *different* threads. (Note that the program in the left thread of Fig. 2.7e is that of Fig. 2.7b.) As in MP, if $a = 1$, then **store** x 1; **flush** x is executed before $a :=$ **load** y (thanks to message passing). Consequently, since **store** z 1 is executed after $a :=$ **load** y when $a = 1$, we know **store** x 1; **flush** x is executed before **store** z 1. Therefore, if upon recovery z=1 (i.e. **store** z 1 has persisted before the crash), then x=1 (**store** x 1; **flush** x must have also persisted before the crash). As before, replacing **flush** x in Fig. 2.7e with **flush**$_{\text{opt}}$ $x; C$ yields the same result upon recovery when C is an **sfence**/**mfence** or an RMW.

Fig. 2.8 summarize the ordering constraints of the subset of the Px86 instructions, that PIEROGI$_{\text{simp}}$ supports.

2.2.2.1 A summary of existing formalizations of Px86

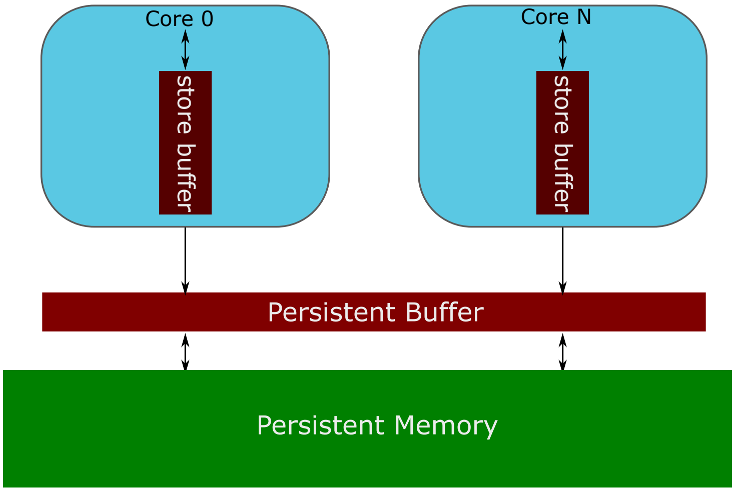


Figure 2.9: The PTSO memory model, as featured in [121]. The volatile components are illustrated in red color (store buffers, persistent buffer), and the non-volatile components are illustrated in green color (persistent memory).

In [121], Raad *et al.* provide operational and declarative semantics for the x86 and SPARC architectures that integrate persistent memory. The formalization (PTSO) incorporates a variation of buffered epoch persistency §2.2.1 as an extension of the Total Store Order consistency model proposed by Sewell et al. [131]. PTSO obtains an additional buffer compared to the standard TSO model, called persistent buffer, which is located between the store buffers and the persistent memory. Specifically, the PTSO model consists of (volatile) local per thread store buffers, a global volatile persistent buffer, and a module of (nonvolatile) persistent memory (Fig. 2.9). In the event of a crash, the only surviving writes are those in the persistent memory module.

When a thread executes a write, it is recorded to its store buffer. Writes are evicted from the store buffer and reach the persistent buffer in FIFO order, either when the store buffer is full or when the thread issues an mfence/CAS instruction, which causes the store buffer to drain. The persistent buffer is modeled as a FIFO queue of sub-buffers. Each sub-buffer represents a distinct epoch. Writes are propagating from the persistent buffer to the persistent memory in a non-deterministic way that reflects the order in which they persist. Nevertheless, the write propagation to persistent memory respects *persist atomicity*. A persist barrier (pfence) enforces epoch ordering by introducing a new empty epoch sub-buffer in the persistent buffer and requiring the corresponding thread buffer to drain. Explicit persists are accomplished with the psync instruction, which in this model stalls the program execution until the pending writes in the persistent buffer have been propagated to the persistent memory and the persistent buffer is drained.

Subsequent work by Raad *et al.* [122] refines the previously mentioned model (Fig.2.9) to more accurately reflect the persistency characteristics of Intel-x86, as detailed in the Intel manual [1]. This adaptation encompasses various persistence primitives such as `clflush`, `clflushopt` and `sfence` (see §2.1.3). The persistent buffer is used again to model the ordering constraints of persists. This paper presents two versions of Px86 operational semantics, one faithful to the Intel manual and one "corrected" version that aims to reflect the intended behavior of Intel's persistence primitives. As before, the time in which an instruction is exiting the store buffer corresponds to the point at which its effects become visible to the other threads, thus impacting VMO. Analogously, the time in which an instruction is exiting the persistent buffer reflects the point at which its effects become persistent, thus impacting PMO. More recent work [120], extends this model to include non-temporal writes and reads/writes to a richer set of Intel-x86 memory types.

2.3 Background work on Software Transactional Memory

This section begins by presenting a set of definitions for formalizing the concepts of Software Transactional Memory which will be utilized throughout this thesis (§2.3.1). Later on, we provide an overview of the background work concerning correctness conditions for (volatile) Software Transactional Memory (§ 2.3.2). Even though we present here numerous correctness conditions, in this thesis, we adapt only *opacity* (Def. 2.3.10) to the persistency setting. The purpose of presenting multiple correctness conditions is to compare their respective strengths and weaknesses as well as motivate the selection of opacity as the most suitable descriptor of correctness for Software Transactional Memory algorithms (STMs).

2.3.1 Definitions for Formalizing Software Transactional Memory

Correctness conditions for STM have predominantly been defined on *histories* over an implementation. A history (Def. 2.3.1) is a sequence of events (invocations and responses) that records all the interactions between the implementation (STM) and its clients (programs that are executed by concurrent threads and use STM operations to achieve synchronization).

Definition 2.3.1 (History). A history is a sequence of events. An event is either (1) an invocation (*inv*) or (2) a response (*resp*) of an operation π out of a set of operations Op.

A client is interacting with an STM implementation by invoking an STM operation (π) (i.e. $inv_t(\pi)$ event) and an STM implementation is interacting with a client by providing a response for the invoked operation (i.e. $resp_t(\pi)$ event). Invocation and response events of the same operation are said to *match* . Events are further parameterized by thread or transaction identifiers from a set Tid ($t \in$ Tid). In a broader context, a thread is capable of executing multiple transactions as long as each transaction is linked to a single thread. Nevertheless, for simplicity in the current model, we assume that each thread executes no more than one transaction, thus transaction identifiers coincide with thread identifiers.

STMs usually provide a number of operations to programmers on shared objects. In this work, we are interested only in read/write objects, and thus, all the following definitions of *validity* and *legality* of transactional histories are specific to such objects. To elaborate, we are only interested in histories that support operations to start (`TMBegin`) and commit (`TMCommit`) a transaction and operations to read and write shared objects

invocations	possible matching responses
$inv_t(\texttt{TMBegin})$	$resp_t(\texttt{TMBegin}(ok), resp_t(\texttt{TMBegin}(abort))$
$inv_t(\texttt{TMCommit})$	$resp_t(\texttt{TMCommit}(commit))$, $resp_t(\texttt{TMCommit}(abort))$
$inv_t(\texttt{TMRead}(x))$	$resp_t(\texttt{TMRead}(v))$, $resp_t(\texttt{TMRead}(abort))$
$inv_t(\texttt{TMWrite}(x, v))$	$resp_t(\texttt{TMWrite}(ok))$, $resp_t(\texttt{TMWrite}(abort))$

Table 2.1: TML history events where $t \in$ Tid, $x \in$ Loc where x is a read/write object, and $v \in$ Val.

($\texttt{TMRead}, \texttt{TMWrite}$). We denote the set of allowed-to-be-accessed by an STM implementation locations (shared objects) as Loc and the set of values that can be assigned or be read as Val. Table 2.1 synopsizes the events that may appear in a history of read-/write objects. As shown, all operations might potentially respond with `abort`, thereby aborting the whole transaction.

We use the following notation on histories: for a history h, $h_{|t}$ is the projection onto the events of transaction t only, and $h[i..j]$ the subsequence of h from $h(i)$ to $h(j)$ inclusive. For a response event e, we let $rval(e)$ denote the value returned by e; for instance, $rval(\texttt{TMBegin}(\texttt{ok})) = \texttt{ok}$. If e is not a response event, then we let $rval(e) = \perp$. We say that two histories h_1, h_2 are equivalent, writing $h_1 \equiv h_2$, if they consist of exactly the same events.

A history h is *alternating* if $h = \epsilon$ or is an alternating sequence of invocation and matching response events starting with an invocation. In other words, h is alternating if it does not contain concurrent operations. For the rest of this thesis, we assume each process invokes at most one operation at a time, and hence, we assume that $h_{|t}$ is alternating for any history h and transaction t. Note that this does not necessarily mean h is alternating itself.

Correctness conditions of transactional memory are usually defined over well-formed histories. Well-formedness formalizes the allowable interaction between an STM implementation and its clients. Here, we consider the following definition of *well-formedness.*

Definition 2.3.2 (Transactional Well-Formedness). A history is ***transactionally well-formed*** if for every $t \in$ Tid, $h_{|t} =< s_0, \ldots, s_m >$ is an alternating history such that $s_0 = inv_t(\texttt{TMBegin})$, for all $0 < i < m$, event $s_i \neq inv_t(\texttt{TMBegin})$ and $rval(s_i) \notin \{\texttt{commit}, \texttt{abort}\}$.

It is important to note that the definition of well-formedness does not allow the reuse of transaction identifiers. A transaction t is considered *committed* if $rval(s_m) = \texttt{commit}$ and *aborted* if $rval(s_m) = \texttt{abort}$. In either case, the transaction t is *completed*; otherwise, t is *live*.

Given a *well-formed* transactional history h, we define a *real-time* order on its transactions denoted as $\prec_h$. We say that $t_1 \prec_h t_2$ if transaction t_1 completes its commit operation before transaction t_2 starts, indicating the real-time ordering of transactions. If neither $t_1 \prec_h t_2$ nor $t_2 \prec_h t_1$ holds, we consider transactions t_1 and t_2 to be concurrent. A history h is *non-interleaved* if it does not contain concurrent transactions. The relation $\prec_h$ constitutes a partial order for h.

Furthermore, given a *well-formed* transactional history h, we define an *operation real-time* order on its transactions, denoted as $\prec_h^{op}$. Specifically, $op_1 \prec_h^{op} op_2$ if operation

op_1 completes before operation op_2. In other words, $op_1 \prec_h^{op} op_2$ holds if the *response* event of op_1 precedes the *invocation* event of op_2 in h. Similarly, operations op_1 and op_2 are concurrent in h when neither $op_1 \prec_h^{op} op_2$ nor $op_2 \prec_h^{op} op_1$ holds. The relation $\prec_h^{op}$ constitutes a partial order for h.

Let's consider an alternating history h_l, which is equivalent to a history h. We can say that h_l respects $\prec$ if for every pair (t_1, t_2) of transactions in h for which $t_1 \prec_h t_2$ holds then $t_1 \prec_{h_l} t_2$ also holds. Moreover, h_l respects $\prec^{op}$ on the set of operations in h, if for every pair (op_1, op_2) of operations in h for which $op_1 \prec_h^{op} op_2$ holds, $op_1 \prec_{h_l}^{op} op_2$ also holds.

To ensure the correctness of an STM history, a minimum two-step procedure is typically followed. The first step involves reordering the transactions of the concurrent STM history to form an equivalent (consisting of the same events) history that has no interleaving of transactions (*sequential history*). Secondly, it should be shown that this reordering is permissible based on the given correctness criterion.

A sequential history has to ensure that the behavior is meaningful with respect to the reads and writes of the transactions. Below, we first formalize the *sequential history* semantics.

Definition 2.3.3 (Valid history)**.** We say an alternating history h is *valid* iff there exists a sequence of stores $\sigma_0, \ldots, \sigma_n \in (\textsc{Loc} \to \textsc{Val})^*$ such that $\sigma_0(x) = 0$ for all $x \in \textsc{Loc}$, and for all i such that $0 \leq i < n$ and $t \in \textsc{Tid}$:

1) if $ev_{2i} = inv_t(\texttt{TMWrite}(x, v))$ and $ev_{2i+1} = resp_t(\texttt{TMWrite}(ok))$ then $\sigma_{i+1} = \sigma_i[x := v]$,
2) if $ev_{2i} = inv_t(\texttt{TMRead}(x))$ and $ev_{2i+1} = resp_t(\texttt{TMRead}(v))$ then $\sigma_i(x) = v$ and $\sigma_{i+1} = \sigma_i$,
3) for all other pairs of events (reads and writes with an abort response, as well as begin and commit events), we require $\sigma_{i+1} = \sigma_i$.

We write $[\![h]\!](\sigma)$ if σ is a sequence of states that makes h valid (since the sequence is unique, if it exists, it can be viewed as the semantics of h).

The point of TM is that the effect of the writes only takes place if the transaction commits. Writes in a transaction that aborts do not affect the memory. However, all reads, including those executed by aborted transactions, must be consistent with previously committed writes. Therefore, only some histories of an object reflect ones that could be produced by a TM. We call these the *legal* histories, and they are defined as follows:

Definition 2.3.4 (Legal histories)**.** Let hs be a non-interleaved history and i an index of hs. Let hs' be the projection of $hs[0..(i-1)]$ onto all events of committed transactions plus the events of the transaction to which $hs(i)$ belongs. Then we say *hs is legal at i* whenever hs' is valid. We say *hs is legal* iff it is legal at each index i. Also, a *legal* transaction is defined as a transaction that belongs to a *legal* history.

Notice here that the definition of legality (Def. 2.3.4) matches the definition given by Dziuma *et al.* [54], according to which a well-formed, non-interleaved history h is *legal* if for each transaction $t \in \textsc{Tid}$ the following points holds for $h_{|t}$:

For every read invocation event by t ($e = inv_t(\texttt{TMRead}(x))$ whose response value is not equal to *abort* ($rval(e) \neq abort$):

- If an $inv_t(\texttt{TMWrite}(x, v))$ precedes e in h, then there exists a subsequent event e' in $h_{|t}$ such that $e' = resp_t(\texttt{TMRead}(v))$,
- otherwise, if there is no committed transaction t' preceding t in h that performed a successful `TMWrite` operation on x, then there exists a subsequent event e' in $h_{|t}$ such that $e' = resp_t(\texttt{TMRead}(v))$, where v is the initial value of address x.
- Finally, if t' is the last transaction that precedes t and performed a successful `TMWrite` operation on x and $e'' = inv_{t'}(\texttt{TMWrite}(x, v))$ is the invocation event of the last successful `TMWrite` operation on x by t', then there exists a subsequent event e' in $h_{|t}$ such that $e' = resp_t(\texttt{TMRead}(v))$.

In all the following definitions we denote by $\mathcal{S}$ the set of all possible well-formed legal histories. We also call any history h that belongs to $\mathcal{S}$ ($h \in \mathcal{S}$) *sequential history.*

A given concrete history h may be incomplete, i.e., it may contain pending operations, represented by invocations that do not have matching responses, or it may obtain transactions that are *live* and have not yet invoked a commit operation. The corresponding sequential history h' must decide for each pending commit invocation: either by adding a responding event e for which $rval(e) \neq abort$ (the effect has taken place), or by adding a responding event e for which $rval(e) = abort$ (the effect has not taken place). The transactions that obtain a pending operation that is not `TMCommit`, are semantically equivalent to aborted transactions, thus in h' each pending operation that is not `TMCommit` is completed with a responding event e for which $rval(e) = abort$. Live transactions in h that have not yet invoked a commit operation are also semantically equivalent to aborted transactions, and thus, in h' they are completed with an invocation event e and responding `TMCommit` event e' for which $rval(e') = abort$.

We denote by $complete(h)$ the set of all possible completions of h that can be formed according to the above description. In other words, its history in $complete(h)$ is an extension of h for which all commit pending transactions appear as aborted or committed, and all the other live transactions appear as aborted.

2.3.2 Correctness Conditions for Software Transactional Memory

This section describes a collection of well-known correctness conditions for volatile Software Transactional Memory. To formalize these correctness conditions, we closely adhere to the formalization proposed by Dziuma *et al.* [54]. For the following definitions, we will use the notation $comm(h)$ to define the subhistory that is formed by all events of h that belong to committed transactions. Below, transactional histories are depicted as diagrams. The following conventions have been made. We denote `TMBegin` as `B`, `TMRead` as `R`, `TMWrite` as `W`, and `TMCommit` as `C`. Time is increasing from the left to the right. Its horizontal dotted line depicts a thread/transaction execution line. For brevity, transactional operations are represented with a line that indicates their duration. The left (resp. right) vertical line to a transactional operation line represents its invocation (resp. response) event. For example, instead of writing $inv_t(\texttt{TMBegin}), resp_t(\texttt{TMBegin}(\texttt{ok}))$ to indicate the successful initialization of the transaction t we attach to the execution line of transaction

t a tangent line with a subscript: `B(ok)`.The operations of aborted and live transactions are depicted in red. The events or response values added to the corresponding sequential history by the function *complete* are depicted in blue.

Strict Serializability: *Strict serializability* was first introduced in [116] as a consistency condition of database systems transactions. In the context of transactional memory, a concurrent history h satisfies strict serializability if it can be mapped to sequential history in which (1) the order of events inside each transaction is preserved, (2) the *real-time ordering* of the transactions is preserved (only transactions that overlap can be reordered). The transactions that haven't been committed at the concurrent history might be included in the mapped sequential history as committed (along with a commit event) or they can be omitted. Strict serializability does not impose any restriction on the values that non-committed (live or aborted) transactions can read. However, committed transactions can not read values that have been written from non-committed transactions. Strict serializability is a non-prefix-closed property.

Definition 2.3.5 (Strict Serializability)**.** A concurrent history h satisfies *strict serializability* if there exists a history $h' \in complete(h)$ and a history s equivalent to $comm(h')$ $(s \equiv comm(h'))$ such that

1) $s \in \mathcal{S}$, and

2) $t_1 \prec_{comm(h')} t_2$ implies $t_1 \prec_s t_2$.

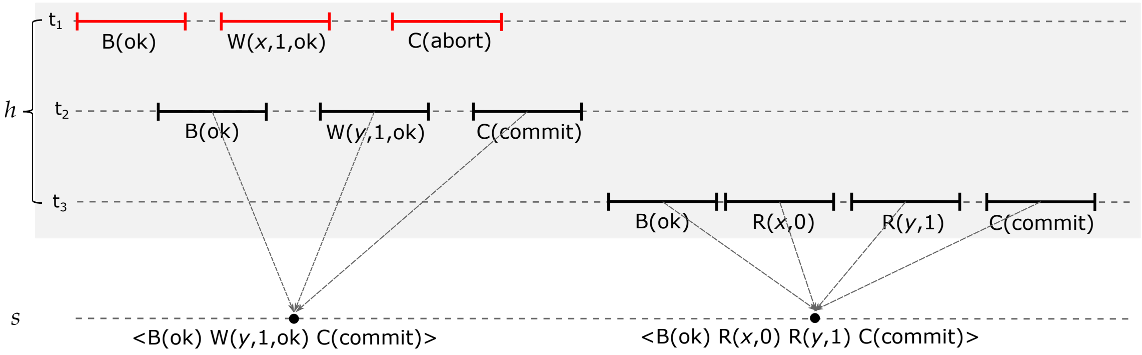


Figure 2.10: A strictly serializable history h.

Fig. 2.10 depicts a strictly serializable history h. In this example, $complete(h)$ returns a set with only history h itself, as h does not contain incomplete transactions. On the other hand, $comm(h')$ contains all the events of $h' \in complete(h)$ except from the events of transaction t_1, which is aborted. The sequential history s which corresponds to the transaction ordering $t_1 \prec t_2$, belongs to $\mathcal{S}$ and respects the real-time ordering of $comm(h')$. Therefore, h is strictly serializable.

Serializability: *Serializability* is a weaker condition than strict serializability which was also introduced in [116]. Serializability and strict serializability differ in the fact that the former does not require the sequential history, which is equivalent to the concurrent one, to respect the real-time order of the transactions. Serializability is also a non-prefix-closed property.

Definition 2.3.6 (Serializability). A concurrent history h satisfies *serializability* if there exists a history $h' \in \mathit{complete}(h)$ and a history s equivalent to h' ($s \equiv \mathit{comm}(h')$) such that $s \in \mathcal{S}$.

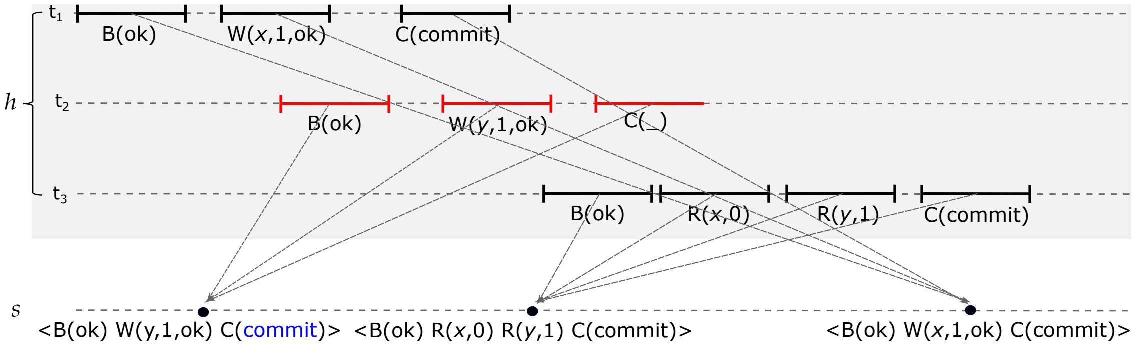

t1
B(ok)
W(x,1,ok)
C(commit)
h
t2
B(ok)
W(y,1,ok)
C(_)
t3
B(ok)
R(x,0)
R(y,1)
C(commit)
s
<B(ok) W(y,1,ok) C(commit)>
<B(ok) R(x,0) R(y,1) C(commit)>
<B(ok) W(x,1,ok) C(commit)>

Figure 2.11: A serializable history h.

Fig. 2.11 depicts a serializable history h. In this example, $\mathit{complete}(h)$ returns a set with two histories that complete h in two possible ways. One element of $\mathit{complete}(h)$ aborts the transaction t_2, while the other, h', commits t_2 successfully. $\mathit{comm}(h')$ contains all the events of h'. The equivalent to $\mathit{comm}(h')$ sequential history s that is depicted in Fig. 2.11 corresponds to the transaction ordering $t_2 \prec t_3 \prec t_1$, and belongs to $\mathcal{S}$. Thus, h is serializable.

Causal Related conditions: For the next two correctness criteria, following the representation of [54], we introduce the following definitions:

Definition 2.3.7 (read-from binary relation). Given an alternating history l, which is equivalent to a concurrent history h ($h \equiv l$), we say that transaction t_1 *reads from* transaction t_2 ($t_2 \prec_l^r t_1$) in h if:

1) Transaction t_2 executes a successful `TMRead` operation π, on address x and returns a value v.

2) Transaction t_1 is the transaction in l that executes the most recent successful `TMWrite` operation, which stores the value v to address x and precedes π.

We denote as $\mathcal{R}_h$ the set of all *read-from* relations that can be derived from h. Let $\prec^r \in \mathcal{R}_h$. The *causal* relation corresponding to $\prec^r$ is defined as the transitive closure of $(\bigcup_i \prec_{h_{|t_i}}) \cup \prec^r$. We assume that $\mathcal{C}_h$ represents the set of all causal relations which are present in h.

Causal consistency: Causal consistency [5, 75] allows each thread/transaction to observe a different view of the transactional history order, with the condition that each view respects the same *causal relation*. Causal consistency is a weaker form of consistency than serializability, which requires that all transactions appear to execute in the same order. However, it is still a useful property for ensuring correctness in concurrent systems, particularly in cases where the order of transactions doesn't matter as long as their effects are correctly propagated. In the case that each transaction's view is the same, causal consistency reduces to serializability.

Definition 2.3.8 (Causal consistency)**.** Formally, a concurrent history h is *causally consistent* if there exists a history $h' \in complete(h)$ and a causal relation $\prec^c \in \mathcal{C}_{comm(h')}$ such that for every transaction t_i in h there exists a non-interleaved history l_i that satisfies the following conditions:

1) l_i is equivalent to $comm(h')(l \equiv h')$,
2) l_i satisfies $\prec^c$ and
3) t_i is *legal*

History h, as depicted in Fig. 2.12 is causal consistent but not serializable. Causal consistency allows its transaction to view a different sequential execution of h. For example, in this case, transaction t_3 observes a sequential execution in which t_2 precedes t_1, while transaction t_4 observes a sequential execution in which t_1 precedes t_2.

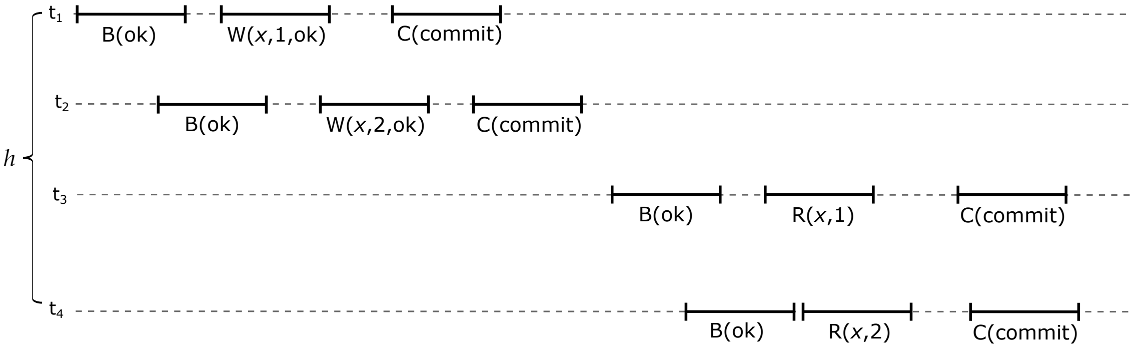


Figure 2.12: A causal consistent history h, that is not serializable.

Causal Serializability: Causal serializability [126] is a stronger condition than causal consistency but weaker than serializability. Essentially, a concurrent history h is causally serializable if it meets the constraints imposed by causal consistency and also if all transactions in h perceive the same order for transactions that update the same location ($\in$ Loc).

Definition 2.3.9 (Causal serializability)**.** A concurrent history h that consists of n transactions is *causally serializable* if there exists a history $h' \in complete(h)$ and a causal relation $\prec^c \in \mathcal{C}_{comm(h')}$ such that for every transaction t_i ($i \in [0, n]$), there exists a non-interleaved history l_i that satisfies the following conditions:

1) l_i is equivalent to $comm(h')$,
2) l_i satisfies $\prec^c$,
3) t_i is *legal* and
4) For each pair of transactions (t_1, t_2) in $comm(h')$ that write on the same address, if $t_1 \prec_{l_i} t_2$ then for all other transactions in h ($i \in [0, n]$), $t_1 \prec_{l_i} t_2$.

The history of Fig. 2.12 is not causally serializable. This is because causal serializability imposes that transactions t_1 and t_2 are ordered in the same way, in all the views of the transactions that constitute h. If the ordering of the transactions in l_{t3} (as defined in 2.3.9) places transaction t_2 before t_1 then l_{t4} should also maintain the same order for t_1 and t_2 leading to an invalid history.

Opacity: We now introduce opacity, the correctness condition upon which we base our proposed criterion (Chapter 3) for persistent transactional memory algorithms.

Opacity [67] is a prefix-closed condition which is stronger than strict serializability. In order for a concurrent history h to be opaque, it should be mapped to a sequential history that preserves the order of the events inside each transaction and the real-time order of the transactions, as with strict serializability. However, non-committed transactions (i.e., aborted, or live transactions) can only read values written by themselves or previously committed transactions.

Above, we summarize the main characteristics of opacity. From now on, we will use the term *internal* read for a read that a transaction performs to locations that it has previously written by itself, and *external* read for a read that a transaction performs to a location that has been written by another transaction.

- Even though the transactions of the concurrent history h can be reordered in the matching sequential history s, the operations within a transaction cannot be reordered.
- Transactions within s should respect the *real-time ordering* constraint. To elaborate, only overlapping transactions in h can be reordered in s. Not overlapping transactions should maintain their order.
- All transactions, including live and aborted transactions, should appear to perform internal reads that are consistent with their previous writes. This means that any internal read of h by a transaction t to a location x should return the last written value by t to x.
- All committed transactions in h must be ordered in s such that the previous constraints are satisfied and consistency is ensured with all previously committed transactions. This means that any external read of a committed transaction to a location x in s should return the value of the immediately preceding in s write performed by a committed transaction at x. In addition, each write of a committed transaction in s must modify memory in an appropriate manner, and become visible to the proceeding transactions.
- The main difference between opacity and other STM correctness conditions is motivated by the potential problems a memory transaction's (unlike a database transaction's) access to an inconsistent state can cause, even if it is later aborted. To prevent any potential problems, opacity requires the *external* reads of aborted or *live* transactions in s to also be consistent with all the previously committed transactions. However, writes of live or aborted transactions must not modify memory. Subsequently, no committed transaction should appear to read a value written by live or aborted transactions in s.

Definition 2.3.10 (Opacity)**.** Formally, a well-formed concurrent history h is *end-to-end opaque* if there exists a history $h' \in complete(h)$ such that there exists a history s equivalent to h' ($s \equiv h'$) that satisfies the following conditions:

1) $s \in \mathcal{S}$, and
2) $t_1 \prec_{h'} t_2$ implies $t_1 \prec_s t_2$.

A history h is *opaque* iff its prefix of h is end-to-end opaque.

Fig. 2.13 depicts a non-opaque history. In h_1 transaction t_1 reads 0 to x for the initial memory. Then transaction t_2 stores 1 at x and y and commits. Finally, transaction t_1 reads 1 at y. In this example, it is impossible to order the transactions sequentially such that Def. 2.3.10 is satisfied. Either t_1 must be ordered first, in which case it should have read the values 0 for both variables x and y, or t_2 must be ordered first, in which case t_1 must have read the newly written value (1) for x and y.

History h_2 as illustrated in Fig. 2.14 is opaque. The transaction order in the corresponding sequential history s is $t_1 \prec t_2 \prec t_3$. Notice that transaction t_2 reads 0 to y despite the fact that transaction t_3 had previously written 1 to y. This is desirable as transaction t_3 eventually aborts, and thus any writes performed by t_3 should not be visible to other transactions. Even though transaction t_2 is *live* in h, it is consistent with the previously committed transactions in h. Specifically, t_2 firstly reads 1 to x, which is the last written value on x by a committed transaction (t_1), and then reads 0 to y. Address y has not yet been updated by any committed transaction; thus, the value read is its initial value. In this example, h' completes history h by adding an aborted commit operation to transaction t_2.

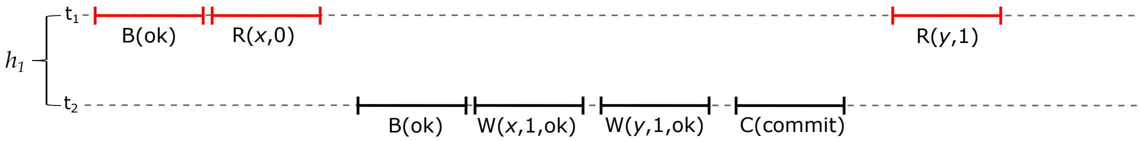
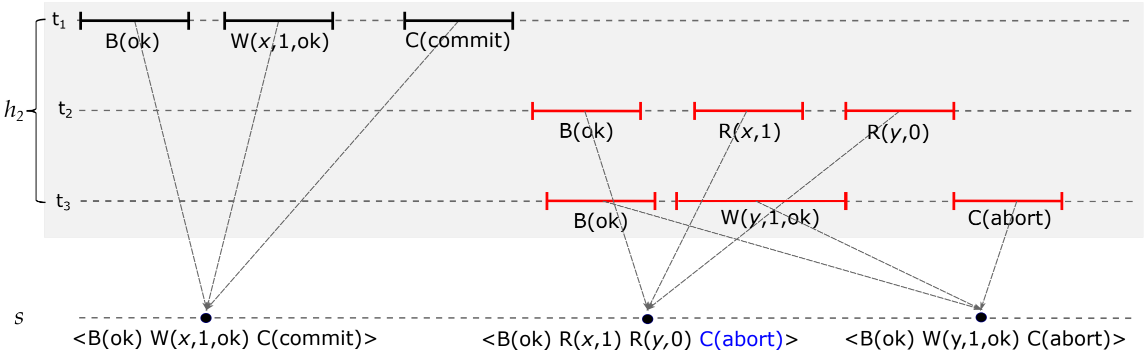


Figure 2.13: A non-opaque history (h_1) .



h_2
t₁
B(ok)
W(x,1,ok)
C(commit)
t₂
B(ok)
R(x,1)
R(y,0)
t₃
B(ok)
W(y,1,ok)
C(abort)
s
<B(ok) W(x,1,ok) C(commit)>
<B(ok) R(x,1) R(y,0) C(abort)>
<B(ok) W(y,1,ok) C(abort)>

Figure 2.14: An opaque history (h_2) .

Weaker variations of opacity: Although opacity can offer robust safety guarantees that render it suitable for transactional memory, some may view it as overly restrictive and needlessly complex to implement in TM systems. This is primarily due to its requirement that every live and abort transaction must be consistent with all prior committed transactions. Below is a brief overview of some less strict correctness conditions that aim to modify various aspects of opacity while preserving its essential safety guarantees.

Virtual world consistency [76] combines opacity with causal consistency. Like opacity, it applies to all transactions (both committed and aborted/live ones). Informally, it guarantees that (1) all committed transactions are serializable, and (2) each (reduced) aborted/live transaction only reads values that are mutually consistent when considering its causal past. While all committed transactions share the same witness sequential execution, each aborted /live transaction has its own unique witness sequential execution that pertains only to its past. Thus, two aborted transactions can possess different witnesses that offer them consistent values but might not be compatible with each other.

Elastic opacity [57] is a safety property that allows the relaxation of the atomicity requirement of transactions. Specifically, elastic opacity allows a transaction to be split into a series of smaller subtransactions, which are referred to as a cut. These subtransactions can be executed independently and in parallel without compromising the consistency and correctness of the overall transaction. The size of these subtransactions can be dynamically adjusted based on conflict detection, allowing for increased concurrency.

In contrast to opacity, *last use opacity* [134] permits a transaction to read from a live transaction if it is commit-pending or if it is still live and has already executed its closing write on the variable being read. The term closing write refers to the last `TMWrite` operation performed by a transaction on a particular address before committing or aborting. This approach enables *early release*, which is a technique that allows a transaction to explicitly remove an address from its *read set* when it no longer relies on its corresponding value. The benefit of *early release* is especially notable (though not exclusively) in systems that use pessimistic concurrency control [133].

2.4 Background work on correctness conditions for concurrent objects

This section starts with providing some definitions for formalizing the correctness of concurrent objects (§2.4.1). We then describe *linearizability* [73] (Def. 2.4.1), the predominant correctness condition for volatile concurrent objects, along with various adaptations of *linearizability* for durable concurrent objects (§2.4.2). Although we discuss various correctness conditions, we only use *durable linearizability* (Def. 2.4.5) to adapt opacity to the persistency setting, which is our focus in later chapters. Finally, we provide a short description of mechanisms for detecting linearized operations after a system crash (§2.4.3).

2.4.1 Definitions for Formalizing Correctness of Concurrent Objects

The advent of persistent memory has spurred research interest in defining notions of correctness that are able to specify the allowable states of a system after a crash. In this section, we review a subset of durable concurrent correctness conditions, which have led to the design of a number of persistent concurrent data structures. In Chapter 3 we transfer

invocations	possible matching responses
$inv_t(\mathtt{R}(x))$	$resp_t(\mathtt{R}(v))$
$inv_t(\mathtt{W}(x, v))$	$resp_t(\mathtt{W}(ok))$

Table 2.2: History events where $t \in \textsc{Tid}$, x is a read/write object and $x \in \textsc{Loc}, andv \in \textsc{Val}$.

the principle of durable concurrent correctness to the area of software transactional memory.

As before, correctness conditions for data structures are defined over histories of implementations. Histories are sequences of events, but now events are not confined to transactions. In the persistency setting, an event can be an invocation, a response, or a crash. We are as before interested in sequential read/write objects, thus we consider histories with only read and write operations ($\pi \in \{\mathtt{R}(x), \mathtt{W}(x, v)\}$ where $x \in \textsc{Loc}, v \in \textsc{Val}$). Table 2.2 summarizes the invocation events along with the possible responses that occur in a history h. An operation is *completed* in h if both its invocation and response event are appearing in h. Otherwise, it is *pending*. We introduce the function $complete_o(h)$, which generates all feasible completions of h by removing a subset of *pending* operation events from h and adding a response event to its remaining pending operations at the end of history h.

Invocation and response events are parametrized by a thread id ($t \in \textsc{Tid}$) that indicates the thread that issues the corresponding operation. Given a history h we define a *real-time* order (partial order) on its operations denoted as $\prec_h$. Specifically, $\pi_1 \prec_h \pi_2$ if the response event of operation π_1 precedes the invocation event of π_2. If neither $\pi_1 \prec_h \pi_2$ nor $\pi_2 \prec_h \pi_1$ holds, then π_1 and π_2 are considered to be concurrent. The subsequence $h_{|t}$ is the projection onto the events of h over transaction t, while the subsequence $h_{|x}$ is the projection onto the events of h over object x. A history h is *sequential* if $h = \epsilon$ or is an alternating sequence of invocation and matching response events, except possibly the last event, starting with an invocation. As before, histories h and h' are equivalent, denoted by $h \equiv h'$, if $h_{|t} = h'_{|t}$ for all $t \in \textsc{Tid}$. A history h is *well-formed* if $h_{|t}$ is sequential for every $t \in \textsc{Tid}$. A sequential history h is *legal* if every subsequence of h for object $x, h_{|x}$, adheres to the sequential specification of x. In our case, that x is a read/write object *legality* implies that every successful $\mathtt{R}(x)$ where $x \in \textsc{Loc}$ must return the last successfully written value on x. We denote the set of the legal sequential histories as generated by a sequential read/write object as $\mathcal{S}_o$. Finally, a property P is said to be local for a history h if for every object x, if $h_{|x}$ satisfies P, then h also satisfies P.

2.4.2 Linearizability and Adaptations of Linearizability to the Persistency Setting.

In this section, we begin by providing a definition of *linearizability*, which is the most commonly used correctness criterion for concurrent objects. Afterward, we provide a number of correctness conditions that can be seen as extensions of linearizability for the persistency setting.

Linearizability: *Linearizability*, first introduced in [73], is the standard correctness condition for concurrent objects. In order for a concurrent history h to be linearizable, it should be mapped to a semantically valid, sequential history that includes the operations of all the different threads that are part of the h, combined. Two conditions should be met. Firstly, the history of execution of the operations of each thread ($h_{|t}$) should be sequential, and each operation of $h_{|t}$ should appear to take effect atomically - at some point between its invocation and response, at the combined sequential history. The point of execution of an operation is known as the *linearization point*. Secondly, the real-time order of the non-concurrent operations of the concurrent history should be preserved in the combined sequential history. It should be mentioned that operations that were invoked but didn't respond at h can be excluded from the sequential history or can be included along with added responses.

Definition 2.4.1 (Linearizability). A concurrent history h is *linearizable* if there exists a history $h' \in complete_o(h)$ such that there exist a history s equivalent to h' ($s \equiv h'$) that satisfies the following conditions:

1) $s \in \mathcal{S}_o$, and

2) $t_1 \prec_{h'} t_2$ implies $t_1 \prec_s t_2$.

Linearizability can be considered a special case of *strict serializability* where transactions consist of a single operation. Fig. 2.15 depicts a history that is not linearizable (h_a) and a linearizable history (h_b). History $h_a \triangleq \langle\ inv_{t_1}(\texttt{W}(x,8)),\ inv_{t_2}(\texttt{R}(x)),\ resp_{t_2}(\texttt{R}(8)),\ inv_{t_2}(\texttt{R}(x)),\ resp_{t_2}(\texttt{R}(0))\rangle$ can be completed in two ways.

The first way involves completing the pending write operation on x by thread t_1 ($h' \triangleq \langle\ inv_{t_1}(\texttt{W}(x,8)),\ inv_{t_2}(\texttt{R}(x)),\ resp_{t_2}(\texttt{R}(8)),\ inv_{t_2}(\texttt{R}(x)),\ resp_{t_2}(\texttt{R}(0)),\ resp_{t_1}(\texttt{W}(ok))\ \rangle$). In this case, there is no legal sequential history equivalent to h'. If the write of value 8 at x by t_1 is placed before the first read of x by t_2, then the second read of x by t_2 become invalid. If the pending write is placed in the middle of the two reads of x by t_2 then both become invalid as the value 8 at x cannot be read before the write of 8 at x takes place, and t_2 cannot read the initial value of x if previously t_1 has written the value 8 at x. Finally, if the pending write is placed after the second read of t_2 the first read of t_2 becomes invalid.

The second way involves omitting the pending write operation on x by thread t_1 ($h' \triangleq \langle\ inv_{t_2}(\texttt{R}(x)),\ resp_{t_2}(\texttt{R}(8)),\ inv_{t_2}(\texttt{R}(x)),\ resp_{t_2}(\texttt{R}(0))\rangle$) In this case, again there is no legal sequential history equivalent to h', as the t_2 is obliged to read only the initial value of x.

History $h_b \triangleq \langle\ inv_{t_1}(\texttt{W}(x,1)),\ inv_{t_2}(\texttt{W}(x,3)),\ resp_{t_1}(\texttt{W}(ok)),\ inv_{t_1}(\texttt{W}(x,2)),\ resp_{t_2}(\texttt{W}(ok)),\ resp_{t_1}(\texttt{W}(ok)),\ inv_{t_2}(\texttt{R}(x)),\ resp_{t_2}(\texttt{R}(3))\rangle$ is linearizable. Placing the write of value 1 at x by t_1 before the write of value 2 at x by t_1 and the write of value 2 at x by t_1 before the write of value 3 at x by t_2 leads to a legal, sequential history that is equivalent to h and respects the real-time order of operations.

2.4.2.1 Persistent Memory Consistency conditions

Linearizability does not take into account any potential crash events. Since *persistent memory* enables the continuation of execution after a system crash, *linearizablity* has been extended to cover system crashes in several ways.

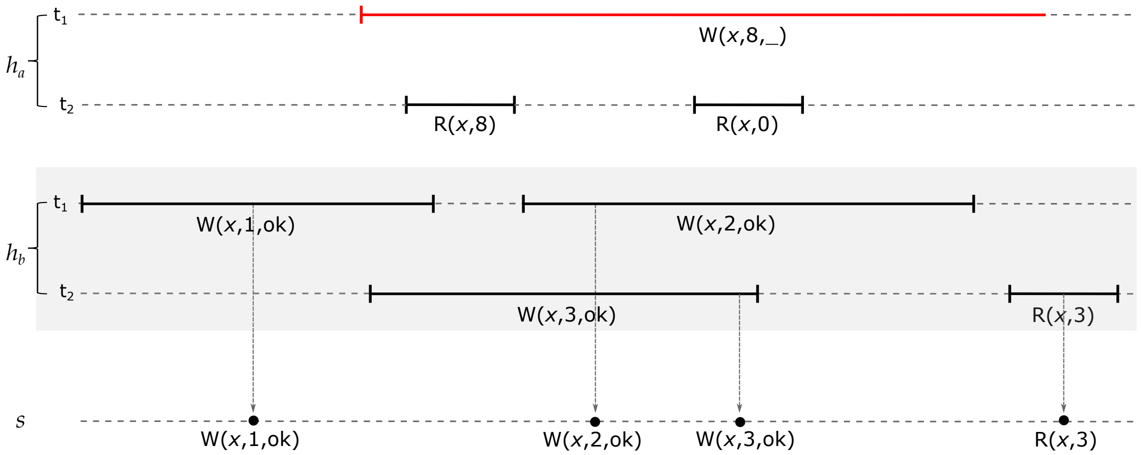


Figure 2.15: A not linearizable history h_a amd a linearizable history h_b.

Strict Linearizability: Aguilera *et al.* [4], defined *strict linearizability.* Strict linearizability ensures the limited effect of operations. Specifically, in the case that a thread crashes while executing an operation, it requires this operation to take effect between its invocation and the crash, but not after the crash. Following the formalization proposed in [21], we introduce a $scomplete_o(h)$, which generates all the possible strict completions of a concurrent history h by 1) adding a matching response for a subset of pending operations in h after the operation's invocation event and before the thread's crash event (if any), 2) removing from h crash events, the remaining pending operations and any subsequent invocation/response event issued by a crashed thread (if any) after the crash event.

Strict linearizability assumes that crashed threads do not continue their execution after the crash. A disadvantage of this model is that it has been proven to exclude some wait-free implementations.

Definition 2.4.2 (Strict Linearizability)**.** Formally, a concurrent history h is *strictly linearizable* if there exists a history $h' \in scomplete_o(h)$, and a history s equivalent to h' $(s \equiv h')$ such that that satisfies the following conditions:

1) $s \in \mathcal{S}_o$, and

2) $t_1 \prec_{h'} t_2$ implies $t_1 \prec_s t_2$.

History $h \triangleq \langle\ inv_{t_2}(\mathtt{W}(x,1)),\ inv_{t_1}(\mathtt{R}(x)),\ resp_{t_2}(\mathtt{W}(ok)),\ inv_{t_2}(\mathtt{W}(x,3)),\ resp_{t_1}(\mathtt{R}(1)),$ **Crash**, $inv_{t_2}(\mathtt{W}(y,1)),\ resp_{t_2}(\mathtt{W}(ok)),\ inv_{t_1}(\mathtt{R}(x)),\ inv_{t_2}(\mathtt{R}(x)),\ resp_{t_1}(\mathtt{R}(1)),\ resp_{t_2}(\mathtt{R}(3))$ $\rangle$ as depicted in Fig. 2.16, has a strict completion that can be mapped to a legal sequential history (s) which respects the real-time order of $h' \triangleq \langle\ inv_{t_2}(\mathtt{W}(x,1)),\ inv_{t_1}(\mathtt{R}(x)),$ $resp_{t_2}(\mathtt{W}(ok)),\ resp_{t_1}(\mathtt{R}(1)),\ inv_{t_1}(\mathtt{R}(x)),\ resp_{t_1}(\mathtt{R}(1))\ \rangle$. History h' does not contain the write of 3 at x from thread t_2, as well as the crash event and all the operations of t_2 after the crash event.

Persistent Atomicity: Guerraoui and Levy [69], have defined *persistent atomicity.* In their model, threads are crashing individually, and due to a recovery procedure that takes

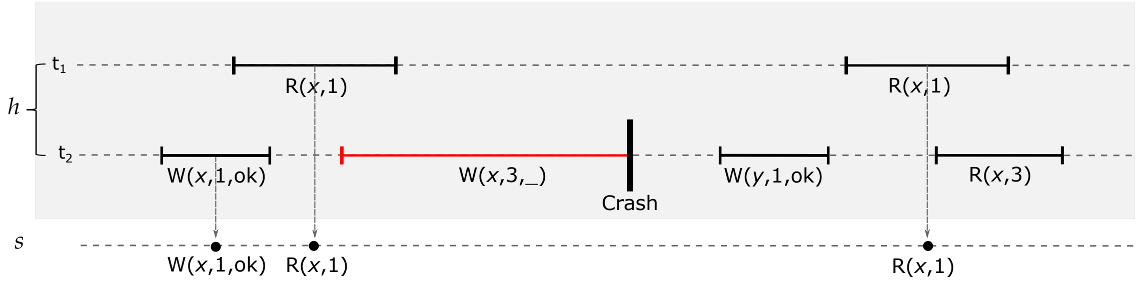


Figure 2.16: A strict linearizable history h.

place after a crash, their execution can resume after the crash. Informally, persistent atomicity ensures that a pending operation of a thread $t, t \in \textsc{Tid}$, that was interrupted by a crash event can take place between its invocation and the invocation of the first operation that thread t issues after the crash. This differs from strict linearizability, which requires any pending operation to take necessarily effect before a crash occurs. One of the limitations that this condition introduces is that it doesn't provide locality: the merge of correct object histories, is not necessarily correct. As before, to formalize persistent atomicity, we define a $\mathit{pcomplete}_o(h)$ function, which constructs all the possible atomically persistent completions of a concurrent history h by 1) adding a matching response for a subset of pending operations in h, after the operation's invocation event and before the thread's first invocation event after the first subsequent crash (if any), 2) removing all the crash events and the remaining pending operations from h.

Definition 2.4.3 (Persistent atomicity)**.** Formally, a concurrent history h adheres to *persistent atomicity* if there exists a history $h' \in \mathit{pcomplete}_o(h)$ and a history s equivalent to h' $(s \equiv h')$ that satisfies the following conditions:

1) $s \in \mathcal{S}_o$, and

2) $t_1 \prec_{h'} t_2$ implies $t_1 \prec_s t_2$.

The history of Fig 2.17 has at least two persistent atomic completions. The first one, which leads to the legal sequential history s_a, includes the pending write of 3 at address x. As seen, this operation is completed after the crash event of t_2 but before the invocation of the next operation of t_2 that follows the crash. In this example, completing this operation earlier would not affect the legality of the s_a. Unlike strictly linearizable completions, s_a does not omit the operations (at least the committed ones) of the crashed thread t_2 after the crash event. Since x at s_a obtains the value 3 after the crash, the subsequent pending read of x by t_2 is completed to return the value 3.

The second one, which leads to the legal sequential history s_b, omits the pending write of 3 at address x. Since in this x at s_b obtains the value 1 after the crash, the subsequent pending read of x by t_2 is completed to return 1.

Recoverable Linearizability: Berryhill *et al.* [21] have proposed *recoverable linearizability*. This condition strengthens persistent atomicity by allowing every pending operation to happen before its thread invokes anything on the same object post-crash. Although it provides locality, this condition doesn't provide consistency around the crash

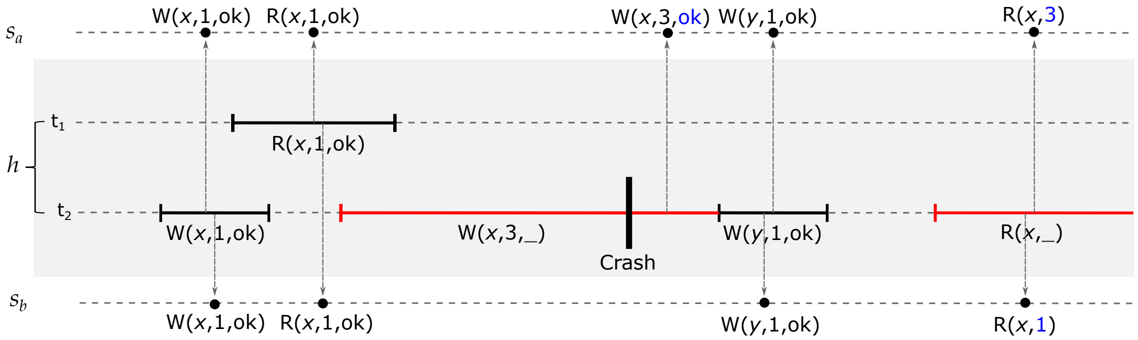


Figure 2.17: A persistent atomic history h.

-a thread can perform an operation on some other object before coming back to the pending operation, breaking well-formedness.

To formally define recoverable linearizability, we introduce a recoverable completion function ($rcomplete_o(h)$). This function presents exactly the same functionality as $scomplete_o(h)$ with the only exception that it does not remove subsequent to the crash event invocation/response events issued/received by a crashed thread. This is because the current model, as with persistent atomicity, assumes that immediately after a crash, a recovery process is taking place, allowing the crashed thread to resume execution. Opposite to the so far presented correctness conditions recoverable linearizability imposes ordering constraints based on the order of events of the original history h rather than the completed history h'.

To avoid the inversion of program order, a second partial order is defined on pairs of operations in h ($\ll_h$), according to which, $\pi_1 \ll_h \pi_2$ if both operations are issued by the same thread, access the same object, and the invocation event of π_1 precedes the invocation event of π_2 in history h.

Definition 2.4.4 (Recoverable Linearizability)**.** Formally, a concurrent history h adheres to *recoverable linearizablity* if there exists a history $h' \in rcomplete_o(h)$ and a history s equivalent to h' ($s \equiv h'$) such that:

1) $s \in \mathcal{S}_o$,

2) $t_1 \prec_h t_2$ implies $t_1 \prec_s t_2$, and

3) $t_1 \ll_h t_2$ implies $t_1 \ll_s t_2$.

Fig. 2.18 illustrates the same history h as Fig. 2.16. Let's consider a completed history h' ($h' \in rcomplete_o(h)$) which adds a response event at the pending write operation at x before the crash event. Specifically $h' \triangleq \langle\ inv_{t_2}(\mathtt{W}(x,1)),\ inv_{t_1}(\mathtt{R}(x)),\ resp_{t_2}(\mathtt{W}(ok)),$ $inv_{t_2}(\mathtt{W}(x,3)),\ resp_{t_1}(\mathtt{R}(1)),\ resp_{t_2}(\mathtt{W}(ok))$, **Crash**, $inv_{t_2}(\mathtt{W}(y,1)),\ resp_{t_2}(\mathtt{W}(ok)),\ inv_{t_1}(\mathtt{R}(x)),$ $inv_{t_2}(\mathtt{R}(x)),\ resp_{t_1}(\mathtt{R}(1)),\ resp_{t_2}(\mathtt{R}(3))\ \rangle$

According to the second clause of the Def. 2.4.4 the write operation by t_2 on x ($W(x,3,ok)$) can be placed in the equivalent legal sequential history, no later than the invocation of an operation of the same thread to the same object ($R(x,3)$). This permits the forming of $s(\in \mathcal{S}_o)$, in which after the crash thread t_2 first reads the value written on x before the crash (1), then writes 3 on x, and finally reads the newly written value.

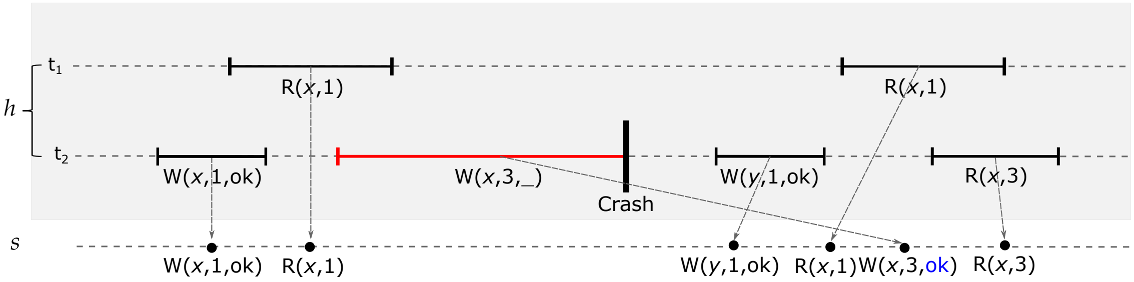


Figure 2.18: A recoverable linearizable history h.

Durable Linearizability: Interestingly, Izraelevitz *et al.* noticed that both the lack of consistency after a crash in recoverable linearizability, as well as the lack of locality in persistent atomicity, derive from the fact that the crash-recovery model that they are based on allows a thread to crash on its own, recover and continue its execution as before. Furthermore, other threads are not affected by the crash. This model doesn't correspond to reality as usually the threads that are executing before a crash do not survive after it. This behavior is captured in the crash-recovery model proposed in [81], where crashes are universal, and threads can operate only in one crash-free region. Informally, a concurrent history h is *durably linearizable* if the history h', which results from removing the crash events of h, is linearizable.

Given a history h, we let $\mathsf{ops}(h)$ denote h restricted to non-crash events. The crash events partition a history into $h = \epsilon_0\mathbf{Crash}_1\epsilon_1\mathbf{Crash}_2...\epsilon_{n-1}\mathbf{Crash}_n\epsilon_n$, such that n is the number of crash events in h, $\mathbf{Crash}_i$ is the ith crash event and $\mathsf{ops}(h) = \epsilon_0\epsilon_1...\epsilon_{n-1}\epsilon_n$, (i.e., h contains no crash events). We call the subhistory ϵ_i the *i-th era* of h. As *sequential* histories do not include crash events, the *well-formedness* condition suggests that every thread identifier appears in at most one era, meaning that thread identifiers cannot be reused after a crash.

Definition 2.4.5 (Durable Linearizability)**.** A well formed history h is *durably linearizable* if $\mathsf{ops}(h)$ is linearizable.

Assuming a universal crash, recoverable linearizability and persistent atomicity are indistinguishable. This is because both conditions allow the pending operations of a thread t to be completed before thread t executes a subsequent to the crash operation. If there is no other operation executed after the crash by the same thread, both conditions allow the matching response to be appended in the end of the history, becoming equivalent to durable linearizability.

Fig. 2.19 illustrates a durable linearizable history h. Specifically $h \triangleq \langle\ inv_{t_2}(\mathtt{W}(x,4))$, $inv_{t_1}(\mathtt{R}(x))$, $resp_{t_1}(\mathtt{R}(4))$, $inv_{t_1}(\mathtt{W}(x,5))$, $resp_{t_1}(\mathtt{W}(ok))$, **Crash** $inv_{t_3}(\mathtt{R}(x))$, $inv_{t_3}(\mathtt{R}(5))\ \rangle$

As seen, no invocation/response event in h has the same thread identifier before and after the system crash. Having h, we can obtain $\mathsf{ops}(h)$ by removing from h the crash event. History $h' \triangleq \langle\ inv_{t_2}(\mathtt{W}(x,4))$, $inv_{t_1}(\mathtt{R}(x))$, $resp_{t_1}(\mathtt{R}(4))$, $inv_{t_1}(\mathtt{W}(x,5))$, $resp_{t_1}(\mathtt{W}(ok))$,

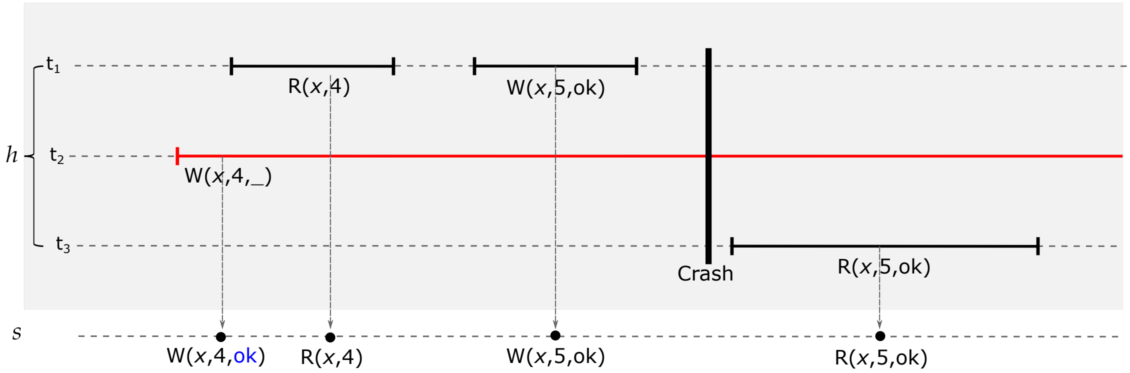


Figure 2.19: A durable linearizable history h.

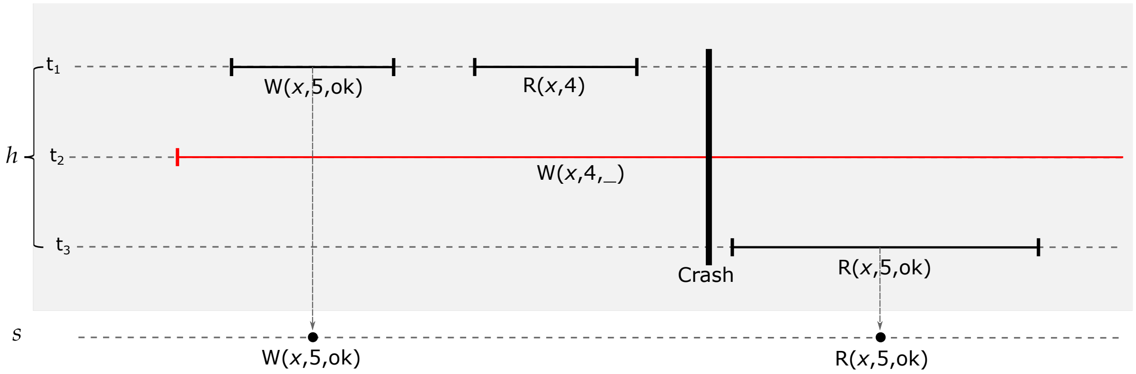


Figure 2.20: A buffered durable linearizable history h.

$inv_{t_3}(\mathtt{R}(x))$, $inv_{t_3}(\mathtt{R}(5))$, $resp_{t_2}(\mathtt{W}(ok))$ $\rangle$ belongs to $complete_o(ops(h))$ and has an equivalent legal sequential history (s) which respect the real-time order of the events of h'.

Buffered Durable Linearizability: *Buffered durable linearizability* is a variation of durable linearizability also proposed in [81]. This condition is more relaxed than durable linearizability in the sense that it allows the removal of committed operations before a crash, as long as the resulting history is legal. The formalization of durable opacity necessitates the definition of a $\prec_h$ *consistent cut* as follows: A $\prec_h$ *consistent cut* is a subhistory p of h for which it holds that if $op_1 \prec_h op_2$ and $op_2 \in p$ then $op_1 \in p$.

Definition 2.4.6 (Buffered Durable Linearizability)**.** A well-formed history h is *buffered durably linearizable* if there exist subhistories $p_0, p_1 ... p_{c-1}$ such that for all $0 \leq i \leq c$, p_i is a $\prec_h$ *consistent cut* of ϵ_i and $p = p_0, p_1 ... p_{c-1}\epsilon_c$ is linearizable.

The history depicted in Fig. 2.20 is buffered durable linearizable, as there exists $p_0 \triangleq \langle$ $inv_{t_1}(\mathtt{W}(x,5)), resp_{t_1}(\mathtt{W}(ok))$ $\rangle$ and $p_1 \triangleq \langle inv_{t_3}(\mathtt{R}(x)), resp_{t_3}(\mathtt{R}(5))$ $\rangle$ that form a linearizable history $p \triangleq \langle$ $inv_{t_1}(\mathtt{W}(x,5)), resp_{t_1}(\mathtt{W}(ok)) inv_{t_3}(\mathtt{R}(x)), resp_{t_3}(\mathtt{R}(5))$ $\rangle$. Unlike durable linearizability, buffered durable linearizability is not a local property. However, buffered durable linearizability is potentially more efficient than durable linearizability, as it requires less flushes.

2.4.3 Recovery and Continuation of Execution

In addition to ensuring memory consistency, persistency algorithms also need to incorporate recovery mechanisms that can identify linearized operations and resume execution securely without compromising memory consistency or performance. The following definitions address how and where a thread should recover in a program.

Detectable execution [60] refers to the program's ability to determine whether a failed operation was linearized before a crash and retrieve its response if it was. Friedman *et al.* [60] present three implementations of the Michael and Scott queue [107], one of which guarantees durable linearizability and detectable execution. This implementation relies on an announcement array [72], which adds an entry for each prospective operation execution. Each entry includes a thread identifier, an operation identifier, a completion flag, and the operation's result. The thread identifier and operation identifier uniquely identify each operation. When a crash occurs, the recovery mechanism inspects the announcement array to determine if an operation was completed before the crash or needs to be re-executed. Ben-Baruh *et al.* [18] demonstrate that a wide range of objects, such as read/write, CAS, and FIFO queue objects, require auxiliary state (e.g., unique identifiers passed as arguments to recoverable operations) for any obstruction-free detectable implementation. Moreover, they investigate the lower bound on space complexity, showing that non-blocking detectable implementations of these objects do not necessarily have unbounded space complexity.

Several detectable algorithms have been proposed in the literature, including those by Ben-Baruh *et al.* [18, 19] and Attiya *et al.* [13]. Ben-David *et al.* [19] demonstrate that for any code consisting only of read, write, and CAS primitives, detectability can be achieved by partitioning the code into *capsules.* Their model assumes individual thread crashes, and each capsule consists of a recoverable CAS operation followed by multiple reads. Essentially, a capsule constitutes a part of the code that guarantees that at the end of it, all the variables are persisted. A capsule can be safely repeated from the beginning any number of times. After a crash, a thread resumes execution from the last executed capsule before the crash.

Attiya *et al.* [13] suggested *nesting-safe recoverable linearizability* (NRL) which deals with the continuation of execution of a client program after a crash takes place. Specifically, it requires the most nested operation to be completed via a recovery mechanism after a crash. The proposed model assumes that threads are crashed individually and that the state of the program, including the stack frame that contains the response value of an operation and the program counter is persistent, thus it is not lost after a crash.

Li *et al.* [100] introduce an alternative approach called *detectable sequential specification* (DSS), which is largely model-agnostic. DSS incorporates detectability within the sequential specification, allowing applications to request detectability on demand while leaving the correct nesting of objects to the application code. To enable detectability, the sequential specification of an object is enhanced with three additional operations per operation π. First, a $pre - \pi$ operation declares that the subsequent operation π is detectable and its outcome should be saved. Then, an $ex - \pi$ operation executes the operation π. Finally, a *resolve* operation is called after a crash to retrieve the result of the most recent operation π for which $pre - \pi$ was invoked before the crash.

Chapter 3

dTML$_{\mathsf{SC}}$ under Strict Persistency

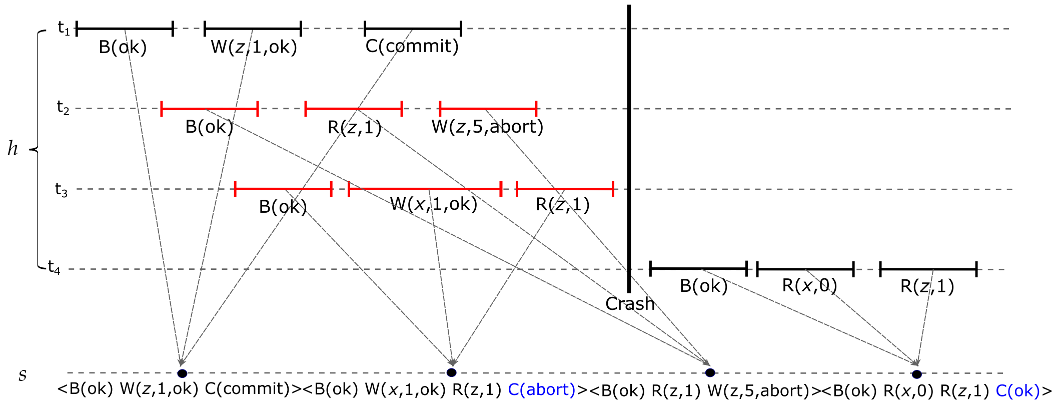
In this chapter, we transfer the principle of durable concurrent correctness to the area of software transactional memory (STM). First, we introduce a novel definition of *durable opacity* extending opacity to handle crashes and recovery in the context of persistent memory (§3.1). Second, we develop operational semantics for *persistent sequential consistency* (§3.2). Third, we use the aforementioned semantics to develop dTML$_{\mathsf{SC}}$, a durably opaque version of an existing STM algorithm, namely the Transactional Mutex Lock (TML) (§3.3). Then, we develop a specification (dTMS2) which is the result of adapting the TMS2 specification, an operational characterization of opacity, to our persistency model (§3.4). We later show that dTMS2 implies durable opacity. We design a proof technique for durable opacity based on refinement. Finally, we apply this proof technique to show that the durable version of TML, dTML$_{\mathsf{SC}}$, is indeed durably opaque (§3.5). The correctness proof is mechanized within Isabelle/HOL.

3.1 Persistency for Transactional Memory

Although numerous correctness conditions for concurrent objects have been established in the context of persistent memory, little emphasis has been placed on developing correctness conditions for software transactional memory in the same context. This section presents *durable opacity* which is a modification of opacity tailored to suit persistent memory. Durable opacity can be thought of as a combination of opacity (as defined in 2.3.10) with durable linearizability (as defined in 2.4.5).

Durable opacity is a correctness condition that is defined over *histories* that record the *invocation* and *response* events of operations executed on the transactional memory like opacity. Unlike opacity, durably opaque histories record system crash events, thus may take the form: $H = \epsilon_0\mathbf{Crash}_1\epsilon_1\mathbf{Crash}_2 ... \epsilon_{n-1}\mathbf{Crash}_n\epsilon_n$, where each ϵ_i is a history (containing no crash events) and $\mathbf{Crash}_i$ is the ith crash event. We call the subhistory ϵ_i the *i-th era* of h.

As with *durable linearizability*, we assume that crash events are universal, meaning that threads that have started their execution before a crash event, are aborted and thus not reused after the crash event. Following Izraelevitz *et al.* [81], for a history h, we let $\mathsf{ops}(h)$ denote h restricted to non-crash events, thus for h, $ops(h) = \epsilon_0\epsilon_1 \dots \epsilon_{n-1}\epsilon_n$,



t1
B(ok)
W(z,1,ok)
C(commit)
t2
B(ok)
R(z,1)
W(z,5,abort)
h
t3
B(ok)
W(x,1,ok)
R(z,1)
t4
B(ok)
R(x,0)
R(z,1)
Crash
s
<B(ok) W(z,1,ok) C(commit)><B(ok) W(x,1,ok) R(z,1) C(abort)><B(ok) R(z,1) W(z,5,abort)><B(ok) R(x,0) R(z,1) C(ok)>

Figure 3.1: A durable opaque history h.

which contains no crash events. We consider here the Def. 2.3.2 of *transactional well-formendness*. Considering that *alternating* histories, as defined in section §2.3.1, do not include crash events, similar to durable linearizability, *durable well-formedness* ensures that transaction/thread identifiers do not appear in more than one era.

Definition 3.1.1 (Durable Opacity)**.** Formally, a history h is *durably opaque* iff it is durably well-formed and $ops(h)$ is opaque. A TM implementation is *opaque* iff each of its histories is opaque.

Figure 3.1 depicts an example of a durably opaque history (h). In order to show that h is durably opaque it is sufficient to show that it is durably well-formed and $ops(h)$ is opaque. This holds as there exists at least one valid sequential history that corresponds to h. For example, the sequential history that matches the ordering: $t_1 \prec t_3 \prec t_2 \prec t_4$ belongs to $\mathcal{S}$ and respects the real-time order of transactions. Notice that t_1, t_2 and t_3 are overlapping so they can be reordered. However since both t_2 and t_3 read the value 1 at address z, t_1 should precede them. Transaction t_2 aborts thus its write at x is not visible to the subsequent transactions. However, the returned value (1) of its read at address z is consistent with the previous writes at z. Transaction t_3 has not been committed before the crash event. Since it is a live transaction, its write is not visible to subsequent transactions. Transaction t_4 reads the initial value (0) at x, as neither t_2 nor t_3 modified it successfully. It also reads the value 1 at z which is successfully written by the preceding transaction t_1. Another legal reordering is : $t_1 \prec t_2 \prec t_3 \prec t_4$.

3.2 Persistent SC Syntax and Semantics

In this section we present a *persistent sequential memory* model. This is a relatively simple model that closely aligns with a developer's intuitive comprehension of non volatile memory.

3.2.1 Persistent SC Language

The syntax of sequential programs is given by the following grammar:

$$
\begin{array}{l}
v, u \in \textsc{Val} \triangleq \mathbb{N} \quad x, y, \ldots \in \textsc{Loc} \quad o \in \textsc{Dobj} \quad f \in \mathrm{F} \quad a, b, \ldots \in \textsc{Reg} \quad t \in \textsc{Tid} \triangleq \mathbb{N} \\
i, j, k, \ldots \in \textsc{Lab} \qquad \hat{a}, \hat{b}, \ldots \in \textsc{AuxVar} \qquad \hat{e} \in \textsc{AuxExp} ::= v \mid \hat{a} \mid \hat{e}{+}\hat{e} \mid \cdots \\
\quad e \in \textsc{Exp} ::= v \mid a \mid e{+}e \mid \cdots \qquad\qquad\qquad B \in \textsc{BExp} ::= \mathsf{true} \mid B \wedge B \mid \cdots \\
\quad \alpha \in \textsc{ASt} ::= \mathbf{skip} \mid a := e \mid a := \mathbf{load}\, x \mid \mathbf{store}\ x\ e \\
\qquad\qquad\qquad \mid r := \mathbf{CAS}\ x\ e\ e \mid \mathbf{flush}\ x \mid o.f \\
\quad ls \in \textsc{LSt} ::= \alpha\ \mathbf{goto}\ j \mid \mathbf{if}\ B\ \mathbf{goto}\ j\ \mathbf{else\ to}\ k \mid \langle \alpha\ \mathbf{goto}\ j, \hat{a} := \hat{e} \rangle \\
\quad \Pi \in \textsc{Prog} \triangleq \textsc{Tid} \times \textsc{Lab} \rightarrow \textsc{LSt} \qquad\qquad pc \in \mathrm{PC} \triangleq \textsc{Tid} \rightarrow \textsc{Lab}
\end{array}
$$

Atomic statements (in ASt) comprise **skip**, assignment, memory reads and writes, and explicit persist instruction. Specifically, $a := e$ evaluates expression e and returns it in register a; $a := \mathbf{load}\, x$ reads from memory location x and returns it in register a; and **store** x e writes the contents of register a to location x. The $r := \mathbf{CAS}\ x\ e_1\ e_2$ denotes 'compare-and-swap' on location x, from the evaluated value of e_1 to the evaluated value of e_2, and sets r to true if the CAS succeeds and to false, otherwise. The $o.f$ denotes a call to an atomic method f of object o. Finally, **flush** x denotes an explicit persist instruction. Even though this model is a simplified description of persistency and is not supported directly by any hardware platform, the **flush** x instruction resembles in strength and functionality Intel's `clflush` x instruction (see §2.1.3).

Formally, we model a program Π as a function mapping each pair (t, i) of thread identifier and label to the *labeled statement* (in LSt) to be executed. A labeled statement may be 1) a plain statement of the form α **goto** j, comprising an atomic statement α to be executed and the label j of the next statement; 2) a conditional statement of the form **if** B **goto** j **else to** k to accommodate branching, which proceeds to label j if B holds and to k, otherwise; or 3) a statement with an auxiliary update $\langle \alpha\ \mathbf{goto}\ j, \hat{a} := \hat{e} \rangle$, which behaves as α **goto** j, but in addition (in the same atomic step) updates the value of the auxiliary variable $\hat{a}$ with the auxiliary expression $\hat{e}$.

We track the control flow within each thread via the *program counter function*, *pc*, which records the program counter of each thread. We assume a designated label, $\iota \in \textsc{Lab}$, representing the *initial label*; i.e. each thread begins execution with $pc(t) = \iota$. Similarly, $\zeta \in \textsc{Lab}$ represents the *final label*. Moreover, if $pc(t) = i$ at the current execution step, then: 1) when $\Pi(t, i) = \alpha\ \mathbf{goto}\ j$ or $\Pi(t, i) = \langle \alpha\ \mathbf{goto}\ j, a := \hat{e} \rangle$, then $pc(t) = j$ at the next step; 2) when $\Pi(t, i) = \mathbf{if}\ B\ \mathbf{goto}\ j\ \mathbf{else\ to}\ k$ at the current step, then if B holds in the current state, then $pc(t) = j$ at the next step; otherwise $pc(t) = k$ at the next step.

3.2.2 Persistent SC Semantics

3.2.2.1 The Persistent SC Machine State

The persistent SC state is modelled by a tuple $\sigma = \langle pc, rec, \mathsf{regs}, \mathsf{vm}, \mathsf{pm}, G \rangle$. $pc : \textsc{Tid} \rightarrow \textsc{Lab}$ maps each thread to the next program counter to be executed. The rec : *bool* component is a flag that indicates when a recovery process is in progress. In the event of a crash, *rec* is set to true to indicate that an implementation-specific recovery process is about to start its execution. Once the recovery process completes, *rec* is reset to false (see below). $\mathbb{T} : \textsc{Tid} \rightarrow \mathsf{regs}$ maps each thread to a record of its registers ($\mathsf{regs} : \textsc{Reg} \rightarrow \textsc{Val}$). The volatile memory is represented as a mapping from locations to values ($\mathsf{vm} : \textsc{Loc} \rightarrow \textsc{Val}$). The persistent memory also constitutes a mapping from locations to values ($\mathsf{pm} : \textsc{Loc} \rightarrow \textsc{Val}$). $G : \textsc{AuxVar} \rightarrow \textsc{Val}$ records the current values of auxiliary variables. We denote the state's σ components as $\sigma.\mathbb{T}$, $\sigma.\mathsf{vm}$, etc. In addition, we use standard function/record update notation (e.g. $\mathsf{vm}' = \mathsf{vm}[x \mapsto v]$ denotes the volatile memory state obtained from vm by mapping the x address to the new value v.) In the initial state (σ_{init}), $pc(t) = \iota$ for all $t \in \textsc{Tid}$, $rec = \mathsf{false}$, $\mathsf{regs}(t)\, r = 0$ for all $t \in \textsc{Tid}$ and $r \in \mathsf{regs}$, $\mathsf{vm}(x) = 0$, $\mathsf{pm}(x) = 0$ for all $x \in \textsc{Loc}$ and $G(a) = 0$ for all $a \in \textsc{AuxVar}$.

3.2.2.2 Modelling Crashes and Recovery

In this model, the CRASH transition simply sets the value of each location (in LOC) in the volatile memory (vm) to its corresponding value in the persistent memory (pm). Furthermore, it resets the registers of each thread. The persistent memory as well as the values of the auxiliary variables recorded in G remain intact after a system crash takes place.

We assume that recovery is executed by a unique system thread *syst* that is different from any program thread. Recovery is only enabled in state σ if $\sigma.rec$ holds. Moreover, we assume a special label $Rec_{pending}$, which corresponds to the label of the first recovery instruction. Upon completion of the recovery procedure, we assume that pc_{syst} is set to $Rec_{complete}$, and that there is a transition from this state to a state in which rec is set to false.

3.2.2.3 Persistent SC Transitions

The transitions of persistent SC are presented in Fig. 3.2 and Fig. 3.3. Note, that, for simplicity and following [80, 90], we conservatively assume that writes persist atomically at the location granularity (representing, e.g. machine words) rather than at the granularity of the width of a cache line. The ASSIGN transition $a := e$, assigns the value of register e to register a. The STORE transition **store** x e, maps the given location x to the value of the given register e in vm. The persistent memory pm remains the same in the pre and post-state. The FLUSH transition maps x in persistent memory to its corresponding value in volatile memory. The CAS instruction has two distinct transitions: one represents a CAS success and the other represents a CAS failure. The CAS-SUCCESS transition requires the value of register e_1 to be equal to the value of address x in the volatile memory. In the post state the value of x in the volatile memory is updated to the value of register e_2 and the register a is assigned true. The CAS-FAIL transition occurs only if the value of x is not equal to the value of register e_1. In such a case, the register α is assigned false while all the other elements of the state remain unchanged. Fig. 3.3 depicts the control flow transitions for a program Π under Persistent SC.

The above operational definitions naturally induce a notion of a execution (or a "run") of persistent SC a certain program Π starting from the initial state σ_{init} as described in section §3.2.2.1. A PROGRAM-CRASH might occur at any point during the execution. A SYSTEM-FLUSH transition corresponds to a write becoming persistent, not because of the execution of a flush instruction, but because its cache line has been evicted to persistent memory, due to the cache replacement mechanism.

(assign)

$$\frac{\alpha = a := e \qquad v = \mathsf{regs}(e) \qquad \mathsf{regs}' = \mathsf{regs}[a \mapsto v]}{\langle \mathsf{regs}, \mathsf{vm}, \mathsf{pm} \rangle \xrightarrow{\alpha} \langle \mathsf{regs}', \mathsf{vm}, \mathsf{pm} \rangle}$$

(store)

$$\frac{\alpha = \mathbf{store}\ x\ e \qquad v = \mathsf{regs}(e) \qquad \mathsf{vm}' = \mathsf{vm}[x \mapsto v]}{\langle \mathsf{regs}, \mathsf{vm}, \mathsf{pm} \rangle \xrightarrow{\alpha} \langle \mathsf{regs}, \mathsf{vm}', \mathsf{pm} \rangle}$$

(load)

$$\frac{\alpha = a := \mathbf{load}\, x \qquad \mathsf{regs}' = \mathsf{regs}[a \mapsto \mathsf{vm}(x)]}{\langle \mathsf{regs}, \mathsf{vm}, \mathsf{pm} \rangle \xrightarrow{\alpha} \langle \mathsf{regs}', \mathsf{vm}, \mathsf{pm} \rangle}$$

(flush)

$$\frac{\alpha = \mathbf{flush}\ x \qquad \mathsf{pm}' = \mathsf{pm}[x \to \mathsf{vm}(x)]}{\langle \mathsf{regs}, \mathsf{vm}, \mathsf{pm} \rangle \xrightarrow{\alpha} \langle \mathsf{regs}', \mathsf{vm}, \mathsf{pm}' \rangle}$$

(CAS-success)

$$\frac{\alpha = a := \mathbf{CAS}\ x\ e_1\ e_2 \qquad v_1 = \mathsf{regs}(e_1) \qquad v_2 = \mathsf{regs}(e_2) \qquad \mathsf{vm}(x) = v_1 \qquad \mathsf{vm}' = \mathsf{vm}[x \mapsto v_2] \qquad \mathsf{regs}' = \mathsf{regs}[a \mapsto \mathsf{true}]}{\langle \mathsf{regs}, \mathsf{vm}, \mathsf{pm} \rangle \xrightarrow{\alpha} \langle \mathsf{regs}', \mathsf{vm}', \mathsf{pm} \rangle}$$

(CAS-fail)

$$\frac{\alpha = a := \mathbf{CAS}\ x\ e_1\ e_2 \qquad v_1 = \mathsf{regs}(e_1) \qquad \mathsf{vm}(x) \neq v_1 \qquad \mathsf{regs}' = \mathsf{regs}[a \mapsto \mathsf{false}]}{\langle \mathsf{regs}, \mathsf{vm}, \mathsf{pm} \rangle \xrightarrow{\alpha} \langle \mathsf{regs}', \mathsf{vm}, \mathsf{pm} \rangle}$$

Figure 3.2: Instruction transitions of Persistent SC for a program Π.

(program-normal)

$$\frac{pc(t) = i \qquad \Pi(t,i) = \alpha\ \mathbf{goto}\ j \qquad \langle \mathbb{T}(t), \mathsf{vm}, \mathsf{pm} \rangle \xrightarrow{\alpha} \langle \mathsf{regs}', \mathsf{vm}', \mathsf{pm}' \rangle \qquad pc' = pc[t \mapsto j] \qquad \mathbb{T}' = \mathbb{T}[t \mapsto \mathsf{regs}']}{\langle pc, \mathbb{T}, \mathsf{false}, \mathsf{vm}, \mathsf{pm}, G \rangle \Rightarrow_{\Pi} \langle pc', \mathbb{T}', \mathsf{false}, \mathsf{vm}', \mathsf{pm}', G \rangle}$$

(program-if)

$$\frac{pc(t) = i \qquad \Pi(t,i) = \mathbf{if}\ B\ \mathbf{goto}\ j\ \mathbf{else\ to}\ k \qquad pc' = pc\left[t \mapsto \begin{cases} j & \mathbb{T}(t)(B) = \mathsf{true} \\ k & \mathbb{T}(t)(B) = \mathsf{false} \end{cases}\right]}{\langle pc, \mathsf{false}, \mathbb{T}, \mathsf{vm}, \mathsf{pm}, G \rangle \Rightarrow_{\Pi} \langle pc', \mathsf{false}, \mathbb{T}, \mathsf{vm}, \mathsf{pm}, G \rangle}$$

(program-ghost)

$$\frac{pc(t) = i \qquad \Pi(t,i) = \langle \alpha\ \mathbf{goto}\ j, \hat{a} := \hat{e} \rangle \qquad \langle \mathbb{T}(t), \mathsf{vm}, \mathsf{pm} \rangle \xrightarrow{\alpha} \langle \mathsf{regs}', \mathsf{vm}', \mathsf{pm}' \rangle \qquad pc' = pc[t \mapsto j] \qquad \mathbb{T}' = \mathbb{T}[t \mapsto \mathsf{regs}'] \qquad G' = G[\hat{a} \mapsto G(\hat{e})]}{\langle pc, \mathsf{false}, \mathbb{T}, \mathsf{vm}, \mathsf{pm}, G \rangle \Rightarrow_{\Pi} \langle pc', \mathsf{false}, \mathbb{T}', \mathsf{vm}', \mathsf{pm}', G' \rangle}$$

(program-crash)

$$\frac{\mathsf{regs}' = (\lambda a.0) \qquad \mathbb{T}' = (\lambda t.\mathsf{regs}') \qquad pc' = pc[syst \mapsto Rec_{pending}]}{\langle pc, \mathsf{false}, \mathbb{T}, \mathsf{vm}, \mathsf{pm}, G \rangle \Rightarrow_{\Pi} \langle pc', \mathsf{true}, \mathbb{T}', \mathsf{pm}, \mathsf{pm}, G \rangle}$$

(system-flush)

$$\frac{\mathsf{pm}' = \mathsf{pm}[x \to \mathsf{vm}(x)]}{\langle pc, b, \mathbb{T}, \mathsf{vm}, \mathsf{pm}, G \rangle \Rightarrow_{\Pi} \langle pc', b, \mathbb{T}, \mathsf{vm}, \mathsf{pm}', G \rangle}$$

Figure 3.3: Control flow transitions of Persistent SC for a program Π.

```
TMBegin
$Bp$ : do $\mathsf{loc}_t$ := load $\mathsf{glb}$;
$B1$ : until $even(\mathsf{loc}_t)$;
$Br$ : return $ok$

TMRead($x$)
$Rp$ : $r_t$ := load $x$;
$R1$ : $c_t$ := load glb;
$R2$ : if $c_t = \mathsf{loc}_t$ then
$Rr$ :     return $r_t$
$Ab$ : else return $abort$

TMCommit
$Cp$ : if $odd(\mathsf{loc}_t)$ then
$C1$ :     $log$.empty();
$C2$ :     ⟨store glb $(\mathsf{loc}_t + 1)$,
            writer := $None$⟩
$Cr$ : return $commit$;
```

```
TMWrite($x, v$)
$Wp$ : if $even(\mathsf{loc}_t)$ then
$W1$ :     $\mathsf{hasWritten}_t$ := CAS glb $\mathsf{loc}_t$ $(\mathsf{loc}_t + 1)$;
$W2$ :     if $\mathsf{hasWritten}_t$ then
$W3$ :         ⟨$x_t := loc_t + 1$, writer := $t$⟩
$Ab$ :     else return $aborted$
$W4$ : if $\neg log$.contains($x$) then
$W5$ :     $c_t$ := load $x$;
$W6$ :     $log$.update($x, c_t$);
$W7$ : store $x$ $v$;
$W8$ : flush $x$;
$Wr$ : return $ok$

TMRecover
$Rec1$ : while $\neg log$.isEmpty()
$Rec2$ :     $c_{syst}$ := $log$.getKey();
$Rec3$ :     store $c_{syst}$ $log$.getVal($c_{syst}$);
$Rec4$ :     flush $c_{syst}$;
$Rec5$ :     $log$.update($c_{syst}, \perp$);
$Rec6$ : store glb 0;
```

Figure 3.4: Durable Transactional Mutex Lock

3.3 Example: Durable Transactional Mutex Lock

We now develop a durably opaque STM: a persistent memory version of the Transactional Mutex Lock (TML) [38].

3.3.1 The dTML$_{\mathsf{SC}}$ Algorithm

In this section, we describe TML and the extensions required for guaranteeing durable opacity under persistent SC. Pseudocode for dTML$_{\mathsf{SC}}$ is given in Fig. 3.4. The lines that constitute our extensions to the original algorithm are highlighted in green colour. All the local variables, apart from the auxiliary ones, are modeled as registers. To distinguish them from global variables, we index the registers with the *id* of the transaction that they belong to. As before, we assume that thread identifiers coincide with the transaction identifiers.

Line numbers are corresponding to label values (in LAB). For the sake of readability, we may use the term program counter ($pc_t, t \in$ TID) to refer to labels. Transactions that are starting (resp. returning successfully from) the execution of a dTML$_{\mathrm{Px86}}$ operation are in a pending (resp. responding) state. The corresponding program counter starts with the starting letter of the pending (resp. responding) operation and ends in p (resp. r) (i.e. Bp for begin pending, Br for begin responding). In case a dTML$_{\mathsf{SC}}$ operation fails, it transitions to Ab, which is an abbreviation for *Aborted*.

3.3.2 The Basic TML Algorithm

In this section, we ignore the lines in green colour and focus on the original algorithm. TML performs writes in an *eager* manner, also known as *direct update*, i.e., it updates shared memory within the write operation itself. This is in contrast to lazy algorithms that store writes locally in a write set, and update shared memory at a later stage, e.g., in the commit operation. Additionally TML adopts a *strict* policy for transactional synchronization: as soon as a transaction attempts to write to a variable, all other transactions running concurrently will be aborted when they will invoke a read or a write operation. To enforce this synchronization policy, TML uses a single global versioned lock, glb, and a local register loc_t that is used for recording a snapshot of glb at the beginning of the transaction t. A writing transaction is in progress iff the value of glb is odd.

A transaction t starts (operation `TMBegin`) by reading glb and storing the read value in the register loc_t (Bp). If the value of glb is odd, another writing transaction is in progress so t does not start. Instead, it reattempts to start by rereading glb.

Operation `TMWrite`(x, v) first checks whether loc_t is even (Wp). If not, t must already be the writing transaction, and hence, it can proceed immediately towards executing the new write on the given address ($W7$). If loc_t is even, it means that the current transaction is not yet a writing transaction, thus it attempts to become a writing transaction by performing a compare-and-swap (**CAS**) operation ($W1$). If this **CAS** succeeds `TMWrite` becomes the writing transaction and increments loc_t ($W3$), making loc_t odd, then proceeds to update x to v ($W7$). In addition, at $W3$, the auxiliary variable writer is set to t. If the **CAS** at $W1$ fails the transaction t aborts.

Operation `TMRead`(x) first reads the value at the given location x and stores it in the register r_t (Rp). At line $R2$, the operation reads the current value of glb. If this value is the same as loc_t, then either

- this transaction is the writing transaction, or
- no other writing transaction has performed any writes since this transaction started.

thus it returns the read value. Otherwise, it aborts (Rr).

A transaction t commits by first checking whether loc_t is odd (Cp). If so, it means that t is a writing transaction (and hence glb is odd), thus it makes glb even by incrementing glb by one. Furthermore, it sets the auxiliary variable writer to *None* ($C2$). If t is a read-only transaction (i.e., loc_t is even), it simply returns *commit*.

3.3.3 Ensuring Durability

Our implementation uses a *durably linearizable* persistent undo log *log* that records the previous values of locations that have been overwritten by incomplete writing transactions. The log is reset to empty when the writing transaction commits. If a crash occurs when an incomplete writing transaction t is in flight, the subsequent recovery operation sets the state to the last consistent state by undoing the writes of t using the undo log. In

Fig. 3.4, we use operations $log.\mathbf{isEmpty}()$, $log.\mathbf{contains}(x)$, $log.\mathbf{getKey}()$, $log.\mathbf{getVal}()$, $log.\mathbf{update}(x, v)$ to stress that these operations are durably linearizable.

Our durable TML algorithm (dTML) makes the following adaptations to TML. Note the the operations build on a model of a crash that resets volatile memory to persistent memory.

1) Within a write operation writing to address x, prior to modifying the value at x, we record the existing address-value pair in *log*, provided that x does not already appear in the undo log (lines $W4 - W6$). After updating the value (which updates the value of x in the volatile store), the update is flushed to persistent memory prior to the write operation returning (line $W8$).
2) We introduce a recovery operation that checks for a non-empty log and transfer the logged values to persistent memory, undoing any writes that have been completed (but not committed) before the crash occurred. Since a crash could occur during recovery, we transfer values from the undo log to persistent memory one at a time.
3) In the commit operation, we note that we distinguish a committing transaction as one with an odd value for loc_t. For a writing transaction, the log must be cleared by setting it to the empty log (line $C1$). Note that this is the point at which a writing transaction has definitely been committed since any subsequent crash and recovery would no longer undo the writes of this transaction.

3.4 Operational Specification for Durable Opacity

In this section, we first model dTML$_{\mathsf{SC}}$ as an Input/Output Automaton. We then develop an operational specification dTMS2 by adapting TMS2, which we later show that it implies *durable opacity*.

3.4.1 Background: Input/Output Automata: IOA

We use Input/Output Automata (IOA) [104] to model both the implementation, dTML$_{\mathsf{SC}}$, and the specification, dTMS2.

Definition 3.4.1 (Input/Output Automaton (IOA))**.** An *Input/Output Automaton (IOA)* is a labeled transition system A with a set of *states* $states(A)$, a set of *actions* $acts(A)$, a set of *start states* $start(A) \subseteq states(A)$, and a *transition relation* $trans(A) \subseteq states(A) \times acts(A) \times states(A)$ (so that the actions label the transitions).

The set $acts(A)$ is partitioned into input actions $input(A)$, output actions $output(A)$ and internal actions $internal(A)$. The internal actions represent events of the system that are not visible to the external environment. The input and output actions are externally visible, representing the automaton's interactions with its environment. Thus, we define the set of *external actions*, $external(A) = input(A) \cup output(A)$. We write $s \xrightarrow{a}_A s'$ iff $(s, a, s') \in trans(A)$.

An *execution* of an IOA A is a sequence $\sigma = s_0 a_0 s_1 a_1 s_2 \ldots s_n a_n s_{n+1}$ of alternating states and actions, such that $s_0 \in start(A)$ and for all states s_i, $s_i \xrightarrow{a_i}_A s_{i+1}$. A *reachable* state of A is a state appearing in an execution of A. An *invariant* of A is any superset of the reachable states of A (equivalently, any predicate satisfied by all reachable states of A). A *trace* of A is any sequence of (external) actions obtained by projecting the external actions of any execution of A. The set of traces of A, denoted $traces(A)$, represents A's externally visible behavior.

3.4.2 IOA for dTML$_{\mathsf{SC}}$

We now provide the IOA model of dTML$_{\text{Px86}}$ (Fig. 3.4) over the persistent SC state. Specifically, the dTML$_{\mathsf{SC}}$ state comprises global (shared) variables $\mathsf{glb} \in \textsc{Loc}$; the volatile memory store $\mathsf{vm} \in \textsc{Loc} \to \textsc{Val}$; and the persistent memory store $\mathsf{pm} \in \textsc{Loc} \to \textsc{Val}$. We also use the following transaction-local variables: the program counter $\mathsf{pc}_t : \textsc{Tid} \to \textsc{Lab}$, $\mathsf{loc}_t \in \textsc{Reg}$, the input address $x_t \in \textsc{Tid} \to \textsc{Val}$ and the input value $v_t \in \textsc{Tid} \to \textsc{Val}$. We also make use of an auxiliary variable writer the value of which is either the transaction id of the current writing transaction (if one exists), or $None$ (if no writing transaction is currently running).

The *durably linearizable* log, *log*, is modeled as a partial mapping from location to values ($log \in \textsc{Loc} \nrightarrow \textsc{Val}$), where $\nrightarrow$ denotes a partial function (modelling *log* in persistent memory). We assume the following operations

- *log*.**isEmpty**(): that returns true whenever the *log* is empty (i.e., all elements are mapped to $\perp$)
- *log*.**contains**(x): that returns true whenever the log contains x (i.e., x is not mapped to $\perp$)
- *log*.**update**(x, v): that updates the logged location x to value v
- *log*.**getKey**(): that non-deterministically returns a location whose value is not $\perp$, and
- *log*.**getVal**(x): that returns the value of x in *log*.

The log is stored in the G state component of the state (§3.2.2.1) and updated according to SC semantics.

Execution of the program is modeled by defining an IOA transition for each atomic step of Fig. 3.4 using the values of pc_t (for transaction t) to model control flow. When t is in flight, but not executing any operation we have $pc_t = Ready$. Similarly, $pc_t = Aborted$ and $pc_t = Committed$ iff t has aborted or committed, respectively. Otherwise, pc_t is a line number corresponding to the instruction of the operation t is executing. Each action that starts a new operation or returns from a completed operation is an external action. The crash action is also external. All other actions (including system flush and recovery) are internal actions. Notice that all the transitions of Fig. 3.4 apart from the transition to *Aborted* correspond to internal actions.

To model system behaviors (crash, system flush, and recovery), we reserve a special transaction id *syst*. A crash and system flush is always enabled, and hence can always be selected for execution. Recovery steps are enabled after a crash has taken place ($rec = \mathsf{true}$) and are only executed by *syst*. The effect of a flush is to copy the value of the address being flushed from vm to pm. Note that a flush can also be executed at specific program locations. In Fig. 3.4, a flush of x occurs at lines $W8$ and $Rec4$. The effect of a crash is to perform the following:

- set the volatile store to the persistent store (since the volatile store is lost),
- set the program counters of all *in-flight transactions* (i.e., transactions that have started but not yet completed) to *Aborted* to ensure that these transaction identifiers are not reused after the system is rebooted, and
- set the label of *syst* to $Rec1$ to model that a recovery is now in progress.

In our model, it is possible for a system to crash during recovery. However, no new transaction may start until after the recovery process has been completed.

3.4.2.1 dTML$_{\mathsf{SC}}$ IOA Invariant ()

In order to prove that the dTML$_{\mathsf{SC}}$ algorithm is durably opaque, we use a certain invariant of the dTML$_{\mathsf{SC}}$ model. The invariant is given as a selection of preconditions, where each precondition describes the pre-state of a transition of the dTML$_{\mathrm{Px86}}$ IOA. This invariant is similar to the corresponding invariants used in the proof of the original TML algorithm for the conventional volatile RAM model proposed in [44].

Briefly, our invariant keeps track of the parity of loc_t and indicates whether loc_t is equal to glb or less than glb. Furthermore, it implies that there is at most one writing transaction, and there is no such transaction when glb is even. It also constrains the possible differences between volatile and persistent memory: volatile and persistent memory are identical except for any location that has been written by a writer or by the recovery procedure but not yet flushed.

Our invariant has been verified in Isabelle/HOL and it is checked for local correctness and stability.

3.4.3 IOA for dTMS2

In this section, we describe the dTMS2 specification, an operational model that ensures durable opacity, which is based on TMS2 [50]. TMS2 itself has been shown to strictly imply opacity [99], and hence has been widely used as an intermediate specification in the verification of transactional memory implementations [10, 11, 46, 47].

We let $f \oplus g$ denote functional override of f by g, e.g., $f \oplus \{x \mapsto u, y \mapsto v\} = \lambda k.\ \mathbf{if}\ k = x\ \mathbf{then}\ u\ \mathbf{elseif}\ k = y\ \mathbf{then}\ v\ \mathbf{else}\ f(k)$.

Formally, dTMS2 is specified by the IOA in Fig. 3.5, which describes the required ordering constraints, memory semantics, and prefix properties. Memory is modelled by a function of type $\textsc{Loc} \to \textsc{Val}$. A key feature of dTMS2 (like TMS2) is that it keeps track of a *sequence* of memory states, one for each committed writing transaction. This makes it simpler to determine whether reads are consistent with previously committed write operations. Each committing transaction containing at least one write adds a new memory

State variables:

$L : \mathit{seq}(\textsc{Loc} \to \textsc{Val})$, initially satisfying $\mathbf{dom}(L) = \{0\}$ and $\mathit{init}(L(0))$
$\mathsf{beginIdx}_t : \mathbb{N}$ for each $t \in \textsc{Tid}$, unconstrained initially
$\mathsf{rdSet}_t : \textsc{Loc} \nrightarrow \textsc{Val}$, initially empty for all $t \in \textsc{Tid}$
$\mathsf{wrSet}_t : \textsc{Loc} \nrightarrow \textsc{Val}$, initially empty for all $t \in \textsc{Tid}$
$\mathsf{pc}_t = PCVal$ for each $t \in \textsc{Tid}$, initially $\mathsf{pc}_t \in NotStarted$ for all $t \in \textsc{Tid}$.

Transition relation:

```
$inv_t(\texttt{TMBegin})$
pre : $\mathsf{pc}_t = NotStarted \wedge t \neq syst$
eff : $\mathsf{pc}_t := BeginPending$;

$inv_t(\texttt{TMRead}(x))$
pre : $\mathsf{pc}_t = Ready \wedge t \neq syst$
eff : $\mathsf{pc}_t := ReadPending(x)$;

$inv_t(\texttt{TMWrite}(x)(v))$
pre : $\mathsf{pc}_t = Ready \wedge t \neq syst$
eff : $\mathsf{pc}_t := WritePending(x, v)$;

$inv_t(\texttt{TMCommit})$
pre : $\mathsf{pc}_t = ready \wedge t \neq syst$
eff : $\mathsf{pc}_t := CommitPending$;

$inv_t(\texttt{TMCancel})$
pre : $\mathsf{pc}_t = Ready \wedge t \neq syst$
eff : $\mathsf{pc}_t := CancelPending$;

$\texttt{doBegin}_t$
pre : $\mathsf{pc}_t = BeginPending \wedge t \neq syst$
eff : $\mathsf{pc}_t = BeginResponding$;
      $\mathsf{beginIdx}_t = |L| - 1$;

$\texttt{doCommit}_t$
pre : $\mathsf{pc}_t = CommitPending \wedge t \neq syst \wedge$
      $((\mathsf{wrSet}_t = \emptyset \wedge \exists n.\ validIdx(t,\ n)) \vee \mathsf{rdSet}_t \subseteq last(L))$
eff : $\mathsf{pc}_t = CommitResponding$;
      if $\mathsf{wrSet}_t \neq \emptyset$ then
         $L := L \mathbin{+\!\!+} \langle last(L) \oplus \mathsf{wrSet}_t \rangle$;

TMCrashRecovery
pre : $True$
eff : $\lambda t \in \textsc{Tid}.$ if $\mathsf{pc}_t \notin \{NotStarted,$
         $Aborted, Committed\}$ then $\mathsf{pc}_t := Aborted$;
      $L := \langle last(L) \rangle$;

$resp_t(\texttt{TMBegin}(ok))$
pre : $\mathsf{pc}_t = BeginResponding \wedge t \neq syst$
eff : $\mathsf{pc}_t := Ready$;

$resp_t(\texttt{TMRead}(v))$
pre : $\mathsf{pc}_t = ReadResponding(v) \wedge t \neq syst$
eff : $\mathsf{pc}_t := Ready$;

$resp_t(\texttt{TMWrite}(ok))$
pre : $\mathsf{pc}_t = WriteResponding \wedge t \neq syst$
eff : $\mathsf{pc}_t := Ready$;

$resp_t(\texttt{TMCommit}(commit))$
pre : $\mathsf{pc}_t = CommitResponding \wedge t \neq syst$
eff : $\mathsf{pc}_t := Committed$;

$resp_t(abort)$
pre : $\mathsf{pc}_t \notin \{NotStarted, Ready, Committed,$
            $CommitResponding, Aborted\}$
eff : $\mathsf{pc}_t := Aborted$;

$\texttt{doWrite}_t(x, v)$
pre : $\mathsf{pc}_t = WritePending(x, v) \wedge t \neq syst$
eff : $\mathsf{pc}_t := WriteResponding$;
      $\mathsf{wrSet}_t := \mathsf{wrSet}_t \oplus \{x \to v\}$;

$\texttt{doRead}_t(x)$
pre : $\mathsf{pc}_t = ReadPending(x) \wedge t \neq syst \wedge$
      $(x \in \mathbf{dom}(\mathsf{wrSet}_t) \vee validIdx(t,\ n))$
eff : if $x \in \mathbf{dom}(\mathsf{wrSet}_t)$ then
         $v := \mathsf{wrSet}_t(x)$;
         $pc_t := ReadResponding(v)$;
      else
         $v := L(n)(x)$;
         $\mathsf{rdSet}_t := \mathsf{rdSet}_t \oplus \{x \to v\}$;
         $pc_t := ReadResponding(v)$;
```

where
$validIdx(t, n) = \mathsf{beginIdx}_t \leq n < |L| \wedge \mathsf{rdSet}_t \subseteq L[n]$

Figure 3.5: The state space and transition relation of dTMS2, which extends TMS2 with a crash event

version to the end of the memory sequence. However, unlike TMS2, following [45], the memory state is considered to be the persistent memory state. Interestingly, the volatile memory state need not be modelled.

The state space of dTMS2 has several components. The first, L, is the sequence of *memory* states. For each transaction t there is a program counter variable pc_t, which ranges over a set of *program counter values*, which are used to ensure that each transaction is well-formed, and to ensure that each transactional operation takes effect between its invocation and response. There is also a *begin index* variable $\mathsf{beginIdx}_t$, that is set to the index of the most recent memory version when the transaction begins. This variable is critical for ensuring the real-time ordering property between transactions. Finally, there is a *read set*, rdSet_t, and a *write set*, wrSet_t, which record the values that a transaction t has read and written during its execution, respectively.

The read set is used to determine whether the values that have been read by the transaction are consistent with the same version of memory (using $validIdx$). The write set, on the other hand, is required because writes in dTMS2 are modeled using *deferred update* semantics: writes are recorded in the transaction's write set, but are not published to any shared state until the transaction is committed.

The *crash* action models both a crash and a recovery. We require that it is executed by the system thread $syst$. It sets the program counter of every in-flight transaction to *aborted*, which prevents these transactions from performing any further actions in the era following the crash (for the generated history). Note that since transaction identifiers are not reused, the program counters of completed transactions need not be set to any special value (e.g., *crashed*) as with durable linearizability. Moreover, after restarting, it must not be possible for any new transaction to interact with memory states prior to the crash. We therefore reset the memory sequence to be a singleton sequence containing the last memory state prior to the crash.

The following theorem ensures that dTMS2 can be used as an intermediate specification in our proof method.

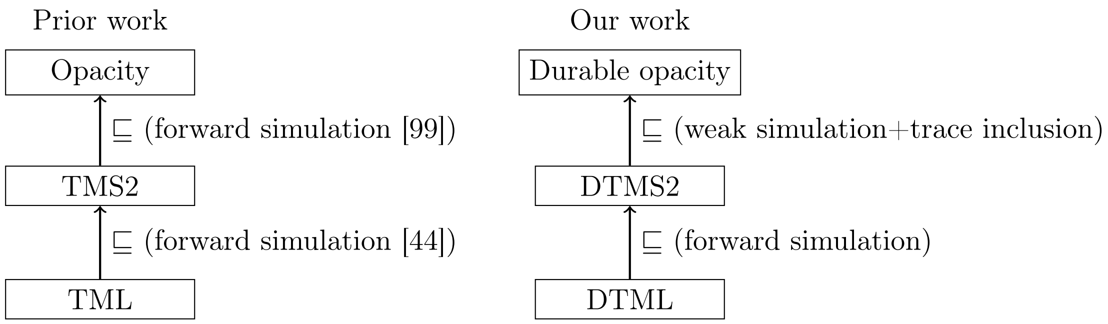
Theorem 3.4.1 (). *Each trace of dTMS2 is durably opaque.*

3.5 Proving Durable Opacity of dTML$_{\mathsf{SC}}$ ()

Previous works [10, 11, 47, 51] have considered proofs of opacity using the operational TMS2 specification [50], which has been shown to guarantee opacity [99]. The proofs show the refinement of the implementation against the TMS2 specification using either forward or backward simulation. For durable opacity, we use a similar proof strategy.

In the following sections, we will begin by presenting Theorem 3.4.1, and subsequently, we will demonstrate a refinement between dTML$_{\mathsf{SC}}$ and dTMS2. Both proofs rely on the simulation relation technique. Specifically, to prove the soundness of dTMS2, we will utilize an inductive technique akin to forward simulation, which we will refer to as *weak simulation*. After establishing a weak simulation between TMS2 and dTMS2, we will then construct a trace inclusion proof to show Theorem 3.4.1. To show that dTML$_{\mathsf{SC}}$ refines dTMS2 and is thus durably opaque, we will illustrate a *forward simulation* between dTML$_{\mathsf{SC}}$ and dTMS2.

To demonstrate that dTMS2 implies durable opacity, we will rely on the result presented by Lesani *et al.* [99], which establishes that TMS2 guarantees opacity. Our next proof regarding dTML$_{\mathsf{SC}}$ refining dTMS2 closely follows the proof technique used by Derick *et al.* [44] for showing that TML refines TMS2.


Prior work
Opacity
⊑ (forward simulation [99])
TMS2
⊑ (forward simulation [44])
TML
Our work
Durable opacity
⊑ (weak simulation+trace inclusion)
DTMS2
⊑ (forward simulation)
DTML

3.5.1 Background: Refinement and Simulation

We now present the definitions of *forward simulation* and *weak simulation* that are used in the subsequent proofs.

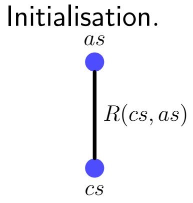
For automata C and A, we say that C is a *refinement* of A iff $traces(C) \subseteq traces(A)$. We can show that C is a refinement of A by proving the existence of a *forward simulation*, which enables one to check step correspondence between the transitions of C and those of A. The definition of forward simulation we use is adapted from that of Lynch and Vaandrager [103].

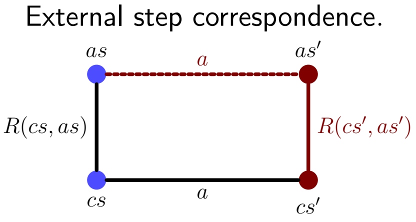
Definition 3.5.1 (FORWARD SIMULATION)**.** A *forward simulation* from a concrete IOA C to an abstract IOA A is a relation $R \subseteq states(C) \times states(A)$ such that each of the following holds.

Initialisation: For each $cs \in start(C)$ there is some $as \in start(A)$ such that $R(cs, as)$.

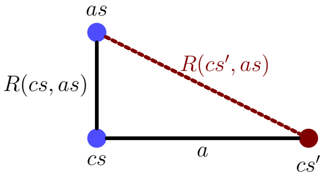
External step correspondence: For each $cs \in reach(C)$, $as \in reach(A)$, $a \in external(C)$, and $cs' \in states(C)$, if $R(cs, as)$ and cs $\xrightarrow{a}_C cs'$ then there is some $as' \in states(A)$ such that $R(cs', as')$ and as $\xrightarrow{a}_A as'$.

Internal step correspondence: For each $cs \in reach(C)$, $as \in reach(A)$, $a \in internal(C)$ and $cs' \in states(C)$, if $R(cs, as)$ and $cs \xrightarrow{a}_C cs'$ then either $R(cs', as)$ or there is some $as' \in states(A)$ and $a' \in internal(A)$ such that $as \xrightarrow{a'}_A as'$ and $R(cs', as')$.

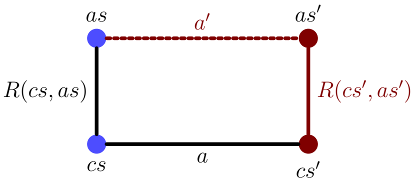




Internal step correspondence (stuttering).



Internal step correspondence (non-stuttering).



Forward simulation is *sound* in the sense that if there is a forward simulation between A and C, then C refines A [103, 109].

Weak simulation allows some external actions of the concrete automaton C to be treated as internal actions.

Definition 3.5.2 (WEAK SIMULATION dTMS2-dTML$_{\mathsf{SC}}$)**.** A *weak simulation* from a concrete IOA of dTMS2 (C) to the abstract IOA A of TMS2 (A) is a relation $R \subseteq$ $states(C) \times states(A)$ such that each of the following holds.

Initialisation. $\forall cs \in start(C).\ \exists as \in start(A).\ R(cs, as)$

External step correspondence: For each $cs \in reach(C)$, $as \in reach(A)$, $a \in external(C)$ $\setminus$ $\{\texttt{TMCrashRecovery}\}$, and $cs' \in states(C)$, if $R(cs, as)$ and $cs \xrightarrow{a}_C cs'$ then there is some $as' \in states(A)$ such that $R(cs', as')$ and $as \xrightarrow{a}_A as'$.

Internal step correspondence: For each $cs \in reach(C)$, $as \in reach(A)$, $a \in internal(C) \cup$ $\{\texttt{TMCrashRecovery}\}$ and $cs' \in states(C)$, if $R(cs, as)$ and cs $\xrightarrow{a}_C$ cs' then either $R(cs', as)$ or there is some $as' \in states(A)$ and $a' \in internal(A)$ such that as$\xrightarrow{a'}_A$ as' and $R(cs', as')$.

The only difference between this definition and the standard notion in Definition 3.5.1 is that we treat crash events as internal events.

Both the weak simulation established between dTMS2 − TMS2 and the forward simulation established between dTML$_{\mathsf{SC}}$ − TMS2 obtain the following form:

$R(cs, as) = globalR(cs, as) \wedge \forall t \in \textsc{Tid}.\ txnR(cs, as, t)$

The simulation relation (R) in both cases is split into two relations: a global relation, *globalR*, and a transactional relation *txnR*. The global relation describes how the shared states of the two automata are related, while the transactional relation specifies the relation between the state of each transaction in the concrete and abstract transition system.

3.5.2 Soundness of dTMS2

In Lemma 2, we show that if $h \in traces(\text{dTMS2})$ then $\mathsf{ops}(h) \in traces(\text{TMS2})$. It has already been shown [99] that every trace of TMS2 is opaque. Putting these two facts together, we have that for every trace $h \in traces(\text{dTMS2})$, $\mathsf{ops}(h) \in traces(\text{TMS2})$ is opaque, and so h is durably opaque.

TMS2 is fully described in [50]. We do not present the automaton explicitly here, but it is *precisely* the dTMS2 automaton with the crash action removed.

We prove Lemma 2 by exhibiting a relation $R \subseteq \mathit{states}(\text{dTMS2}) \times \mathit{states}(\text{TMS2})$, which satisfy the properties given in Def. 3.5.1 (weak simulation).

Lemma 1. *For any relation $R \subseteq \mathit{states}(dTMS2) \times \mathit{states}(TMS2)$, if R is a weak simulation from dTMS2 to TMS2 then for every $h \in \mathit{traces}(dTMS2)$, $\mathit{ops}(h) \in TMS2$.*

Proof. The proof is a simple induction on the length of the executions of dTMS2. More specifically, for each execution e of dTMS2, we inductively construct an execution e' of TMS2 such that $\mathit{ops}(\mathit{trace}(e)) = \mathit{trace}(e')$ which is sufficient. This construction is entirely standard, and ensures at each step that the final states of each execution are related by R. This guarantees (given the definition of weak simulation) that we can always extend e' appropriately. □

We turn now to our main lemma.

Lemma 2. *For every trace $h \in \mathit{traces}(dTMS2)$, $\mathit{ops}(h) \in \mathit{traces}(TMS2)$.*

Proof. By Lemma 1, it is enough to exhibit a weak simulation. We define our weak simulation $R \subseteq \mathit{states}(\text{dTMS2}) \times \mathit{states}(\text{TMS2})$ as follows (we explain the components of this relation shortly):

$$(cs, as) \in \mathit{simR} \iff \exists i \in \mathbb{N}.(cs, as) \in \mathit{globalR}(i) \wedge \forall t.(cs, as) \in \mathit{txnR}(i, t) \tag{3.1}$$

Recall that crash events in dTMS2 cause dTMS2's memory sequence to be shortened to a length 1 sequence. There is no corresponding event in TMS2. Thus, the primary difficulty in our proof is relating the sequence of memories in states of dTMS2 with those of TMS2. The existentially quantified index i in Equation 3.1 allows us to do this. Informally, if $(cs, as) \in R$ then the memory sequence in c is equal to the suffix of the memory sequence in a beginning at index i.

The global relation *globalR*, indexed by i, is the conjunction of the following:

$$|cs.M| + i = |as.L| \tag{3.2}$$

$$\forall n < |cs.M|.cs.M[n] = as.L[n + i] \tag{3.3}$$

The transactional relation *txnR* , indexed by i, and transaction index t is the conjunction of the following:

$$cs.\mathsf{pc}_t \notin \{NotStarted, Committed, Aborted\} \implies cs.\mathsf{beginIdx}_t + i = as.\mathsf{beginIdx}_t \tag{3.4}$$

and

$$cs.\mathsf{pc}_t \neq \mathit{Aborted} \implies cs.\mathsf{pc}_t = as.\mathsf{pc}_t \tag{3.5}$$

$$cs.\mathsf{pc}_t \neq \mathit{Aborted} \implies cs.\mathsf{rdSet}_t = as.\mathsf{rdSet}_t \tag{3.6}$$

$$cs.\mathsf{pc}_t \neq \mathit{Aborted} \implies cs.\mathsf{wrSet}_t = as.\mathsf{wrSet}_t \tag{3.7}$$

We now prove that R is a weak simulation.

3.5.2.1 Initialisation

Initially, we let $i = 0$. Because initially $cs.M = as.L = [m]$ where m is the initial memory state, we have

$$\begin{aligned} |cs.M| + i &= |cs.M| + 0 \\ &= |as.L| \end{aligned}$$

and

$$\begin{aligned} \forall n < |cs.M|. cs.M[n] &= as.L[n + 0] \\ &= as.L[n] \end{aligned}$$

and so for initial states cs, as, we have $(cs, as) \in \mathit{globalR}(0)$. Also, we have $(cs, as) \in \mathit{txnR}(t, 0)$ as required.

3.5.2.2 External Step Correspondence

There are **eight** cases to consider. We directly address two. The other cases are very similar. (Note that we do not treat the `TMCrashRecovery` action as external in this weak simulation.)

$\underline{\mathit{inv}_t(\texttt{TMBegin})}$ **case:**

Let $a = \mathit{inv}_t(\texttt{TMBegin})$ for some thread t and let $cs \xrightarrow{a}_C cs'$ be a transition of dTMS2. Let as be an abstract state such that $(cs, as) \in R$, and let i be the index that witnesses the existential quantification of R. We first show that the precondition of $\mathit{inv}_t(\texttt{TMBegin})$ is satisfied by as. Note that $cs.\mathsf{pc}_t = \mathit{NotStarted}$, and thus (by Equation 3.5), we have $as.\mathsf{pc}_t = \mathit{NotStarted}$, which is sufficient. Because as satisfies a's precondition, we can let as' be the unique state satisfying $as \xrightarrow{a}_A as'$. It remains to show that $(cs', as') \in R$. First observe that $(cs', as') \in \mathit{globalR}(i)$, because $cs'.M = cs.M$, $as'.L = as.L$ and $(cs, as) \in \mathit{globalR}(i)$. Furthermore, note that

$$\begin{aligned} as'.\mathsf{beginIdx}_t &= |as.L| && \text{Transition relation of TMS2} \\ &= |cs.M| + i && \text{Equation 3.2} \\ &= cs'.\mathsf{beginIdx}_t + i && \text{Transition relation of dTMS2} \end{aligned}$$

as required for Equation 3.4. It is easy to check the other conditions that $(cs', as') \in \mathit{txnR}(t, i)$.

$\underline{\mathit{inv}_t(\texttt{TMCommit})}$ **case:**

Let $a = \mathit{inv}_t(\texttt{TMCommit})$ for some thread t and let $cs \xrightarrow{a}_C cs'$ be a transition of dTMS2. Let as be an abstract state such that $(cs, as) \in R$, and let i be the index that witnesses the existential quantification of R. We first show that the precondition of $\mathit{inv}_t(\texttt{TMCommit})$ is satisfied by as. Note that $cs.\mathsf{pc}_t = \mathit{Ready}$, and thus (by Equation 3.5), we have $as.\mathsf{pc}_t = \mathit{Ready}$, which is sufficient. Because as satisfies a's precondition, we can let as' be the unique state satisfying $as \xrightarrow{a}_A as'$. It remains to show that $(cs', as') \in R$. First observe that $(cs', as') \in \mathit{globalR}(i)$, because $cs'.M = cs.M$, $as'.L = as.L$ and $(cs, as) \in \mathit{globalR}(i)$. Furthermore, the only local variables that change are $cs'.\mathsf{pc}_t$ and $as'.\mathsf{pc}_t$ so $(cs', as') \in \mathit{txnR}(t, i)$ is essentially immediate from $(cs, as) \in \mathit{txnR}(t, i)$.

3.5.2.3 Internal Step Correspondence

There are several cases to consider. We directly address two. The other cases are very similar. We first address the *Crash* action.

`TMCrashRecovery` **case:**

Let $a = \texttt{TMCrashRecovery}$ for some thread t and let $cs \xrightarrow{a}_C cs'$ be a transition of dTMS2. Let as be an abstract state such that $(cs, as) \in R$, and let i be the index that witnesses the existential quantification of R. We must show that $(cs', as) \in R$. It is sufficient to prove that

$$(cs', as) \in globalR(|as.L| - 1) \land \forall t.(cs', as) \in txnR(t, |as.L| - 1)$$

So we let $i' = |as.L| - 1$. Note that $cs'.M = [last(cs.M)]$. Thus

$$\begin{aligned} |as.L| &= 1 + |as.L| - 1 \\ &= |cs'.M| + |as.L| - 1 \end{aligned}$$

as required for Equation 3.2. Also, if $n < |cs'.M|$ then $n = 0$, and

$$\begin{aligned} cs'.M[0] &= last(cs.M) \\ &= as.M[|cs.M| - 1 + i] && \text{Equation 3.3} \\ &= as.L[|as.L| - 1] && \text{Equation 3.2} \end{aligned}$$

as required for Equation 3.3. To prove the transactional relation, note that for all t, $cs'.\mathsf{pc}_t \in \{NotStarted, Committed, Aborted\}$ so there is nothing to prove for Equation 3.5. Furthermore, if $cs'.\mathsf{pc}_t = NotStarted$ then $cs.\mathsf{pc}_t = NotStarted$ and therefore the other properties of $txnRel(t, |as.L| - 1)$ are straightforwardly maintained.

$\texttt{doCommit}_t \land \mathbf{dom}(\mathsf{wrSet}_t) \neq \emptyset$ **case:**

Now, let $a = \texttt{doCommit}_t$, $\mathbf{dom}(\mathsf{wrSet}_t) \neq \emptyset$ for some transaction t and let $cs \xrightarrow{a}_C cs'$ be a transition of dTMS2. Let as be an abstract state such that $(cs, as) \in R$, and let i be the index that witnesses the existential quantification of R. In this case, we show that the dTMS2 transition simulates the $\texttt{doCommit}_t \land \mathbf{dom}(\mathsf{wrSet}_t) \neq \emptyset$ transition in TMS2. As usual, we show that the precondition holds in as. Again, Equation 3.5 is enough to prove that the program counterpart of the precondition holds. We must also show that if $cs.\mathsf{rdSet}_t$ is consistent with respect to $last(cs.M)$ it is also consistent with respect to $last(as.L)$. But

$$\begin{aligned} last(cs.M) &= cs.M[|cs.M| - 1] \\ &= as.L[|cs.M| - 1 + i] \\ &= as.L[|as.M| - 1] \\ &= last(as.L) \end{aligned}$$

which is sufficient. Clearly, because $cs.\mathsf{wrSet}_t = as.\mathsf{wrSet}_t$, the nonemptiness of $cs.\mathsf{wrSet}_t$ implies the nonemptiness of $as.\mathsf{wrSet}_t$. Because, as satisfies the precondition of a, we let

as' be the unique state satisfying $as \xrightarrow{a}_A as'$. It remains to show that $(cs', as') \in R$. To do so, we let i be the index witnessing this fact, so that we prove

$$(cs', as) \in globalR(i) \wedge \forall t.(cs', as) \in txnR(t, i)$$

First, observe that

$$\begin{aligned} |as'.L| &= |as.L| + 1 \\ &= |cs.M| + i + 1 \\ &= |cs'.M| + i \end{aligned}$$

as required for Equation 3.2. It is straightforward to see that 3.3 is preserved. The only local variables that change are $cs'.\mathsf{pc}_t$ and $as'.\mathsf{pc}_t$, which are both equal to $CommitPending$, so $(cs', as') \in txnR(t, i)$.

Similar arguments prove that the internal step correspondence condition is met for the other actions. □

We now show that dTMS2 is sound (Theorem 3.4.1).

Proof. Let $h \in traces(\text{dTMS2})$. By Lemma 2, $ops(h) \in traces(\text{TMS2})$ and so $ops(h)$ is opaque [99]. Now, by Definition. 3.1.1, h is durably opaque.

□

3.5.3 Durable Opacity of dTML$_{\mathsf{SC}}$ ()

We now describe the simulation relation used in the Isabelle proof.

3.5.3.1 Global Relation of the dTML$_{\mathsf{SC}}$-dTMS2 Simulation Relation

We first describe $globalR$, which assumes the following auxiliary definitions where cs is the concrete state (of dTML$_{\mathsf{SC}}$) and as is the abstract state (of dTMS2). Our simulation relation assumes the following auxiliary definitions, where cs is the concrete state and as is the abstract state. We define $intHalf(n) \triangleq \lfloor \frac{n}{2} \rfloor$, which returns the integer part of n divided by 2. These definitions are used to compensate for the fact that the commit of a writing transaction in the dTML$_{\mathsf{SC}}$ algorithm takes effect (i.e., linearizes) at line $C1$ when the log is set to empty.

$$\begin{aligned} writes(cs, as) &\triangleq \textbf{if } cs.\mathsf{writer} = t \wedge pc_t \neq C2 \textbf{ then } as.\mathsf{wrSet}_t \textbf{ else } \emptyset \\ logicalGlb(cs) &\triangleq \textbf{if } cs.\mathsf{writer} = t \wedge pc_t = C2 \textbf{ then } cs.\mathsf{glb} + 1 \textbf{ else } cs.\mathsf{glb} \\ wrCount(cs) &\triangleq intHalf(logicalGlb(cs)) \end{aligned}$$

Function $writes(cs, as)$ returns the (abstract) write set of the writing transaction. This is the write set of the writing transaction, t, in the abstract state as provided t hasn't already linearized its commit operation, and is the empty set otherwise. Function $logicalGlb(cs)$ compensates for a lagging value of glb after a writing transaction's commit operation is linearized. Namely, it returns the glb incremented by 1 if a writer is already at $C2$. Finally, $wrCount(cs)$ is used to determine the number of committed writing transactions in cs since the most recent crash since $cs.glb$ is initially 0 and reset to 0 by the recovery operation, and moreover, $cs.glb$ is incremented twice by each writing transaction: once at line $W1$ and again at line $C2$ when the writing transaction commits.

$$
\begin{aligned}
&globalR(cs, as) = \\
&(\neg \mathsf{Recovering} \Rightarrow cs.\mathsf{vm} = last(as.L) \oplus writes(cs, as) \wedge && (3.8)\\
&\qquad\qquad wrCount(cs) + 1 = |as.L|) \wedge && (3.9)\\
&(cs.\mathsf{vm} \oplus cs.log) = last(as.L) \wedge && (3.10)\\
&\forall t. t \neq syst \wedge cs.pc_t = NotStarted \Rightarrow as.pc_t = NotStarted && (3.11)
\end{aligned}
$$

Conditions (3.8) and (3.9) assume that a recovery procedure is not in progress. By (3.8), the concrete volatile store is the last memory in $as.L$ overwritten with the write set of an in-flight writing transaction that has not linearized its commit operation. By (3.9), the number of memories recorded in the abstract state (since the last crash) is equal to $wrCount(cs) + 1$. By (3.10), the last abstract (persistent) store can be calculated from $cs.\mathsf{vm}$ by overriding it with the mappings in cs.log. Note that this is equivalent to undoing all uncommitted transactional writes. Finally, (3.11) ensures that every identifier for a transaction that has not started at the concrete level also has not started at the abstract level.

3.5.3.2 Transactional Relation of the dTML$_{\mathsf{SC}}$-dTMS2 Simulation Relation

We now turn to $txnR$. Its specification is very similar to the specification of $txnR$ in the proof of TML [44]. A part of $txnR$ maps concrete program counters to their abstract counterparts, which enables steps of the concrete program to be matched with abstract steps. To elaborate, this mapping provides adequate information for identifying the linearization points (the point in which an operation appears to take effect) of dTML$_{\mathsf{SC}}$. The linearization points of dTML$_{\mathsf{SC}}$ correspond to the concrete steps of the dTML$_{\mathsf{SC}}$ operations that simulate the execution of the corresponding abstract (dTMS2) operations. These steps are called non-stuttering steps, while all the other steps are called stuttering steps.

$cs.pc:$	$NotStarted$	$Aborted$
$as.pc:$	$NotStarted$	$Aborted$
$cs.pc:$	$Ready$	$Committed$
$as.pc:$	$Ready$	$Committed$
$cs.pc:$	$Bp, B1 \wedge odd(cs.\mathsf{loc}_t)$	$B1 \wedge even(cs.\mathsf{loc}_t), Br$
$as.pc:$	$BeginPending$	$BeginResponding$
$cs.pc:$	$Rp, R1 \wedge (cs.\mathsf{loc}_t \neq cs.\mathsf{glb}),$ $R2 \wedge (cs.\mathsf{loc}_t \neq cs.\mathsf{glb})$	$R1 \wedge (cs.\mathsf{loc}_t = cs.\mathsf{glb}),$ $R2 \wedge (cs.\mathsf{loc}_t = cs.\mathsf{glb}), Rr$
$as.pc:$	$ReadPending(cs.x_t)$	$ReadResponding(cs.v_t)$
$cs.pc:$	$Wp - W7$	$W8, Wr$
$as.pc:$	$WritePending(cs.x_t, cs.v_t)$	$WriteResponding$
$cs.pc:$	$Cp, C1$	$C2, Cr$
$as.pc:$	$CommitPending$	$CommitResponding$

Table 3.1: Mapping of dTML$_{\mathsf{SC}}$ to dTMS2 pc values.

As expected, the dTML$_{\mathsf{SC}}$ pc values are mapped to the same dTMS2 pc value in stuttering steps and transition to different dTMS2 pc values in non-stuttering steps.

For example, concrete pc values Wp,$W1$,$W2$ $\dots$, $W7$ correspond to abstract pc value $WritePending(cs.x_t, cs.v_t)$, whereas $W7$ corresponds to $WriteResponding$, indicating that, in our proof, the execution of line $W7$ corresponds to the execution of an abstract $\texttt{doWrite}_t(cs.x_t, cs.v_t)$ operation.

More precisely the linearization points of dTML$_{\mathsf{SC}}$ are as follows: The dTML$_{\mathsf{SC}}$ automaton transitions of invocation, response, and crash events are considered linearization points and thus simulate the corresponding invocation, response, and crash events of dTMS2. Operation `TMBegin` never aborts, and it linearizes at Bp when the value loaded in Bp is even. The linearization point of operation `TMRead` is at $R1$. Specifically, if no writing transaction has been performed by the time of $R1$ execution ($cs.\mathsf{loc}_t = cs.\mathsf{glb}$), then the `TMRead` operation linearizes to a successful read operation. Otherwise, it aborts. Operation `TMWrite` linearizes when the memory is updated at $W7$. Finally, operation `TMCommit` obtains two linearization points depending on whether the transaction has successfully executed a `TMWrite` operation. If so, $cs.\mathsf{loc}_t$ is odd, and `TMCommit` linearizes to a successful commit at $C1$. Otherwise, $cs.\mathsf{loc}_t$ is even, and `TMCommit` linearizes at Cp.

Moreover, $txnR$ specifies that each in-flight transaction t satisfies properties:

$$as.\mathsf{beginIdx}_t \le \mathit{intHalf}(cs.\mathsf{loc}_t) \tag{3.12}$$

$$as.\mathsf{rdSet}_t \subseteq as.L[\mathit{intHalf}(cs.\mathsf{loc}_t)] \tag{3.13}$$

Condition (3.12) ensures that a transaction does not introduce a transaction before its abstract begin index and (3.13) ensures that all reads are consistent with the last write in the abstract memories. Note that although dTMS2 (like TML2) allows read sets of a transaction to be validated against any memory in $mems$ after the transaction's begin index, dTML$_{\mathsf{SC}}$ (like TML) uses a single global lock for synchronisation, which means that transactions are forced to validate against the last memory in $mems$.

Conditions (3.12) and (3.13) helps to establish $as.validIdx(t, cs.\mathsf{loc}_t)$ for in-flight transaction t. To see this, first observe that in dTML$_{\mathsf{SC}}$, every in-flight transaction satisfies $cs.\mathsf{loc}_t \le \mathit{logicalGlb}(cs)$, and by (3.9) above, $\mathit{intHalf}(\mathit{logical\_glb}(cs)) < |as.L|$. Therefore,

$$cs.\mathsf{pc}_t \notin \{NotStarted, Bp\} \Rightarrow \mathit{intHalf}(cs.\mathsf{loc}_t) < |as.L| \tag{3.14}$$

Together, (3.12), (3.13) and (3.14) imply $as.validIdx(t, cs.\mathsf{loc}_t)$ for all in-flight transactions t.

Relation $txnR$ must also provide enough information to enable linearization of a commit operation against the correct abstract step. $txnR$ requires that for each in-flight the following equivalence must hold true, except when the `TMBegin` operation is occurring and during the interval between a successful compare-and-swap execution at line $W1$ and the subsequent write at line $W7$:

$$even(cs.\mathsf{loc}_t) \iff (as.\mathsf{wrSet}_t = \emptyset) \tag{3.15}$$

Because dTML$_{\mathsf{SC}}$ uses the parity of loc_t to determine whether a transaction is read-only, condition (3.15) enables us to prove the appropriate precondition when dTML$_{\mathsf{SC}}$ simulates a commit action.

Finally, $txnR$ must ensure that the recovery operation is such that the volatile store matches the last abstract store in $as.L$ prior to the crash. To achieve this, we require that $|as.L| = 1$ when *syst* is executing the recovery procedure, and the volatile store for the address being flushed at $Rec4$ matches the abstract state before the crash, i.e., $cs.\mathsf{vm}(cs.c_{syst}) = (as.L)[0]c_{syst}$. Since the recovery loop only terminates after the log is emptied, this ensures that the concrete memory state is consistent with the abstract memory prior to executing any transactions after a crash has occurred.

We now describe precisely the step correspondence between dTML$_{\mathsf{SC}}$ and dTMS2. For this, we use a step corresponding function sc of the form $sc(cs, t, \alpha) = \beta$, where cs is a concrete state, t is a transaction identifier, α is an internal action of dTML$_{\mathsf{SC}}$, and β is the internal action of dTMS2 that corresponds to α in case α indicates a non-stuttering step. In case that α indicates a stuttering step sc returns $\bot$.

- A **begin operation** takes effect when a transaction t reads an even value for glb at Bp. The pc mapping provided by $txnR$ (Fig. 3.1) guarantees that the corresponding abstract action is $\mathtt{doBegin}_t$. Thus, if $\alpha = Bp$ and $even(cs.\mathsf{glb})$ then $sc(cs, t, \alpha) = \mathtt{doBegin}_t$.
- A **read operation** takes effect when a transaction executes Rp and $cs.\mathsf{glb}$ at this point obtains the same value with $cs.\mathsf{loc}_t$. This indicates that there is no writing transaction in progress, or that has been committed since transaction t began its execution. Having this, it is guaranteed that the value read in Rp is either the last written value by transaction t or, if such write has not occurred, the last write on x by a committed transaction. This value corresponds to the value of x at the $as.L[intHalf(cs.\mathsf{loc}_t)]$ element of the abstract memory. As t is in-flight while executing a read operation by Equations 3.12, 3.13 and 3.14 it can be inferred that $intHalf(cs.\mathsf{loc}_t)$ meets the ordering constrains imposed by the dTMS2, i.e. $validIdx(t, intHalf(cs.\mathsf{loc}_t))$. The concrete to abstract mapping of program counter values (Fig. 3.1) guarantees that the value that has been provided as address and the value that has been returned coincide for both TMS2 and dTML$_{\mathsf{SC}}$. Therefore, if $\alpha = Rp$ and $\mathsf{loc}_t = \mathsf{glb}$ then $sc(cs, t, \alpha) = \mathsf{doRead}_t(cs.x_t) \wedge \exists n.n = intHalf(cs.\mathsf{loc}_t) \wedge validIdx(t, n)$.
- A **write operation** takes effect when a transaction t executes $W6$. The pc mapping provided by $txnR$ (Fig. 3.1) guarantees that the corresponding abstract action is $\mathtt{doWrite}_t(,t)$. Thus if $\alpha = W7$ a then $sc(cs, t, \alpha) = \mathtt{doWrite}_t(cs.x_t, cs.v_t)$. As before the pc mapping provided by $txnR$ ensures that $as.pc_t$ obtains the correct value.
- A **commit operation** takes effect at Cp for a read-only transaction and at $C1$ for a writing transaction. In both cases ($\alpha = Cp$ or $\alpha = C1$) its holds that $sc(cs, t, \alpha)$ $= \mathtt{doCommit}_t$. Equation (3.15) is used for determining if t is a read-only transaction ($as.\mathsf{wrSet}_t = \emptyset$). If so, the precondition of dTMS2 requires that $\exists n.validIdx(t, n)$. In a similar way to the case of a read operation, we can obtain that $validIdx(t, intHalf(cs.\mathsf{loc}_t))$ holds.

  If t is a writing transaction ($as.\mathsf{wrSet}_t \neq \emptyset$), the precondition of dTMS2 requires that $\mathbf{dom}(as.\mathsf{rdSet}_t) \wedge \subseteq last(as.L)$. We already have that at $C1$ $validIdx(t, intHalf(cs.\mathsf{loc}_t))$ holds by Equations 3.12, 3.13, 3.14 and that $logicalGlb(cs) = \mathsf{loc}_t$ by the dTML$_{\mathsf{SC}}$ invariant (see §3.4.2.1). By Equation 3.9, we can infer that loc_t, in this case, corresponds to the last element of the abstract memory $as.L$ and by unfolding the $validIdx$ definition, we can conclude that $\mathsf{rdSet}_t \subseteq last(as.L)$.

  In all other cases $sc(cs, t, \alpha) = \bot$.

3.6 Related Work

In this work, we have defined durable opacity, a new correctness condition for STMs, inspired by durable linearizability [23] for concurrent objects. The condition assumes a history with crashes such that in-flight transactions are aborted (i.e., do not continue) after a crash takes place, and simply requires that the history satisfies opacity [67, 68] after the crashes are removed. This is a strong notion of correctness but ensures safety for STMs in the same way that durable linearizability [23] ensures safety for concurrent objects. An alternative weaker correctness condition for persistent STMs is PSER (persistent serializability) [123], which extends serializability to the persistency setting.

Our focus has been on the formalization of durable opacity and the development of an example algorithm and verification technique. Our implementation assumes the persistent SC memory model, which is a simplified model that does not consider explicit persist instructions or instruction reorderings that may occur in a more realistic setting. Khyzha *et al.* [90] have proposed a more sophisticated SC-based persistent memory model. While their model does not consider TSO store buffers, it includes per-location persistence buffers which are able to simulate asynchronous persist operations and persist barriers.

Our example implementation, dTML$_{\mathsf{SC}}$ extends TML with a persistent undo log, and associated modifications such as the introduction of a recovery operation. The undo log technique is used by several persistent STMs [30,34,77,92,102,150] as means of achieving failure atomicity. The technique requires, in the worst case scenario, two flushes (or two persis barriers in case optimized flushes are used) per write, a flush of the corresponding to the write log entry, and a flush of the actual write after the update of the log.

An alternative technique comprises using a redo log [62, 66, 101, 125, 141]. As discussed by Ramalhete *et al.* [124] , this technique requires two persist barriers (assuming the use of optimized flushes) per writing transaction, regardless of the number of updates within the transaction. To elaborate, a writing transaction firstly updates the persistent log with the new write, prior to updating the memory. The new write is flushed to persistent memory only after the transaction has been successfully committed. In the event of a crash, the writes stored in the persistent log are reapplied and subsequently, the persistent log is emptied.

Another option [37] is based on shadowing data. This method requires constantly maintaining a complete replica of the data. Other persistent transactional memory algorithms rely on applying hardware modifications for achieving failure atomicity [15, 79, 87, 127, 137]. We decide here to not focus on hardware persistent memory transactions, as their application require more intervening methods that go beyond the general applicability of software transactional memories.

3.6.1 Persistent Memory Implementations

dTML$_{\mathsf{SC}}$ serves as a baseline example to demonstrate our verification method, which does not specifically address implementation details such as garbage collection or persistent memory allocation. For reference purposes, below we provide a brief overview of well-established frameworks for adapting volatile algorithms to the persistency setting including persistent STMs.

Koburn *et al.* [34] implemented NV-heaps to integrate persistent objects into conventional programs while addressing safety issues commonly found in predominantly persistent memory models. NV-heaps utilize ACID transactions to ensure safety and enable reasoning about the timing and order of data structure changes becoming persistent. They offer type-safe pointers and employ garbage collection through reference counting. However, NV-heaps have some drawbacks, including the high cost of transactions, the use of locks that can result in unbounded rollback effects during crashes, and the extensive logging required in persistent memory, leading to a trade-off between safety and efficiency. Additionally, both NV-heaps and Mnemosyne [141] only support updates to persistent memory within transactions and critical sections. Other systems like Stasis [130] and BerkeleyDB [113] are also based on ACID transactions and offer consistency guarantees for persistent memory in the realm of databases.

ATLAS [30] provides durability semantics for lock-based implementations. All the synchronization operations are expressed in terms of lock and unlock. The consistency of data structures is guaranteed only for the non-lock-objects as data structures can be modified and temporarily violate invariants when locked. ATLAS ensures that the outermost critical sections, which are protected by one or more mutexes, are failure-atomic by identifying failure-atomic sections (FASEs). These sections ensure that if at least one update to a persistent location within a FASE is durable, then all the updates within that FASE are durable. An undo persistent log is kept that tracks the synchronization operations and persistent stores and allows the recovery to rollback FASEs that were interrupted by crashes. The log entries can be parallelized. A log entry consists of the store type, size, and destination address, as well as the original value at that address. A helper is used to minimize cache line flushes, which, along with logging, constitutes the predominant efficiency costs.

Izrealevirz *et al.* [79], developed a logging mechanism based on undo and redo log properties named JUSTDO logging. This mechanism aims to reduce the memory size of log entries while preserving data integrity after crash occurrences. Unlike optimistic transactions [30], JUSTDO logging resumes the execution of interrupted FASEs to their last store instruction and then executes them until completion. One disadvantage of this strategy is that the FASEs cannot be rolled back after a system failure. As a consequence, there is no tolerance for bugs inside the FASEs. In this system, it is assumed that the cache memory is persistent, and the system also requires that all load/store instructions access persistent data. A small log is maintained for each thread, that records its most recent store within a FASE. The small per-thread logs simplify log management and reduce memory requirements.

As mentioned in §2.4.3, Ben-David *et al.* [19], developed a system that can transform algorithms that consist of read, write and CAS operations in shared memory, to equivalent *detectable* suitable for persistent memory. The system aims to create concurrent algorithms that guarantee consistency after a crash. This is done by introducing persist checkpoints, which record the current state of the execution and from which the execution can continue after a fault. Two consecutive checkpoints form a *capsule*. When a fault occurs inside a capsule the program execution is continued from the previous capsule boundary. It is ensured that every capsule is correct (can be repeated safely). The authors assume the Parallel Persistent Model (PPM) which consists of P processors, each with a local ephemeral (volatile) memory of limited size and a sharing persistent memory. Two persistent write buffers are kept along with a persistent bit in order to solve write-after-read conflicts. A copy of the ephemeral variables or the valid write buffer is kept in persistent memory at the end of each capsule. Those values are restored after a crash. CAS operations are replaced with recoverable CAS operations [13], which are wrapped with a mechanism that guarantees that CAS executions are not repeated after a successful execution. The modified algorithm has constant computational and recovery delays, both of which can be decreased by applying several optimizations.

Mnemosyne [141] provides a low-level interface to persistent memory with high-level transactions based on TinySTM [56] and a redo log that is purposely chosen to reduce ordering constraints. The log is flushed at the commit of each transaction. As a result, the memory locations that are written by a transaction remain unmodified until committed. Each read operation checks whether data has been modified and if so, returns the buffered value instead of the value from the memory. The size of the log increases proportionally to the size of the transaction, potentially making the checking time-consuming.

Joshi *et al.* [86], proposed durable hardware transactional memory (DHTM), a hardware-based solution for ACID transactions, that use commercial HTM to provide atomic visibility and extends it with hardware support for redo logging to provide failure atomicity. The same logging infrastructure is also used to support L1 overflows and, thus, transactions of larger size.

FliT [144] is a C++ algorithm that can be used for making any linearizable data structure persistent. The primary optimization offered by FliT concerns minimizing the number of flush operations required for durability. It achieves this by selectively flushing only those writes that are subsequently read. The method for performing this involves utilizing counters, to monitor ongoing stores for each variable. When a store operation begins, it marks the corresponding memory location by incrementing its associated counter. During load operations, the counter for a given memory location is checked, and a flush instruction is only executed if the location is tagged. This approach ensures that flush instructions are performed only when necessary. This technique provides flexibility in terms of counter placement. Counters can be positioned adjacent to each variable or stored in a separate hash table.

Chapter 4

Logics for Px86

In this chapter, we present a program logic that addresses programs following the Px86 memory model. In contrast to *persistent sequential consistency* (Chapter 4), Px86 is a realistic memory model that addresses the out-of-order persistency of stores and the asynchronous behavior of explicit persist instructions such as `clflushopt`. Concisely, Px86 is a combination of relaxed buffered persistency with TSO. Our logic, PIEROGI, aims to help programmers reason about low-level operations such as memory accesses and fences, as well as persistency primitives such as flushes. PIEROGI benefits from a simple underlying operational semantics based on *views*, is able to handle *optimized* flush operations, and is mechanized in the Isabelle/HOL.

Specifically, PIEROGI can reason efficiently about x86 persistency thanks to two key recent advances: 1) $\text{Px86}_{\text{view}}$ [32] (which is known to be equivalent to the declarative Px86 model [78,122]), the view-based operational semantics of x86 persistency; and 2) the C11 Owicki-Gries logic [39, 40, 43] to reason about view-based operational semantics, which we adapt to $\text{Px86}_{\text{view}}$.

In the endeavor of establishing a logic for the Px86 model, we encountered two main challenges.

(1) ***Capturing Program Correctness After a Crash***. Our understanding of what it means for a program to be correct after a system crash has evolved over time. Similar to previous works on verifying Px86 programs [32,119], the initial version of our logic, $\text{PIEROGI}_{\text{simp}}$, focused solely on reasoning about the behavior of a program *up to the first crash.* In this version, system crashes do not correspond to state transitions and, hence, do not affect the program state. Instead, the correctness of persistent memory is defined as a predicate over the state's *memory component* and *view components* related to persistency. We have used $\text{PIEROGI}_{\text{simp}}$ to verify several litmus tests (see Chapter 5). Later, while trying to verify a $\text{Px86}_{\text{view}}$ version of dTML_{SC} (Chapter 6), we recognized the importance of reasoning about program behavior after a system restart due to a crash. Consequently, we have extended $\text{PIEROGI}_{\text{simp}}$ to allow operational reasoning about post-crash behavior, including recovery and subsequent execution ($\text{PIEROGI}_{\text{full}}$). To establish $\text{PIEROGI}_{\text{full}}$, we develop an extended version of $\text{Px86}_{\text{view}}$, which includes a CRASH transition that enables reasoning after a system crash. Furthermore, it incorporates a recovery mechanism to signify when the system is under recovery. We denote the original version of $\text{Px86}_{\text{view}}$ as $\alpha\text{Px86}_{\text{view}}$ and the enhanced version as $\beta\text{Px86}_{\text{view}}$.

(2) ***Defining Adequate View-Based Assertions.*** While verifying the $\text{Px86}_{\text{view}}$ version of $\text{dTML}_{\mathsf{SC}}$ ($\text{dTML}_{\text{Px86}}$), we realized that the assertions of $\textsc{Pierogi}_{\text{simp}}$ are insufficient for reasoning about certain phenomena. Specifically, in the $\text{dTML}_{\text{Px86}}$ case, it is often necessary to reason about memory patterns by considering the order in which writes occur. To address this, $\textsc{Pierogi}_{\text{full}}$ was enhanced with assertions capable of expressing more complex memory patterns.

Throughout this chapter, the term $\text{Px86}_{\text{view}}$ will be used to collectively denote both the $\alpha\text{Px86}_{\text{view}}$ and $\beta\text{Px86}_{\text{view}}$ models. Furthermore, the term $\textsc{Pierogi}$ will be used to refer to both $\textsc{Pierogi}_{\text{simp}}$ and $\textsc{Pierogi}_{\text{full}}$.

In this chapter, we begin with presenting the view-based operational semantics of x86 persistency (§4.1). Next, we describe the assertion language and proof rules of $\textsc{Pierogi}_{\text{simp}}$ and $\textsc{Pierogi}_{\text{full}}$ (§4.2). Afterward, we discuss the Isabelle/HOL mechanization (§4.3). Finally, we discuss related work (§4.4). The Isabelle/HOL mechanization of $\textsc{Pierogi}_{\text{simp}}$ can be found at https://doi.org/10.6084/m9.figshare.18469103.v2. The Isabelle/HOL mechanization of $\textsc{Pierogi}_{\text{full}}$ can be found at
https://doi.org/10.6084/m9.figshare.25037312.v2.

4.1 $\text{Px86}_{\text{view}}$ Syntax and Semantics

Both versions of our logic ($\textsc{Pierogi}_{\text{simp}}$, $\textsc{Pierogi}_{\text{full}}$) are built upon the $\text{Px86}_{\text{view}}$ semantics proposed by Cho *et al.* [32]. We chose these semantics because they provide a simple abstraction of the Px86 architectural details. In the following section, we will present the original $\text{Px86}_{\text{view}}$ model as well as our modified version, $\beta\text{Px86}_{\text{view}}$. We use $\alpha\text{Px86}_{\text{view}}$ and $\beta\text{Px86}_{\text{view}}$ for establishing $\textsc{Pierogi}_{\text{simp}}$ and $\textsc{Pierogi}_{\text{full}}$ respectively.

4.1.1 Programming Language

The syntax of our language is given below, which is the syntax from prior work [32]. We use precisely the same formalization to model programs as with the persistent SC language (§3.2.1). A program is as before a function that maps a thread identifier (t) and the label of the labeled statement that is currently executed by t to the labeled statement that is going to be executed next. The control flow within each thread is again tracked via the program counter function pc which records the program counter of each thread. As before, we assume $\iota \in \textsc{Lab}$ is representing the initial label of each thread, and $\zeta \in \textsc{Lab}$ the final label.

The $\text{Px86}_{\text{view}}$ programming language supports all the atomic primitives of the SC language as well as the optimized flush ($\textbf{flush}_{\text{opt}}$), the memory fence (**mfence**), and the store fence (**sfence**) instructions of Px86 instruction set.

$$\begin{array}{l}
v, u \in \textsc{Val} \triangleq \mathbb{N} \quad x, y, \ldots \in \textsc{Loc} \quad o \in \textsc{DObj} \quad f \in \textsc{F} \quad a, b, \ldots \in \textsc{Reg} \quad t \in \textsc{Tid} \triangleq \mathbb{N} \\
i, j, k, \ldots \in \textsc{Lab} \qquad \hat{a}, \hat{b}, \ldots \in \textsc{AuxVar} \qquad \hat{e} \in \textsc{AuxExp} ::= v \mid \hat{a} \mid \hat{e}+\hat{e} \mid \cdots \\
\quad e \in \textsc{Exp} ::= v \mid a \mid e+e \mid \cdots \qquad\qquad B \in \textsc{BExp} ::= \mathsf{true} \mid B \wedge B \mid \cdots \\
\quad \alpha \in \textsc{ASt} ::= \textbf{skip} \mid a := e \mid a := \textbf{load}\, x \mid \textbf{store}\ x\ e \\
\qquad\qquad\qquad \mid \textbf{sfence} \mid \textbf{mfence} \mid \textbf{flush}\ x \mid \textbf{flush}_{\text{opt}}\ x \\
\qquad\qquad\qquad \mid a := \textbf{CAS}\ x\ e\ e \mid \textbf{flush}\ x \mid o.f \\
\quad ls \in \textsc{LSt} ::= \alpha\ \textbf{goto}\ j \mid \textbf{if}\ B\ \textbf{goto}\ j\ \textbf{else to}\ k \mid \langle \alpha\ \textbf{goto}\ j, \hat{a} := \hat{e} \rangle \\
\quad \Pi \in \textsc{Prog} \triangleq \textsc{Tid} \times \textsc{Lab} \to \textsc{LSt} \qquad\qquad pc \in \textsc{PC} \triangleq \textsc{Tid} \to \textsc{Lab}
\end{array}$$

4.1.2 Px86$_{\text{view}}$ Semantics

We now introduce the Px86$_{\text{view}}$ semantics as proposed by Cho *et al.* [32]. As previously mentioned, we will refer to the original Px86$_{\text{view}}$ model as αPx86$_{\text{view}}$ and to our enhanced version as βPx86$_{\text{view}}$.

4.1.2.1 The Px86$_{\text{view}}$ Machine State

Like previous view-based models, Px86$_{\text{view}}$ employs a non-standard memory capturing all previously executed writes, alongside with so-called "thread views" that track several position(s) of each thread in that history and enforce limitations on the ability of the thread to read from and write to the memory. In addition, the thread views contain the necessary information for determining the possible contents of the persistent memory upon a system crash. Formally, Px86$_{\text{view}}$'s memory and thread states are defined as follows.

Definition 4.1.1 (Px86$_{\text{VIEW}}$'s MEMORY)**.** A memory M is a list of messages, the first of which is a store CM : $\textsc{Loc} \to \textsc{Val}$, and the subsequent messages have the form $\langle \textsc{Loc} := \textsc{Val} \rangle$. Initially, we assume $M = \langle CM \rangle$, where $CM(x) = 0$ for all $x \in \textsc{Loc}$. We use $w.\mathsf{loc}$ and $w.\mathsf{val}$ to refer to the two components of a message w. We use standard list notations for memories (e.g. $M_1 +\!\!+ M_2$ for appending memories, $[w]$ for a singleton memory, and $|M|$ for the length of M). We refer to the indices of the memory list as *timestamps.* For $ts > 0$, the location and a value of a message m are denoted as $m.\mathsf{loc}$ and $m.\mathsf{val}$, respectively. Every execution of a **store** $x\ v$ instruction adds a message $\langle x := v \rangle$ to the memory list. We assume that the memory's initial state includes a single message that serves for initializing every location x ($\in \textsc{Loc}$). In the case of αPx86$_{\text{view}}$ the initial message maps every memory location to 0. On the contrary, in the case of βPx86$_{\text{view}}$ the initial message maps every memory location x either to 0 (initial state) or the last value persisted to x before a crash. A $a := \textbf{load}\, x$ instruction that happens to read the first message of the memory returns $M[0](x)$, otherwise it returns $M[ts].\mathsf{val}$ (assuming $M[ts].\mathsf{loc} = x$). In the case of the αPx86$_{\text{view}}$ model the value of $M[0](x)$ is always 0. To capture both scenarios, we use the notation $M[ts] \equiv \langle x := v \rangle$. We say that a message with timestamp ts_1 and location x is not overwritten from timestamp p's perspective if the following holds: $\forall ts \in (ts_1, p].\, M[ts].\mathsf{loc} \neq x$. We denote this as $x \notin M(ts_2..ts_1]$. Furthermore, we use $\sqcup$ for obtaining the maximum among timestamps (i.e. $ts_1 \sqcup ts_2 = \max(ts_1, ts_2)$), and extend this notation pointwise to functions.

Definition 4.1.2 (Px86$_{\text{VIEW}}$'S THREAD STATE)**.** A *thread state* $T \in \textsc{Thread}$ is a record consisting of the following fields: $\mathsf{coh} : \textsc{Loc} \to \mathbb{N}$, $\mathsf{v_{rNew}} : \mathbb{N}$, $\mathsf{v_{pReady}} : \mathbb{N}$, $\mathsf{v_{pAsync}} : \textsc{Loc} \to$

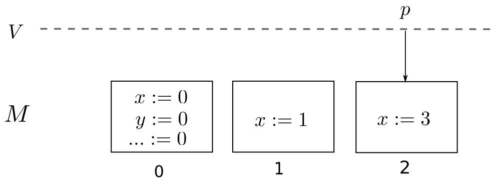


Figure 4.1: A snapshot of the $\mathrm{Px86_{view}}$ memory and a thread state *view*.

$\mathbb{N}$, and $\mathsf{v_{pCommit}} : \textsc{Loc} \to \mathbb{N}$. The above components of the thread states are called *views*. A thread state $T \in \textsc{Thread}$ is a record of *views* of M and registers. Each $\mathrm{Px86_{view}}$ instruction moves the views in such way that constrain its ordering with previously or later issued instructions and reflect its effect in persistent and volatile memory. We use standard function/record update notation (e.g. $T' = T[\mathsf{coh}(x) \mapsto ts]$ denotes the thread state obtained from T be modifying the x entry in the coh component of T to ts). In addition, $\mapsto_{\sqcup}$ is used to incorporate certain timestamps in fields (e.g. $T[\mathsf{v_{rNew}} \mapsto_{\sqcup} ts]$ denotes the thread state obtained from T be modifying the $\mathsf{v_{rNew}}$ component of T to $T.\mathsf{v_{rNew}} \sqcup ts$). We denote by $T.\mathsf{maxcoh}$ the maximum among the coherence view timestamps ($T.\mathsf{maxcoh} = \bigsqcup_x T.\mathsf{coh}(x)$). In addition, we denote by $\sigma.\mathsf{maxpCommit}(x)$ the maximum among the persistency view timestamps for location x ($\sigma.\mathsf{maxpCommit} = \bigsqcup_t \sigma.\mathbb{T}(t).\mathsf{v_{pCommit}}(x)$). Fig. 4.1 depicts a memory (M) that contains the initial message, the message $\langle x := 1\rangle$ and the message $\langle x := 3\rangle$. The arrow p represents a *view* which equals to timestamp 2.

In the table below, we provide a short description of the views of the thread state of $\mathrm{Px86_{view}}$. We denote the pre-state state as σ, the post-state as σ', and the executing thread as t.

View: $\mathsf{coh} : \textsc{Loc} \to \mathbb{N}$

Moved by: store, load, CAS: Both a **store** and a successful **CAS** on x will update $\sigma.\mathbb{T}(t).\mathsf{coh}(x)$ to match the length of the memory in the pre-state. A **load** and a failed **CAS** on x updates $\sigma.\mathbb{T}(t).\mathsf{coh}(x)$ to the timestamp of the message of the read value.

Purpose: In conjunction with $\mathsf{v_{rNew}}$, determines the range of observable values by t for the specified location. When a memory message with the location x is about to be added to the memory by a **store** or a successful **CAS** operation, its timestamp in the post-state is equal to $\sigma.\mathbb{T}(t).\mathsf{coh}(x)$. Additionally, a message that is accessed by a **load** or **CAS** instruction must have a timestamp that is greater than or equal to the value of $\sigma.\mathbb{T}(t).\mathsf{coh}(x)$.

View: $\mathsf{v_{rNew}} : \mathbb{N}$

Moved by: mfence, **load**(external), **CAS**(fail-external/success): When t executes an **mfence**, $\sigma'.\mathbb{T}(t).\mathsf{v_{rNew}}$ is updated to the timestamp of the latest write performed by t provided that it is greater than the current value of $\mathsf{v_{rNew}}$ ($\sigma.\mathbb{T}(t).\mathsf{v_{rNew}}$). When t executes an external **load** or an external failed **CAS**, $\mathsf{v_{rNew}}$ is updated to the timestamp of the read message, again provided that it is greater than the current value of $\mathsf{v_{rNew}}$. When t executes a successful **CAS** it updates $\sigma'.\mathbb{T}(t).\mathsf{v_{rNew}}$ to the length that memory had in the pre-state.

Purpose Together with coh determines the set of visible values for the given location to t. No memory message that is read by t (via a **load** or a **CAS** instruction) obtains a timestamp that is overwritten from the $(\sigma.\mathbb{T}(t).\mathsf{v_{rNew}})$'s perspective.

View: $\mathsf{v_{pReady}} : \mathbb{N}$

Moved by: **load**(external), **CAS**(fail-external/success), **mfence**, **sfence**: Instructions **load**(external), **CAS**(fail-external/success) and **mfence** update $\sigma'.\mathbb{T}(t).\mathsf{v_{pReady}}$ in the same way that they update $\sigma'.\mathbb{T}(t).\mathsf{v_{rNew}}$. The **sfence** instruction updates $\sigma'.\mathbb{T}(t).\mathsf{v_{pReady}}$ in similar manner as **mfence**.

Purpose: It is used for ordering **load sfence**, **mfence** and **CAS** instructions with subsequent $\mathbf{flush}_{\text{opt}}$ instructions.

View: $\mathsf{v_{pAsync}} : \textsc{Loc} \rightarrow \mathbb{N}$

Moved by: **flush**, $\mathbf{flush}_{\text{opt}}$: When t executes a **flush** on x, $\sigma'.\mathbb{T}(t).\mathsf{v_{pAsync}}(x)$ is updated to the timestamp of the latest write performed by t provided that it is greater than $\sigma'.\mathbb{T}(t).\mathsf{v_{pAsync}}(x)$. When t executes a $\mathbf{flush}_{\text{opt}}$ on x, $\mathsf{v_{pAsync}}(x)$ is updated to the maximum between $\sigma.\mathbb{T}(t).\mathsf{coh}(x)$, $\sigma.\mathbb{T}(t).\mathsf{v_{pReady}}(x)$ and $\sigma.\mathbb{T}(t).\mathsf{v_{pAsync}}(x)$.

Purpose: Determines the set of values that may hold for a given location in persistent memory after the execution of an **sfence** preceded by the execution of a $\mathbf{flush}_{\text{opt}}$. Any memory message the value of which is about to be persisted after the execution of a barrier ($\mathbf{sfence}, \mathbf{mfence}, \mathbf{CAS}$), is not overwritten from the $\sigma'.\mathbb{T}(t).\mathsf{v_{pAsync}}$'s perspective in the post-state of a $\mathbf{flush}_{\text{opt}}$ execution.

View: $\mathsf{v_{pCommit}} : \textsc{Loc} \rightarrow \mathbb{N}$

Moved by: **flush**, **CAS**(success), **mfence** and **sfence**: A **flush** on x updates $\sigma'.\mathbb{T}(t).\mathsf{v_{pCommit}}(x)$ to the maximum between the timestamp of the latest write by t, and $\sigma.\mathbb{T}(t).\mathsf{v_{pCommit}}(x)$. Instructions **sfence** and **mfence** update $\sigma'.\mathbb{T}(t).\mathsf{v_{pCommit}}(x)$ of all $x \in \textsc{Loc}$ to the maximum between $\sigma.\mathbb{T}(t).\mathsf{v_{pAsync}}(x)$ and $\sigma.\mathbb{T}(t).\mathsf{v_{pCommit}}(x)$. A successful **CAS** instruction updates $\sigma'.\mathbb{T}(t).\mathsf{v_{pCommit}}(x)$ to the length that the memory had in the pre-state.

Purpose: Contributes to determining the set of values that may hold for a given location in persistent memory. The set of values for a location x that a thread can observe in persistent memory is common for all the threads. The set is determined by the maximum value of $\sigma.\mathbb{T}(t).\mathsf{v_{pCommit}}(x)$ among all the threads ($\sigma.\mathsf{maxpCommit}$). No memory message whose value reached the persistent memory after the execution of a persistent barrier or a **flush** instruction has a timestamp that is overwritten from the $\sigma.\mathsf{maxpCommit}$'s perspective after the completion of a **flush** or persist barrier.

Below we define the αPx86$_{\text{view}}$'s and βPx86$_{\text{view}}$'s machine state. The βPx86$_{\text{view}}$ state consists of all the components of αPx86$_{\text{view}}$, but also adopts the recovery mechanism used in the persistent SC semantics (§3.2.2.1), in order to signify when a system is under recovery. Specifically, it includes the $\mathit{rec} : \mathit{bool}$ component. When a crash occurs, rec is assigned the value true to signal the initiation of an implementation-specific recovery process. After the recovery process finishes its execution, rec is reset to false.

Definition 4.1.3 (αPx86$_{\text{VIEW}}$'s machine state)**.** A *αPx86$_{view}$ machine state* is a tuple $\sigma = \langle pc, \mathbb{T}, M, G\rangle$ where $pc : \textsc{Tid} \rightarrow \textsc{Lab}$ is a mapping assigning the next program label to be executed by each thread, $\mathbb{T} : \textsc{Tid} \rightarrow \textsc{Thread}$ is a mapping assigning the current thread state to each thread, $M \in \textsc{Memory}$ is the current memory, and $G : \textsc{AuxVar} \rightarrow \textsc{Val}$ is storing the current values of the auxiliary variables. We assume that G is extended to expressions $\hat{e} \in \textsc{AuxExp}$ in a standard way.

Definition 4.1.4 (βPx86$_{\text{VIEW}}$'s machine state)**.** A *βPx86$_{view}$ machine state* is a tuple $\sigma = \langle pc, \mathit{rec}, \mathbb{T}, M, G\rangle$ where $\mathit{rec} : \mathit{bool}$ is a flag that indicates when a recovery process is in progress. In the event of a crash, rec is set to true to indicate that an implementation-specific recovery process is about to start its execution. Once the recovery process is completed, rec is reset to false. All the remaining components of the machine state are defined as in in Def. 4.1.3.

We denote the components of a machine state σ by $\sigma.\mathit{pc}$, $\sigma.\mathbb{T}$, etc. For both αPx86$_{\text{view}}$ and βPx86$_{\text{view}}$ in the initial state (σ_{init}), $pc(t) = \iota$ for all $t \in \textsc{Tid}$, $\mathit{regs}(t)\, r = 0$ for all $t \in \textsc{Tid}$ and $r \in \textsc{Reg}$, and $G(a) = 0$ for all $a \in \textsc{AuxVar}$. Furthermore, in the initial state, the memory M contains only the initial message which maps all the memory locations ($x \in \textsc{Loc}$) to 0. In the case of βPx86$_{\text{view}}$, also $\mathit{rec} = \mathsf{false}$.

4.1.2.2 Px86$_{\text{view}}$ Transitions

The transitions presented below closely follow the model in [32] with minor presentational simplifications. Note that as in §3.2.2.3 we assume that writes persist atomically at the location granularity.

Transitions under the αPx86$_{view}$ model. The transitions of αPx86$_{\text{view}}$ are presented in Fig. 4.2 and Fig. 4.3.

(PROGRAM-NORMAL)

$$\frac{\begin{array}{c} pc(t) = i \qquad \Pi(t,i) = \alpha\ \textbf{goto}\ j \\ \langle \mathbb{T}(t), M\rangle \xrightarrow{\alpha} \langle T', M'\rangle \\ pc' = pc[t \mapsto j] \qquad \mathbb{T}' = \mathbb{T}[t \mapsto T'] \end{array}}{\langle pc, \mathbb{T}, M, G\rangle \Rightarrow_{\Pi} \langle pc', \mathbb{T}', M', G\rangle}$$

(PROGRAM-IF)

$$\frac{\begin{array}{c} pc(t) = i \qquad \Pi(t,i) = \textbf{if}\ B\ \textbf{goto}\ j\ \textbf{else to}\ k \\ pc' = pc\left[t \mapsto \begin{cases} j & \mathbb{T}(t).\mathsf{regs}(B) = \mathsf{true} \\ k & \mathbb{T}(t).\mathsf{regs}(B) = \mathsf{false} \end{cases}\right] \end{array}}{\langle pc, \mathbb{T}, M, G\rangle \Rightarrow_{\Pi} \langle pc', \mathbb{T}, M, G\rangle}$$

(PROGRAM-GHOST)

$$\frac{\begin{array}{c} pc(t) = i \qquad \Pi(t,i) = \langle \alpha\ \textbf{goto}\ j, \hat{a} := \hat{e}\rangle \\ \langle \mathbb{T}(t), M\rangle \xrightarrow{\alpha} \langle T', M'\rangle \\ pc' = pc[t \mapsto j] \qquad \mathbb{T}' = \mathbb{T}[t \mapsto T'] \qquad G' = G[\hat{a} \mapsto G(\hat{e})] \end{array}}{\langle pc, \mathbb{T}, M, G\rangle \Rightarrow_{\Pi} \langle pc', \mathbb{T}', M', G'\rangle}$$

Figure 4.2: Control flow transitions of αPx86$_{\text{view}}$ for a program Π

$$(\textsc{assign})\quad \frac{\begin{array}{c}\alpha = a := e\\ v = T.\mathsf{regs}(e)\\ T' = T[\mathsf{regs}(a) \mapsto v]\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

$$(\textsc{store})\quad \frac{\begin{array}{c}\alpha = \mathbf{store}\ x\ e\\ v = T.\mathsf{regs}(e)\\ M' = M \mathbin{+\!\!+} [\langle x := v\rangle]\\ T' = T[\mathsf{coh}(x) \mapsto |M|]\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M'\rangle}$$

$$(\textsc{load-internal})\quad \frac{\begin{array}{c}\alpha = a := \mathbf{load}\, x\\ M[ts] = \langle x := v\rangle\\ T.\mathsf{coh}(x) = ts\\ T' = T[\mathsf{regs}(a) \mapsto v]\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

$$(\textsc{load-external})\quad \frac{\begin{array}{c}\alpha = a := \mathbf{load}\, x\\ M[ts] = \langle x := v\rangle\\ T.\mathsf{coh}(x) < ts\\ x \notin M(ts..T.\mathsf{v}_{\mathsf{rNew}}]\end{array} \qquad T' = T\begin{bmatrix}\mathsf{regs}(a) \mapsto v,\\ \mathsf{coh}(x) \mapsto ts,\\ \mathsf{v}_{\mathsf{rNew}} \mapsto_{\sqcup} ts,\\ \mathsf{v}_{\mathsf{pReady}} \mapsto_{\sqcup} ts\end{bmatrix}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

$$(\textsc{sfence})\quad \frac{\begin{array}{c}\alpha = \mathbf{sfence}\\ T' = T\begin{bmatrix}\mathsf{v}_{\mathsf{pReady}} \mapsto_{\sqcup} T.\mathsf{maxcoh},\\ \mathsf{v}_{\mathsf{pCommit}} \mapsto_{\sqcup} T.\mathsf{v}_{\mathsf{pAsync}}\end{bmatrix}\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

$$(\textsc{flush})\quad \frac{\begin{array}{c}\alpha = \mathbf{flush}\ x\\ T' = T\begin{bmatrix}\mathsf{v}_{\mathsf{pAsync}}(x) \mapsto_{\sqcup} T.\mathsf{maxcoh},\\ \mathsf{v}_{\mathsf{pCommit}}(x) \mapsto_{\sqcup} T.\mathsf{maxcoh}\end{bmatrix}\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

$$(\textsc{flushopt})\quad \frac{\begin{array}{c}\alpha = \mathbf{flush}_{\mathrm{opt}}\ x\\ T' = T[\mathsf{v}_{\mathsf{pAsync}}(x) \mapsto_{\sqcup} T.\mathsf{coh}(x) \sqcup T.\mathsf{v}_{\mathsf{pReady}}]\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

$$(\textsc{mfence})\quad \frac{\begin{array}{c}\alpha = \mathbf{mfence}\\ T' = T\begin{bmatrix}\mathsf{v}_{\mathsf{rNew}} \mapsto_{\sqcup} T.\mathsf{maxcoh},\\ \mathsf{v}_{\mathsf{pReady}} \mapsto_{\sqcup} T.\mathsf{maxcoh}\end{bmatrix}\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

$$(\textsc{cas-success})\quad \frac{\begin{array}{c}\alpha = a := \mathbf{CAS}\ x\ e_1\ e_2\\ v_1 = T.\mathsf{regs}(e_1)\\ v_2 = T.\mathsf{regs}(e_2)\\ M[ts] = \langle x := v_1\rangle\\ x \notin M(ts..|M| - 1]\\ M' = M \mathbin{+\!\!+} [\langle x := v_2\rangle]\end{array} \qquad T' = T\begin{bmatrix}\mathsf{regs}(a) \mapsto \mathsf{true},\\ \mathsf{coh}(x) \mapsto |M|,\\ \mathsf{v}_{\mathsf{rNew}} \mapsto_{\sqcup} |M|,\\ \mathsf{v}_{\mathsf{pReady}} \mapsto_{\sqcup} |M|,\\ \mathsf{v}_{\mathsf{pCommit}} \mapsto_{\sqcup} T.\mathsf{v}_{\mathsf{pAsync}}\end{bmatrix}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M'\rangle}$$

$$(\textsc{cas-fail-internal})\quad \frac{\begin{array}{c}\alpha = a := \mathbf{CAS}\ x\ e_1\ e_2\\ M[ts] = \langle x := v\rangle\\ T.\mathsf{coh}(x) = ts\\ (x \in M(ts..|M| - 1] \vee v \neq T.\mathsf{regs}(e_1))\end{array} \qquad T' = T\,[\mathsf{regs}(a) \mapsto \mathsf{false}]}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

$$(\textsc{cas-fail-external})\quad \frac{\begin{array}{c}\alpha = a := \mathbf{CAS}\ x\ e_1\ e_2\\ M[ts] = \langle x := v\rangle\\ T.\mathsf{coh}(x) < ts\\ (x \in M(t..|M| - 1] \vee v \neq T.\mathsf{regs}(e_1))\end{array} \qquad T' = T\begin{bmatrix}\mathsf{regs}(a) \mapsto \mathsf{false},\\ \mathsf{coh}(x) \mapsto ts,\\ \mathsf{v}_{\mathsf{rNew}} \mapsto_{\sqcup} ts,\\ \mathsf{v}_{\mathsf{pReady}} \mapsto_{\sqcup} ts\end{bmatrix}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

Figure 4.3: Instruction transitions of $\mathrm{Px86}_{\mathrm{view}}$ for a program Π

Transitions under the βPx86$_{view}$ model. The instruction transitions of βPx86$_{\text{view}}$ remain consistent with those of αPx86$_{\text{view}}$ (Fig. 4.3). The control flow transitions are updated as follows (Fig 4.4).

(PROGRAM-NORMAL)

$$\frac{\begin{array}{c} pc(t) = i \qquad \Pi(t,i) = \alpha\ \mathbf{goto}\ j \\ \langle \mathbb{T}(t), M\rangle \xrightarrow{\alpha} \langle T', M'\rangle \\ pc' = pc[t \mapsto j] \qquad \mathbb{T}' = \mathbb{T}[t \mapsto T'] \end{array}}{\langle pc, \mathsf{false}, \mathbb{T}, M, G\rangle \Rightarrow_{\Pi} \langle pc', \mathsf{false}, \mathbb{T}', M', G\rangle}$$

(PROGRAM-IF)

$$\frac{\begin{array}{c} pc(t) = i \qquad \Pi(t,i) = \mathbf{if}\ B\ \mathbf{goto}\ j\ \mathbf{else\ to}\ k \\ pc' = pc\left[t \mapsto \begin{cases} j & \mathbb{T}(t).\mathsf{regs}(B) = \mathsf{true} \\ k & \mathbb{T}(t).\mathsf{regs}(B) = \mathsf{false} \end{cases}\right] \end{array}}{\langle pc, \mathsf{false}, \mathbb{T}, M, G\rangle \Rightarrow_{\Pi} \langle pc', \mathsf{false}, \mathbb{T}, M, G\rangle}$$

(PROGRAM-GHOST)

$$\frac{\begin{array}{c} pc(t) = i \qquad \Pi(t,i) = \langle \alpha\ \mathbf{goto}\ j, \hat{a} := \hat{e}\rangle \\ \langle \mathbb{T}(t), M\rangle \xrightarrow{\alpha} \langle T', M'\rangle \\ pc' = pc[t \mapsto j] \qquad \mathbb{T}' = \mathbb{T}[t \mapsto T'] \\ G' = G[\hat{a} \mapsto G(\hat{e})] \end{array}}{\langle pc, \mathsf{false}, \mathbb{T}, M, G\rangle \Rightarrow_{\Pi} \langle pc', \mathsf{false}, \mathbb{T}', M', G'\rangle}$$

(CRASH)

$$\frac{\begin{array}{c} T.\mathsf{coh} = (\lambda x.0) \qquad T.\mathsf{v}_{\mathsf{rNew}} = 0 \\ T.\mathsf{v}_{\mathsf{pReady}} = 0 \qquad T.\mathsf{v}_{\mathsf{pAsync}} = (\lambda x.0) \\ T.\mathsf{v}_{\mathsf{pCommit}} = (\lambda x.0) \qquad \forall x \in \textsc{Loc}.\ CM(x) \in [x]^{\mathsf{P}}(\langle \mathbb{T}, M\rangle) \\ pc' = pc[syst \mapsto Rec_{pending}] \end{array}}{\langle pc, \mathsf{false}, \mathbb{T}, M, G\rangle \Rightarrow_{\Pi} \langle pc', \mathsf{true}, \lambda t \in \textsc{Tid}.T, \langle CM\rangle, G\rangle}$$

Figure 4.4: Updated control flow transitions of Px86$_{\text{view}}$ for a program Π

4.1.2.3 Modelling Crashes

Crashes under the αPx86$_{view}$ model: The above operational definitions naturally induce a notion of a execution (or a "run") of Px86$_{\text{view}}$ on a certain program Π starting from some initial state of the form $\langle \lambda t.\, \iota, \mathbb{T}, M, G\rangle$. A system crash might occur at any point during the execution. Again, following the model of [32], the persistent memory (PM) is not modeled as a concrete part of the state. Instead, the possible contents of the PM can be inferred from the machine state (specifically from the memory and the $\mathsf{v}_{\mathsf{pCommit}}$ views of the different threads), as defined next.

Definition 4.1.5. A persistent memory $PM : \textsc{Loc} \to \textsc{Val}$ is *possible in a state* σ if for every $x \in \textsc{Loc}$, there exists some ts such that $\sigma.M[ts] = \langle x := PM(x)\rangle$ and $x \notin \sigma.M(ts..\sigma.\mathsf{maxpCommit}(x)]$.

Notice that the machine state of Px86$_{\text{view}}$ does not include any component designated to recovery. This is because the Px86 semantics adequately describe the state immediately after a system crash, but do not address recovery processes. As a result, PIEROGI$_{\text{simp}}$ is limited to reasoning about program states up until a crash and its immediate aftermath.

Crashes under the βPx86$_{view}$ model: The CRASH transition (Fig. 4.4) creates a new initial message and resets the views of each thread. This allows reasoning about programs *after* a crash, which prior works [24, 32], do not handle. Specifically, the state CM is *possible* immediately after a crash in state σ if for every $x \in \textsc{Loc}$, there exists some ts such that $\sigma.M[ts] \equiv \langle x := CM(x) \rangle$ and $x \notin \sigma.M(ts..\sigma.\mathsf{maxpCommit}(x)]$. To formalise this, we use the *persistent memory view* expression (see §4.2.1.2). We adopt the same assumption for recovery as in Chapter 3. Specifically, we assume that recovery is executed by a system unique system thread *syst* that is different from any program thread. Recovery is only enabled in state σ if $\sigma.rec$ holds. Moreover, we assume a special label $Rec_{pending}$, which represents the label of the first recovery instruction. Upon completion of the recovery procedure, the pc_{syst} is set to $Rec_{complete}$, and we assume that there is a transition from this state to a state in which *rec* is set to false.

4.2 The Pierogi Proof Rules and Reasoning Principles

We proceed with a description of our verification framework. As with prior work [39], the view-based semantics for persistent TSO [32] allows us to use the standard Owicki–Gries rules [8,115] for compound statements. The main adjustment is the introduction of a new specialised assertion language capable of expressing properties about the different "views" described intuitively in §2.2.2. As such, since view updates are highly non-deterministic, the standard "assignment axiom" of Hoare Logic (and by extension Owicki–Gries) is no longer applicable. Moreover, unlike SC, reads in a weak memory setting have a side-effect: their interaction with the memory location being read causes the view of the executing thread to advance. Therefore, we resort to a set of proof rules that describe how views are modified and manipulated, as formalised by our view-based assertions.

4.2.1 View-Based Expressions

In this section we will discuss the view-based expressions /assertions of Pierogi. As with prior work on the RC11 model [88], we interpret Pierogi expressions directly over a view-based state. We use expressions tailored for the view-based Px86$_{\text{view}}$ model [32], which allow us to express relationships between different system components, including the persistent memory.

4.2.1.1 Informal Description of View-Based Expressions

In this section we will discuss the view-based expressions of Pierogi$_{\text{simp}}$ and Pierogi$_{\text{full}}$ collectively. Our expressions fall into one of five categories: 1) *current view* expressions, which describe the current views of different system components (e.g. the persistent view); 2) *conditional view* expressions [39], which describe a view on a location after reading a particular value on a *different* location; 3) *last view* expressions, which hold if a component is viewing the last write to a location; 4) *last entry* expressions, which return the timestamp of a memory message before a given limit and 5) *write-count* expressions, which describe the number of writes to a location.

***Current view expressions*:** Our current view expressions comprise $[x]_t$, $[x]^{\mathsf{P}}$ and $[x]^{\mathsf{A}}_t$, as described below; as shown in §2.2.2, each of these expressions describes a *set* of possible values.

$[x]_t$ denotes the *thread view* of thread t: the set of values t may read for x.

$[x]^{\mathsf{P}}$ denotes the *persistent memory view*: the set of values that x may hold in (persistent) memory.

$[x]_t^{\mathsf{A}}$ denotes the *asynchronous memory view* of thread t: the set of values that can be persisted after a barrier instruction (**sfence**/**mfence**/RMW) is executed by t (see rule OP in Fig. 4.7). Asynchronous views are updated after executing a $\mathbf{flush}_{\text{opt}}$; however, unlike persistent memory views, the values in asynchronous views are not guaranteed to persist until a subsequent barrier is executed by the same thread.

***Conditional view expression*:** The conditional view expression is of the form $\langle x, v\rangle[y]_t$, and captures the crux of message passing.

$\langle x, v\rangle[y]_t$ returns a set of values that t may read for y after it reads value v for x. In particular, if $\langle x, v\rangle[y]_t = S$ holds for some set S and t executes $a := \mathbf{load}\, x$, then in the state immediately after the load, if $a = v$, then $[y]_t \subseteq S$ (see LP_2 in Fig. 4.7).

***Last-view expressions*:** Last-view expressions (*cf.* [48]) are boolean-valued and hold if a particular component is synchronized (i.e. observes the latest value) on the given location. Such expressions provide determinism guarantees on **load** and **flush**. For instance, if the view of t is the last write on x, then a read from x by t will load this last value.

$[\![x]\!]_t$ holds iff t is currently viewing the *last* write to x. Thus, for example, if $[\![x]\!]_t$ holds, then a **load** from x by t reads the last write to x. Note that unlike architectural operational models [131], in the view model [32], writes are visible to all threads as soon as they occur.

$[\![x]\!]_t^{\mathsf{F}}$ holds iff a **flush** of x by t is guaranteed to flush the *last* write to x to persistent memory.

$[\![x]\!]_t^{\mathsf{M}}$ holds iff after performing an **mfence** the *thread view* of t will only contain the last write to x, thus a future read at x will definitely return the last write to x.

$[\![x : v]\!]$ holds iff the last written value to x is v.

***Last-entry expressions*:** Last-entry expressions return the timestamp of the memory message with a location equal to the given location and a timestamp less or equal to the given limit.

$[\![x]\!]_t$ holds iff t is currently viewing the *last* write to x. Thus, for example, if $[\![x]\!]_t$ holds, then a **load** from x by t reads the last write to x. Note that unlike architectural operational models [131], in the view model [32], writes are visible to all threads as soon as they occur.

$\mathsf{LE}(x)$ returns the timestamp of the last memory message on location x.

$\mathsf{LE}(ts, x)$ returns the timestamp of the last memory message on location x, before the timestamp ts.

$\mathsf{LE}_{\mathsf{coh}}(y, t, x)$ returns the timestamp of the last memory message on location x before the timestamp that corresponds to the coherence view of thread t for location y.

$\vec{x}$ returns the value of the last write at location x.

***Write-count expression*:** Lastly, the write-count expression is of the form $|x, v|$, as described below. This expression is useful for inferring view expressions from known facts about the number of writes in the system with a particular value (see Fig. 5.8).

$|x, v|$ returns the number of writes to x with value v. If $|x, v| = n$ and t writes to $y \neq x$, or writes a value $u \neq v$, then $|x, v| = n$ continues to hold afterwards.

4.2.1.2 The Semantics of Pierogi Expressions

***The semantics of* $\textsc{Pierogi}_{simp}$ *expressions*:** We now present the formal definitions of the subset of the expressions introduced in §4.2.1 that belongs to $\textsc{Pierogi}_{\text{simp}}$, in terms of $\alpha\text{Px86}_{\text{view}}$ machine states. When formalizing *view* expressions, we start with auxiliary functions that return the sets of observable timestamps visible to the components in question, then extract the values in memory corresponding to these timestamps. To facilitate this, we define:

$$\mathsf{MemLoc}(x, ts, M) \triangleq \textbf{if } (ts = 0) \textbf{ then } x \textbf{ else } M[ts].\mathsf{loc} \tag{4.1}$$

$$\mathsf{MemVal}(ts, M) \triangleq \textbf{if } (ts = 0) \textbf{ then } 0 \textbf{ else } M[ts].\mathsf{val} \tag{4.2}$$

$$\mathsf{Vals}(M, TS) \triangleq \{\mathsf{MemVal}(ts, M) \mid ts \in TS\} \tag{4.3}$$

where $M \in \textsc{Memory}$, $x \in \textsc{Loc}$ and TS is a set of timestamps. Def. 4.1 returns the location of the message that corresponds to a given timestamp ts in memory M. If the given timestamp is 0, then the corresponding message is the initial message, and thus, the returned location is equal to the location provided as an argument, x. Likewise, Def. 4.2 returns the value of the message that corresponds to a given timestamp ts in memory M. If the corresponding message is the initial, then the returned value is 0. Finally, Def. 4.3 returns the values at the given set of timestamps, TS.

Thread view: To define the meaning of the thread view expression, $[x]_t$, we use:

$$\mathsf{TS}^{\mathsf{OF}}_{\mathsf{t}}(\sigma, x, ts) \triangleq \{ts' \mid \mathsf{MemLoc}(x, ts', \sigma.M) = x \wedge \sigma.\mathbb{T}(t).\mathsf{coh}(x) \leq ts' \wedge x \notin \sigma.M(ts'..ts]\}$$
$$\mathsf{TS}_{\mathsf{t}}(\sigma, x) \triangleq \mathsf{TS}^{\mathsf{OF}}_{\mathsf{t}}(\sigma, x, \sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{rNew}})$$

$\mathsf{TS}^{\mathsf{OF}}_{\mathsf{t}}(\sigma, x, ts)$ returns the set of *timestamps* that are *observable from* timestamp ts for thread t to read for location x in state σ; and $\mathsf{TS}_{\mathsf{t}}(\sigma, x)$ returns the set of *timestamps* that are *observable* for t to read x in σ. Note that after instantiating t to $\sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{rNew}}$ in $\mathsf{TS}^{\mathsf{OF}}_{\mathsf{t}}(\sigma, x, ts)$, we obtain the premises of the load rules in Fig. 4.3. Then, $[x]_t \triangleq \lambda\sigma.\, \mathsf{Vals}(\sigma.M, \mathsf{TS}_{\mathsf{t}}(\sigma, x))$, i.e. is the set of values in $\sigma.M$ corresponding to the timestamps in $\mathsf{TS}_{\mathsf{t}}(\sigma, x)$.

Persistent memory view: For the persistent memory view expression, $[x]^{\mathsf{P}}$, we use:

$$\mathsf{TS}^{\mathsf{P}}(\sigma, x) = \{ts \mid \mathsf{MemLoc}(x, ts, \sigma.M) = x \wedge x \notin \sigma.M(ts..\sigma.\mathsf{maxpCommit}(x)]\}$$

which returns the set of *timestamps* that are observable to the *persistent memory* for x in σ. Then, $[x]^{\mathsf{P}} \triangleq \lambda\sigma.\, \mathsf{Vals}(\sigma.M, \mathsf{TS}^{\mathsf{P}}(\sigma, x))$. Note that the second conjunct within the definition of $\mathsf{TS}^{\mathsf{P}}(\sigma, x)$ is precisely the condition that links Px86$_{\text{view}}$ states to PM states (Def. 4.1.5). Given this definition, we have:

Proposition 1. A persistent memory $PM : \textsc{Loc} \to \textsc{Val}$ is possible in a state σ iff $PM(x) \in [x]^{\mathsf{P}}(\sigma)$ for every $x \in \textsc{Loc}$.

Asynchronous memory view: To define the meaning of the asynchronous memory view, $[x]^{\mathsf{A}}_t$, we use:

$$\mathsf{TS}^{\mathsf{A}}_{\mathsf{t}}(\sigma, x) \triangleq \{ts \mid \mathsf{MemLoc}(x, ts, \sigma.M) = x \wedge x \notin \sigma.M(ts..\sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{pAsync}}(x)]\}$$

which returns the timestamps of the asynchronous view of thread t in location x and state σ. Then, as before, $[x]^{\mathsf{A}}_t \triangleq \lambda\sigma.\, \mathsf{Vals}(\sigma.M, \mathsf{TS}^{\mathsf{A}}_{\mathsf{t}}(\sigma, x))$.

Conditional view: The functions used to define conditional memory view, $\langle x, v\rangle[y]_t$, are slightly more sophisticated than those above. We define:

$$\mathsf{TS}^{\mathsf{OV}}_{\mathsf{t}}(\sigma, x, v) \triangleq \left\{ ts' \;\middle|\; \begin{array}{l} \exists ts \in \mathsf{TS}_{\mathsf{t}}(\sigma, x).\ \mathsf{MemVal}(ts, \sigma.M) = v \wedge \\ \quad ts' = \textbf{if}\ ts = \sigma.\mathbb{T}(t).\mathsf{coh}(x)\ \textbf{then}\ \sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{rNew}} \\ \qquad\qquad \textbf{else}\ ts \sqcup \sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{rNew}} \end{array} \right\}$$

$$\mathsf{TS}^{\mathsf{CO}}_{\mathsf{t}}(\sigma, x, v, y) \triangleq \bigcup\{\mathsf{TS}^{\mathsf{OF}}_{\mathsf{t}}(\sigma, y, ts) \mid ts \in \mathsf{TS}^{\mathsf{OV}}_{\mathsf{t}}(\sigma, x, v)\}$$

where $\mathsf{TS}^{\mathsf{OV}}_{\mathsf{t}}(\sigma, x, v)$ returns the set of timestamps that t can observe for x with value v. Assuming ts is a timestamp that t can observe for x, and the value for x at ts is v, the corresponding timestamp ts' that $\mathsf{TS}^{\mathsf{OV}}_{\mathsf{t}}(\sigma, x, v)$ returns is $\sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{rNew}}$ if t's coherence view for x is ts, and the maximum of ts and $\sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{rNew}}$, otherwise. Given this, $\mathsf{TS}^{\mathsf{CO}}_{\mathsf{t}}(\sigma, x, v, y)$ returns the timestamps that t can observe for y, from any timestamp $ts \in \mathsf{TS}^{\mathsf{OV}}_{\mathsf{t}}(\sigma, x, v)$. Finally, the set of conditional values is defined by $\langle x, v\rangle[y]_t \triangleq \lambda\sigma.\, \mathsf{Vals}(\sigma.M, \mathsf{TS}^{\mathsf{CO}}_{\mathsf{t}}(\sigma, x, v, y))$.

Last view expressions: We use the following auxiliary definition:

$$\mathsf{LE}(M, x) \triangleq \bigsqcup\{ts \mid \mathsf{MemLoc}(x, ts, M) = x\}$$

which returns the timestamp of the last write to x in M. Then, the last view assertions are given by:

- $[\![x]\!]_t \triangleq \lambda\sigma.\ \mathsf{TS}_{\mathsf{t}}(\sigma, x) = \{\mathsf{LE}(\sigma.M, x)\}$, i.e. t's view of x in σ *is* the last write to x in σ.
- $[\![x]\!]^{\mathsf{F}}_t \triangleq \lambda\sigma.\ \mathsf{LE}(\sigma.M, x) \leq \sigma.\mathbb{T}(t).\mathsf{maxcoh} \sqcup \sigma.\mathsf{maxpCommit}(x)$, i.e. the maximum of t's maximum coherence view and the maximum commit view of x (over all threads) is beyond the last write to x in σ. This means that executing a **flush** x operation in t will cause the last write of x to be flushed (see FLUSH rule in Fig. 4.3).
- $[\![x]\!]^{\mathsf{M}}_t \triangleq \lambda\sigma.\ \mathsf{LE}(\sigma.M, x) \leq \sigma.\mathbb{T}(t).\mathsf{maxcoh}$ i.e. the t's maximum coherence view is beyond the last write to x in σ. This means that executing a **mfence** operation in t will cause the thread view of x to contain of the last write to x. In this way, a future read of x is guaranteed to return the last write to x. (see MFENCE rule in Fig. 4.3).

- $\llbracket x : v \rrbracket \triangleq \lambda\sigma.\ \mathsf{MemVal}(\mathsf{LE}(\sigma.M, x), \sigma.M) = v$ i.e. the last written value to x is v.

Value-count expression: Finally, the value count expression is defined as follows:

$$|x, v| \triangleq \lambda\sigma.\, |\{ts \mid \mathsf{MemVal}(ts, \sigma.M) = v \wedge \mathsf{MemLoc}(x, ts, \sigma.M) = x\}|$$

The semantics of PIEROGI$_{full}$ expressions. PIEROGI$_{\text{full}}$ includes all the view expressions of PIEROGI$_{\text{simp}}$ with the addition of four more *last entry* expressions. To accommodate the fact that the first message of a βPx86$_{\text{view}}$ memory corresponds to a store (CM : LOC $\to$ VAL) which may not map all the memory locations to value 0, we update the auxiliary definitions *Memloc* (4.1), *MemVal* (4.2) and *Vals* (4.3) as follows:

$$\mathsf{MemLoc}(x, ts, M) \triangleq \textbf{if } (t = 0) \textbf{ then } x \textbf{ else } M[ts].\mathsf{loc} \tag{4.4}$$

$$\mathsf{MemVal}(x, ts, M) \triangleq \textbf{if } (t = 0) \textbf{ then } M[0](x) \textbf{ else } M[ts].\mathsf{val} \tag{4.5}$$

$$\mathsf{Vals}(TS, x, M) \triangleq \{\mathsf{MemVal}(x, ts, M) \mid ts \in TS\} \tag{4.6}$$

All the PIEROGI$_{\text{simp}}$ view expressions introduced above retain their structure. However, when utilized, *MemLoc*, *MemVal*, and *Vals* refer to equations 4.4, 4.5, and 4.6, respectively.

Next, we present the formalization of the four additional *last entry* expressions of PIEROGI$_{\text{full}}$, namely $\mathsf{LE}(x)$, $\mathsf{LE}(t, x)$, $\mathsf{LE}(x)$ and $\vec{x}$. These expressions enable reasoning about written values before a given timestamp.

Their formal definition starts with the following auxiliary definition:

$$\mathsf{MemLastEntryLim}(x, t, M) \triangleq \bigsqcup\{ts \mid \mathsf{MemLoc}(x, t, M) = x \wedge x \leq t\},$$

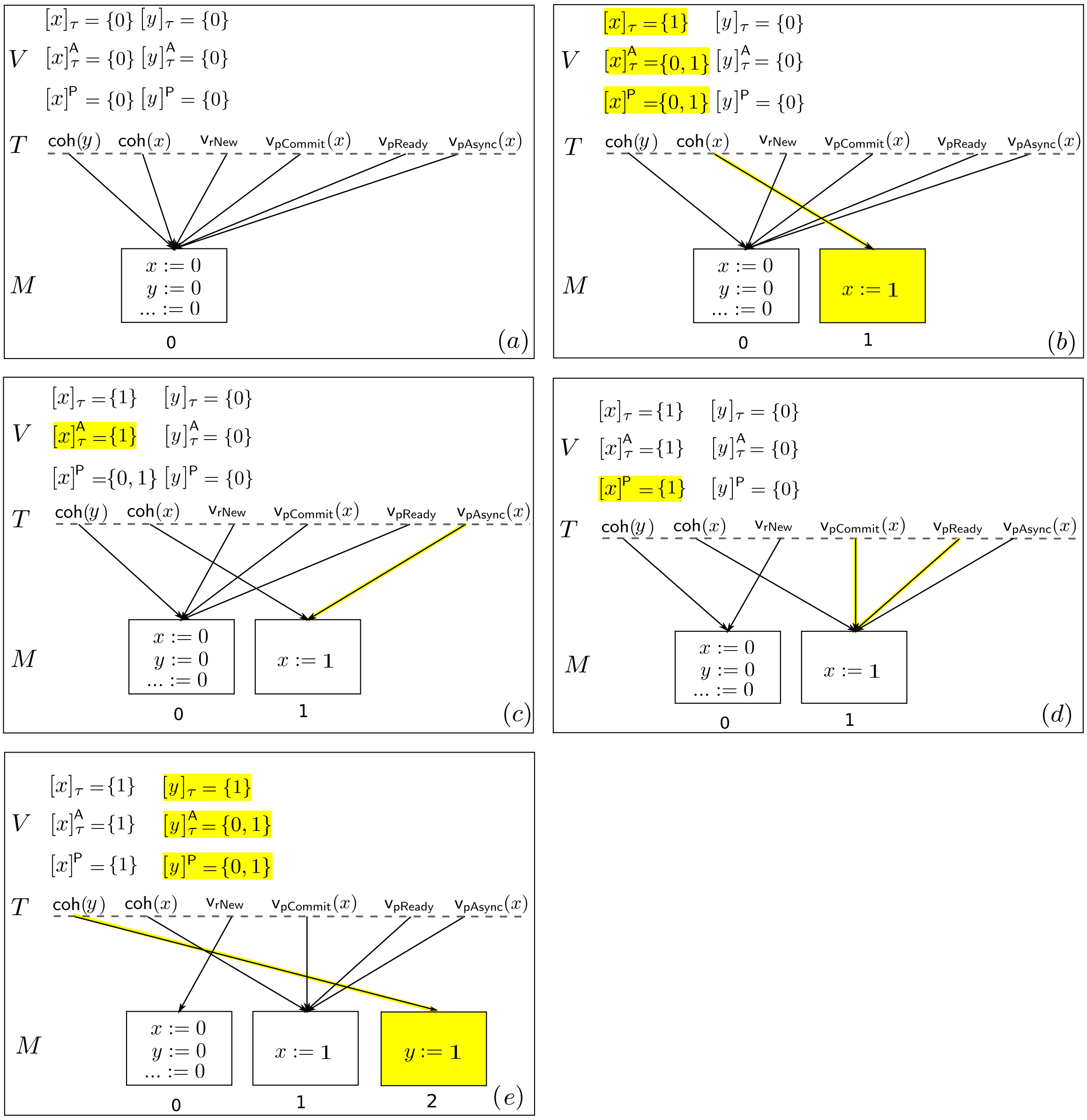
which, assuming $t \in [0, |M|)$ returns the maximum timestamp of the memory messages with location x and timestamp less or equal to timestamp t.

$$\begin{aligned}
\mathsf{LE}(x) &\triangleq \lambda\sigma.\, \mathsf{MemLastEntryLim}(x, |\sigma.M| - 1, \sigma.M) \\
\mathsf{LE}(t, x) &\triangleq \lambda\sigma.\, \mathsf{MemLastEntryLim}(x, t, \sigma.M) \\
\mathsf{LE}_{\mathsf{coh}}(y, t, x) &\triangleq \lambda\sigma.\, \mathsf{MemLastEntryLim}(x, \sigma.\mathbb{T}(t).\mathsf{coh}(y), \sigma.M)
\end{aligned}$$

Finally, the expression $\vec{x}$ returns the value of the last write at location x in the given state:

$$\vec{x}.\sigma \triangleq \mathsf{MemVal}(x, \mathsf{LE}(x).\sigma, \sigma.M)$$

4.2.1.3 A Program Execution Example


V
$[x]_\tau = \{0\}$ $[y]_\tau = \{0\}$
$[x]^{\mathsf{A}}_\tau = \{0\}$ $[y]^{\mathsf{A}}_\tau = \{0\}$
$[x]^{\mathsf{P}} = \{0\}$ $[y]^{\mathsf{P}} = \{0\}$
T
coh(y) coh(x) v$_{\mathsf{rNew}}$ v$_{\mathsf{pCommit}}$(x) v$_{\mathsf{pReady}}$ v$_{\mathsf{pAsync}}$(x)
M
$x := 0$
$y := 0$
$... := 0$
0
(a)
$[x]_\tau = \{1\}$ $[y]_\tau = \{0\}$
$[x]^{\mathsf{A}}_\tau = \{0, 1\}$ $[y]^{\mathsf{A}}_\tau = \{0\}$
$[x]^{\mathsf{P}} = \{0, 1\}$ $[y]^{\mathsf{P}} = \{0\}$
$x := 1$
1
(b)
$[x]_\tau = \{1\}$ $[y]_\tau = \{0\}$
$[x]^{\mathsf{A}}_\tau = \{1\}$ $[y]^{\mathsf{A}}_\tau = \{0\}$
$[x]^{\mathsf{P}} = \{0, 1\}$ $[y]^{\mathsf{P}} = \{0\}$
(c)
$[x]_\tau = \{1\}$ $[y]_\tau = \{0\}$
$[x]^{\mathsf{A}}_\tau = \{1\}$ $[y]^{\mathsf{A}}_\tau = \{0\}$
$[x]^{\mathsf{P}} = \{1\}$ $[y]^{\mathsf{P}} = \{0\}$
(d)
$[x]_\tau = \{1\}$ $[y]_\tau = \{1\}$
$[x]^{\mathsf{A}}_\tau = \{1\}$ $[y]^{\mathsf{A}}_\tau = \{0, 1\}$
$[x]^{\mathsf{P}} = \{1\}$ $[y]^{\mathsf{P}} = \{0, 1\}$
$y := 1$
2
(e)

(a)
store x 1; (b)
flush$_{\text{opt}}$ x; (c)
sfence; (d)
store y 1 (e)

Figure 4.5: A depiction of a subset of the *current views*, the thread state (T), and Px86$_{\text{view}}$ memory list (M) after the execution of each instruction of the program below. Its message of the memory is indexed with a timestamp. The first message corresponds to the *initial message* of the memory. The highlighted components of the state capture the effects of the preceding instruction.

Fig. 4.5 illustrates how the *view* components (see Table in 4.1.2.1) of the thread state T as well as the *thread view*, *persistent memory view* and *asynchronous memory view* expressions are changing during the execution of program Fig. 2.7d. Its execution state is denoted with a letter, starting with a for the initial state. In the beginning, the memory $\sigma.M$ includes only the *initial message*, and all the view components of t's thread state are 0 (pointing to the initial message). The set of view expressions for thread t illustrated in V contain only the value 0. After the execution of **store** x 1 the message $\langle x := 1\rangle$ is added to the memory. The store transition causes the coherence view of t for x ($\sigma.\mathbb{T}(t).\mathsf{coh}(x)$) to become 1. This impacts the *thread view* expression of x for t which now contains the value 1. Since neither the $\sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{pCommit}}(x)$ nor the $\sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{pAsync}}(x)$ view alters, the *persistent memory view* for x and the *asynchronous memory view* of t for x continues to contain the value of the initial message for x (0) as well as the values of the messages with location x that follows it (1). The view expressions in V of thread t for location y remain the same. The execution of the instruction $\textbf{flush}_{\text{opt}}$ x causes the $\sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{pAsync}}(x)$ to point to message 1. As a result, the *asynchronous memory view* of x for t is updated to contain only the value 1. The **sfence** instruction causes the $\sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{pCommit}}(x)$ view for t to point to the same message as the $\sigma.\mathbb{T}(t).\mathsf{v}_{\mathsf{pAsync}}(x)$ view. Thus the *persistent memory view* of t for x becomes equal to its *asynchronous memory view.* Finally, the execution of the instruction **store** y 1 adds to memory M the message $\langle y := 1\rangle$. All the views in T remain the same apart from the coherence view of y ($\sigma.\mathbb{T}(t).\mathsf{coh}(y)$) which now points to message 2. As a result the *thread view* of t for y is updated to contain only the new written value 1. The *persistent view* and the *asynchronous memory view* for y are updated to contain the values 0 and 1. The view expressions for location x remain the same as before.

4.2.2 Owicki–Gries Reasoning

We now present the $\textsc{Pierogi}_{\text{simp}}$ and $\textsc{Pierogi}_{\text{full}}$ proof systems, as extensions of Hoare Logic with Owicki–Gries reasoning to account for concurrency. The main differences between $\textsc{Pierogi}_{\text{simp}}$ /$\textsc{Pierogi}_{\text{full}}$ and standard Owicki–Gries are that 1) our program annotations contain view-based assertions that allow reasoning about weak and persistent memory behaviors; and 2) we define a crash invariant to describe the recoverable state of the program after a crash. As explained in the remainder of this section, the definition of the crash invariant differs for $\textsc{Pierogi}_{\text{simp}}$ and $\textsc{Pierogi}_{\text{full}}$. Our proof rules are *syntactic*, and thus can be understood and used without having to understand the details of the underlying $\text{Px86}_{\text{view}}$ model.

We let $\textsc{Assertion}_{\textsc{pv}}$ be the set of *assertions* (i.e. predicates over $\alpha\text{Px86}_{\text{view}}$ (resp. $\beta\text{Px86}_{\text{view}}$) states) that use view-based expressions (§ 4.2.1). A *proof outline* is a tuple $(\mathit{in}, \mathit{ann}, I, \mathit{fin})$, where $\mathit{in}, \mathit{fin} \in \textsc{Assertion}_{\textsc{pv}}$ are the initial and final assertions, I is the crash invariant and ann is an *annotation function* that models program annotations. Specifically, $\mathit{ann} \in \textsc{Ann} = \textsc{Tid} \times \textsc{Lab} \rightarrow \textsc{Assertion}_{\textsc{pv}}$, associates each program point (t, i) with its associated assertion. In the case of $\textsc{Pierogi}_{\text{simp}}$ the crash invariant, $I \in \textsc{Inv} \subset \textsc{Assertion}_{\textsc{pv}}$, is defined over persistent views only, i.e. it only comprises the persistent view expressions of the form $[x]^{\mathsf{P}}$. However, in the case of $\textsc{Pierogi}_{\text{full}}$ the crash invariant is defined over any view expression ($I \in \textsc{Assertion}_{\textsc{pv}}$). A *proof outline* is a tuple $(\mathit{in}, \mathit{ann}, I, \mathit{fin})$, where $\mathit{in}, \mathit{fin} \in \textsc{Assertion}_{\textsc{pv}}$ are the initial and final assertions.

Definition 4.2.1 ($\textsc{Pierogi}_{\textsc{simp}}$ Valid proof outline**).** A $\textsc{Pierogi}_{\text{simp}}$ proof outline $(\mathit{in}, \mathit{ann}, I, \mathit{fin})$ is *valid* for a program Π iff the following hold:

Initialisation. For all $t \in \textsc{Tid}$, $\mathit{in} \Rightarrow \mathit{ann}(t, \iota)$.

Finalisation. $(\bigwedge_{t \in \textsc{Tid}} \mathit{ann}(t, \zeta)) \Rightarrow \mathit{fin}$.

Local correctness. For all $t \in \textsc{Tid}$ and $i \in \textsc{Lab}$, either:

- $\Pi(t, i) = \alpha$ **goto** j and $\{\mathit{ann}(t, i)\}\ \alpha\ \{\mathit{ann}(t, j)\}$; or
- $\Pi(t, i) = $ **if** B **goto** j **else to** k and both $\mathit{ann}(t, i) \wedge B \Rightarrow \mathit{ann}(t, j)$ and $\mathit{ann}(t, i) \wedge \neg B \Rightarrow \mathit{ann}(t, k)$ hold; or
- $\Pi(t, i) = \langle \alpha$ **goto** $j, \hat{a} := \hat{e}\rangle$ and $\{\mathit{ann}(t, i)\}\ \alpha\ \{\mathit{ann}(t, j)[\hat{e}/\hat{a}]\}$.

Stability. For all $t_1, t_2 \in \textsc{Tid}$ such that $t_1 \neq t_2$ and $i_1, i_2 \in \textsc{Lab}$:

- if $\Pi(t_1, i_1) = \alpha$ **goto** j, then $\{\mathit{ann}(t_2, i_2) \wedge \mathit{ann}(t_1, i_1)\}\ \alpha\ \{\mathit{ann}(t_2, i_2)\}$;
- if $\Pi(t_1, i_1) = \langle \alpha$ **goto** $j, \hat{a} := \hat{e}\rangle$, then $\{\mathit{ann}(t_2, i_2) \wedge \mathit{ann}(t_1, i_1)\}\ \alpha\ \{\mathit{ann}(t_2, i_2)[\hat{e}/\hat{a}]\}$.

Persistence. There exists $t \in \textsc{Tid}$ such that for all $i \in \textsc{Lab}$, $\mathit{ann}(t, i) \Rightarrow I$.

Definition 4.2.2 ($\textsc{Pierogi}_{\textsc{full}}$ Valid proof outline). A $\textsc{Pierogi}_{\text{full}}$ proof outline $(\mathit{in}, \mathit{ann}, \mathit{fin}, I)$ is *valid* for a program Π iff the following hold:

Initialisation. For all $t \in \textsc{Tid}$, $\mathit{in} \Rightarrow I \wedge \mathit{ann}(t, \iota)$.

Finalisation. $I \wedge (\bigwedge_{t \in \textsc{Tid}} \mathit{ann}(t, \zeta)) \Rightarrow \mathit{fin}$.

Local correctness. For all $t \in \textsc{Tid}$ and $i \in \textsc{Lab}$, either:

- $\Pi(t, i) = \alpha$ **goto** j and $\{I \wedge \mathit{ann}(t, i)\}\ \alpha\ \{I \wedge \mathit{ann}(t, j)\}$; or
- $\Pi(t, i) = $ **if** B **goto** j **else to** k and both
  - $I \wedge \mathit{ann}(t, i) \wedge B \Rightarrow \mathit{ann}(t, j)$ and
  - $I \wedge \mathit{ann}(t, i) \wedge \neg B \Rightarrow \mathit{ann}(t, k)$ hold; or
- $\Pi(t, i) = \langle \alpha$ **goto** $j, \hat{a} := \hat{e}\rangle$ and $\{I \wedge \mathit{ann}(t, i)\}\ \alpha\ \{(I \wedge \mathit{ann}(t, j))[\hat{e}/\hat{a}]\}$.

Stability. For all $t_1, t_2 \in \textsc{Tid}$ such that $t_1 \neq t_2$ and $i_1, i_2 \in \textsc{Lab}$:

- if $\Pi(t_1, i_1) = \alpha$ **goto** j, then $\{I \wedge \mathit{ann}(t_2, i_2) \wedge \mathit{ann}(t_1, i_1)\}\ \alpha\ \{\mathit{ann}(t_2, i_2)\}$;
- if $\Pi(t_1, i_1) = \langle \alpha$ **goto** $j, \hat{a} := \hat{e}\rangle$, then $\{I \wedge \mathit{ann}(t_2, i_2) \wedge \mathit{ann}(t_1, i_1)\}\ \alpha\ \{\mathit{ann}(t_2, i_2)[\hat{e}/\hat{a}]\}$.

Persistence. Both of the following hold:

- for all $\alpha \in \mathit{Recovery}$, $\{I\}\ \alpha\ \{I\}$
- $\{I\}\ \mathit{crash}\ \{I\}$

For both Def. 4.2.1 and Def. 4.2.2 Initialisation (resp. Finalisation) ensures that the initial (resp. final) assertion of each thread holds in the initial (resp. final) state. Local correctness ensures the validity of the program annotation of each thread, while global correctness ensures the stability of the program annotation of each thread under the execution of other threads. In essence, the local correctness proof for a thread t checks for each atomic statement of t if its post-condition (given as annotation) can be established by its pre-condition (given as annotation). Similarly, the global correctness proof for a thread t checks that the pre-condition of each atomic statement of t is stable against the atomic statements of the other threads. Persistence in the case of $\alpha\text{Px86}_{\text{view}}$ ensures that the crash invariant holds at every program point for some thread, excluding the crash and recovery event, which do not correspond to operational transitions. Persistence in the case of $\beta\text{Px86}_{\text{view}}$ ensures that the crash invariant holds at every program point for some thread, including the crash transition and the recovery process.

4.2.3 Pierogi Proof Rules ()

One of the main benefits of Pierogi is the ability to perform proofs at a high level of abstraction. In this section, we provide the set of proof rules that we use. The annotation within a proof outline is, in essence, an invariant mapping of each program location to an assertion that holds at the program location. Thus, we prove local correctness by checking that each atomic step of a thread establishes the assertions in that thread. Similarly, we check stability by checking each assertion in one thread against each atomic step of the other threads. To enable proof abstraction, we introduce a set of proof rules that describe the interaction between the assertions from §4.2.1 and the atomic program steps. We will use the standard decomposition rules from Hoare Logic to reduce proof outlines and enable our rules over atomic steps to be applied.

In summary, to show that a proof outline is valid under Pierogi$_{\text{simp}}$ (Def. 4.2.1) or Pierogi$_{\text{full}}$ (Def. 4.2.2) we use two types of rules: standard decomposition rules and rules for atomic statements and View-Based Assertions. Furthermore, when needed, we use a well-formedness condition which is common for both Pierogi$_{\text{simp}}$ and Pierogi$_{\text{full}}$. In the remainder of this section, we provide a summary of these rules.

Standard Decomposition Rules. The standard decomposition rules we use are given in Fig. 4.6, which allow one to weaken preconditions and strengthen postconditions, and decompose conjunctions and disjunctions.

$$\textsf{Cons}\,\frac{P' \Rightarrow P \quad Q \Rightarrow Q' \qquad \{P\}\ \Pi\ \{Q\}}{\{P'\}\ \Pi\ \{Q'\}} \qquad \textsf{Conj}\,\frac{\{P_1\}\ \Pi\ \{Q_1\} \qquad \{P_2\}\ \Pi\ \{Q_2\}}{\{P_1 \wedge P_2\}\ \Pi\ \{Q_1 \wedge Q_2\}} \qquad \textsf{Disj}\,\frac{\{P_1\}\ \Pi\ \{Q_1\} \qquad \{P_2\}\ \Pi\ \{Q_2\}}{\{P_1 \vee P_2\}\ \Pi\ \{Q_1 \vee Q_2\}}$$

Figure 4.6: Standard decomposition rules of Pierogi

Rules for Atomic Statements and View-Based Assertions. Weak and persistent memory models (e.g. Px86) are inherently non-deterministic. Moreover in contrast to sequential consistent, in view-based operational semantics (such as Px86$_{\text{view}}$) instructions such as $a := \textbf{load}\, e$ may have a side-effect since they may update the view of the thread performing the **load** (*cf.* [39]). Therefore, unlike Hoare Logic, which contains a single rule for assignment, we have a set of rules for atomic statements, describing their interaction with view-based assertions. Each of the rules in this section has been proven sound with respect to the view-based semantics in Isabelle/HOL. ***Pierogi$_{simp}$* rules for atomic atomic statements and view-based expressions:** A selection of the Pierogi$_{\text{simp}}$ rules for the atomic statements is given in Fig. 4.7, where the statement is assumed to be executed by thread t. The first column contains the pre/post condition triple, the second any additional constraints and the third, labels that we use to refer to the rules in our descriptions below. Unless explicitly mentioned as a constraint, we do not assume that threads, locations and values are distinct; e.g. rule $\textsf{LP}_3$ (referring to t and t') holds regardless of whether $t = t'$ or not.

The rules in Fig. 4.7 provide high-level insights into the low-level semantics of Px86$_{\text{view}}$ without having to understand the operational details. The $\textsf{LP}_\textsf{i}$ rules are for statement $a := \textbf{load}\, x$. Rule $\textsf{LP}_1$ states that if t's thread view of x is the set of values S, then in the post state a is an element of S and moreover t's thread view of x is a subset of S (since t's thread view may have shifted). By $\textsf{LP}_2$, provided the conditional view of t on y (with condition $x = u$) is S, if the load returns value u, then the view of t is

Precondition	Statement	Postcondition	Const.	Ref.
$\{[x]_t = S\}$	$a := \mathbf{load}\, x$	$\{a \in S \wedge [x]_t \subseteq S\}$		LP_1
$\{u \in [x]_t \Rightarrow \langle x, u\rangle [y]_t = S\}$		$\{a = u \Rightarrow [y]_t \subseteq S\}$		LP_2
$\{\lvert x, u\rvert = 1 \wedge [x]_{t'} = \{u\}\}$		$\{a = u \Rightarrow [x]_t = \{u\}\}$		LP_3
$\{true\}$	$\mathbf{store}\ x\ v$	$\{[x]_t = \{v\}\}$		SP_1
$\{[x]_{t'} = S\}$		$\{[x]_{t'} = S \cup \{v\}\}$	$t \neq t'$	SP_2
$\{[x]_{t'}^{\mathsf{A}} = S\}$		$\{[x]_{t'}^{\mathsf{A}} = S \cup \{v\}\}$		SP_3
$\{[x]^{\mathsf{P}} = S\}$		$\{[x]^{\mathsf{P}} = S \cup \{v\}\}$		SP_4
$\{[y]_t = S \wedge v \notin [x]_{t'}\}$		$\{\langle x, v\rangle [y]_{t'} \subseteq S\}$	$t \neq t'$	SP_5
$\{true\}$		$\{\llbracket x \rrbracket_t \wedge \llbracket x \rrbracket_t^{\mathsf{F}} \wedge \llbracket x \rrbracket_t^{\mathsf{M}}\}$		SP_6
$\{\lvert x, v\rvert = n\}$		$\{\lvert x, v\rvert = n + 1\}$		SP_7
$\{true\}$		$\{\llbracket x : v \rrbracket\}$		SP_8
$\{[x]_t = S\}$	$\mathbf{flush}\ x$	$\{[x]^{\mathsf{P}} \subseteq S \wedge [x]_t^{\mathsf{A}} \subseteq S\}$		FP_1
$\{[x]^{\mathsf{P}} = S\}$		$\{[x]^{\mathsf{P}} \subseteq S\}$		FP_2
$\{\llbracket x \rrbracket_{t'} \wedge [x]_{t'} = \{u\} \wedge \llbracket x \rrbracket_t^{\mathsf{F}}\}$		$\{[x]^{\mathsf{P}} = \{u\}\}$		FP_3
$\{[x]_t = S \vee [x]_t^{\mathsf{A}} = S\}$	$\mathbf{flush}_{\mathrm{opt}}\ x$	$\{[x]_t^{\mathsf{A}} \subseteq S\}$		OP
$\{[x]_t^{\mathsf{A}} = S \vee [x]^{\mathsf{P}} = S\}$	$\mathbf{sfence}$	$\{[x]^{\mathsf{P}} \subseteq S\}$		SFP
$\{\llbracket x \rrbracket_{t'} \wedge \llbracket x \rrbracket_t^{\mathsf{M}} \wedge [x]_{t'} = \{u\}\}$	$\mathbf{mfence}$	$\{[x]_t = \{u\}\}$		MFP_1
$\{[x]_t = S\}$		$\{[x]_t \subseteq S\}$		MFP_2
$\{[x]_t^{\mathsf{A}} = S \vee [x]^{\mathsf{P}} = S\}$		$\{[x]^{\mathsf{P}} \subseteq S\}$		MFP_3
$\{[y]_{t'} = S\}$	$a := \mathbf{CAS}\ x\ e_1\ e_2$	$\{[y]_{t'} \subseteq S\}$	$x \neq y$	CP_1
$\{[x]_{t'}^{\mathsf{A}} = S\}$		$\{a \Rightarrow [x]_{t'}^{\mathsf{A}} = S \cup \{e_2\}\}$		CP_2
$\{true\}$		$\{a \Rightarrow e_2 \in [x]^{\mathsf{P}}\}$		CP_3
$\{true\}$		$\{(a \Rightarrow (\llbracket x : e_2 \rrbracket \wedge [x]_t = \{e2\}))\}$		CP_4
$\{\llbracket x : v \rrbracket\}$		$\{\llbracket x : e_2 \rrbracket \vee \llbracket x : v \rrbracket\}$		CP_5
$\{\llbracket x : v \rrbracket\}$		$\{\neq a\}$	$v \neq e_1$	CP_6

Figure 4.7: Selected proof rules for atomic statements executed by thread t

shifted so that $[y]_t \subseteq S$. We only have $[y]_t \subseteq S$ in the postcondition because there may be multiple writes to x with value u; reading x read may shift the view to the latter write, thus *reducing* the set of values that t can read for y. LP_3 describes conditions for a deterministic load by thread t. The precondition assumes that there is only one write to x with value u, that *some* thread t' sees the last write to x with value u. Then, if t reads u, its thread view of x is also constrained to just the set containing u.

The store rules, SP_i, reflect the fact that a new write modifies the views of the other threads as well as the persistent memory and asynchronous views. The first four rules describe the interaction of a **store** by thread t with current view assertions. By SP_1, the **store** ensures that the current view of t is solely the value v written by t. This is because, in Px86$_{\mathrm{view}}$, new writes are introduced by the executing thread, t, with a maximal timestamp (see STORE rule in Fig. 4.3), and t's view is updated to this new write. SP_2, SP_3 and SP_4 are similar, and assuming that the view (of another thread, persistent memory, and asynchronous view, respectively) in the pre-state is S, shows that the view in the post-state is $S \cup \{v\}$. Rule SP_5 allows one to *introduce* a conditional observation assertion $\langle x, v\rangle [y]_{t'}$ where $t' \neq t$. The pre-state of SP_5 assumes that t's view of y is the set S, and that t' cannot view value v for y. Rule SP_6 introduces last-view assertions for t after t performs a write to x, and finally SP_7 states that the number of writes to x with value v increases by 1 after executing **store** x v.

Rules FP_i describe the effect of **flush** x on the state. FP_1 states that, provided that the current view of t for x is the set of values S, after executing **flush** x, we are guaranteed that both the persistent view and asynchronous view of t for x are subsets of S. We obtain a subset in the post state since the Px86$_{\text{view}}$ semantics potentially move the persistent and asynchronous views forward. Similarly, by FP_2 if the current persistent view of x is S, then after executing **flush** x the persistent view will be a subset of S. Finally, FP_3 provides a mechanism for establishing a deterministic persistent view u for x. The precondition assumes that *some* thread's view of x is the last write with value u and that t's view is such that the flush is guaranteed to flush to this last write to x.

Rule OP describes how the asynchronous view of t in the postcondition of **flush**$_{\text{opt}}$ x is related to the current view of t and the asynchronous view in the precondition. Rule SFP describes the relationship between the persistent view in the postcondition and the asynchronous view and the persistent view in the precondition for an **sfence** instruction.

The mfence rules, MFP_i, describe how mfence affects the current views of the currently executing thread (t). Rule MFP_1 is analogous to rule FP_3, providing a mechanism for establishing a deterministic thread view value for x. Rule MFP_2 expresses the fact that after executing an **mfence** instruction, the thread view of t for any address x is a subset of what it was in the pre-state. Finally, by rule MFP_3, provided that the asynchronous view of t and the persistent view for x is the set of values S, after executing **mfence**, we are guaranteed that the persistent view for x is a subset of S.

Lastly, we demonstrate some selected rules regarding the **CAS** instruction. In the following, $a = \mathit{true}$ indicates a successful execution of **CAS**, while $a = false$ indicates a failed execution of **CAS**. Rule CP_1 states that given that $x \neq y$ the thread view of *some* thread t for x after executing a **CAS** instruction is a subset or equal the thread view it had for x in the pre-state. By CP_2, assuming that the asynchronous view of *some* thread t for x in the pre-state is S, its asynchronous view in the post-state is $S \cup \{e_2\}$. Rule CP_3 states that the successful execution of a **CAS** instruction on x guarantees that the newly written value on x (e_2) by **CAS** will be present in the persistent view of x. By rule CP_4, if a **CAS** on x by a thread t succeeds then the last written value on x becomes the value written by **CAS** (e_2) and the thead view of x for t is updated to contain only the value e_2. Rule CP_5, states that after executing a **CAS** on x either the last written value on x becomes e_2, indicating that the **CAS** succeeded or it remains the same as in the pre-state. Finally, by CP_6 given that in the pre-state, the last written value on x is different from the **CAS** argument e_1, the **CAS** definitely fails, and thus in the post-state $a = false$.

In our Isabelle/HOL development, we have also proved the stability of several assertions (see Fig. 4.8 for a selection). An assertion P is *stable* over a statement α executed by t iff $\{P\}$ α $\{P\}$ holds.

***PIEROGI$_{full}$* rules for atomic atomic statements and view-based expressions:**

Statement	Stable Assert.	Const.	Ref.
$a := \mathbf{load}\, x$	$\{[y]_{t'} = S\}$	$t \neq t'$	LS_1
	$\{[y]^{\mathsf{P}} = S\}$		LS_2
	$\{[y]^{\mathsf{A}}_{t'} = S\}$		LS_3
	$\{a = k\}$		LS_4
	$\{\llbracket y \rrbracket_{t'}\}$		LS_5
	$\{\llbracket y : v \rrbracket\}$		LS_6
	$\{\llbracket y \rrbracket^{\mathsf{M}}_{t'}\}$		LS_7
$\mathbf{flush}\ x$	$\{[y]_{t'} = S\}$		FS_1
	$\{[y]^{\mathsf{P}} = S\}$	$x \neq y$	FS_2
	$\{\llbracket y \rrbracket_{t'}\}$		FS_3
	$\{\llbracket y \rrbracket^{\mathsf{F}}_{t'}\}$		FS_4
	$\{\lvert y, v \rvert = n\}$		FS_5
	$\{\llbracket y : v \rrbracket\}$		FS_6
$\mathbf{sfence}$	$\{[x]_{t'} = S\}$		SFS_1
	$\{\lvert x, v \rvert = n\}$		SFS_2
	$\{[x]^{\mathsf{A}}_{t} = S\}$		SFS_3
$\mathbf{mfence}$	$\{\llbracket x \rrbracket_{t'}\}$		MFS_2
	$\{\llbracket x \rrbracket^{\mathsf{M}}_{t'}\}$	$t \neq t'$	MFS_3
	$\{\lvert x, v \rvert = n\}$		MFS_4
	$\{[x]^{\mathsf{A}}_{t} = S\}$		MFS_5

Statement	Stable Assert.	Const.	Ref.
$\mathbf{store}\ x\ v$	$\{[y]_{t'} = S\}$	$x \neq y$	WS_1
	$\{[y]^{\mathsf{P}} = S\}$	$x \neq y$	WS_2
	$\{[y]^{\mathsf{A}}_{t'} = S\}$	$x \neq y$	WS_3
	$\{a = k\}$		WS_4
	$\{\llbracket y \rrbracket_{t'}\}$	$x \neq y$	WS_5
	$\{\llbracket y \rrbracket^{\mathsf{F}}_{t'}\}$	$x \neq y$	WS_6
	$\{\lvert y, v' \rvert = n\}$	$x \neq y \vee v \neq v'$	WS_7
	$\{\llbracket y : v \rrbracket\}$	$x \neq y$	WS_8
	$\{\llbracket y \rrbracket^{\mathsf{M}}_{t'}\}$	$x \neq y$	WS_9
	$\{\llbracket y \rrbracket^{\mathsf{F}}_{t'}\}$	$x \neq y$	WS_{10}
$\mathbf{flush}_{\mathrm{opt}}\ x$	$\{[y]_{t'} = S\}$		OS_1
	$\{[y]^{\mathsf{P}} = S\}$		OS_2
	$\{\lvert y, v \rvert = n\}$		OS_3
$a := \mathbf{CAS}\ x\ e_1\ e_2$	$\{v \notin [y]_{t'}\}$	$x \neq y$	CS_1
	$\{\llbracket x : v \rrbracket\}$	$v \neq e_1$	CS_2
	$\{[y]^{\mathsf{A}}_{t} = S\}$	$x \neq y \vee \neg \llbracket x : e_1 \rrbracket$	CS_3
	$\{[y]^{\mathsf{P}} = S\}$	$x \neq y \vee \neg \llbracket x : e_1 \rrbracket$	CS_4
	$\{\llbracket y \rrbracket_{t'}\}$	$x \neq y \vee \neg \llbracket x : e_1 \rrbracket$	CS_5

Figure 4.8: Selection of stable assertions for atomic statements executed by thread t

Precondition	Statement	Postcondition	Const.	Ref.
$\{[x]_t = \{u\}\}$	$r := \mathbf{load}\, x$	$\{r = \vec{x}\}$		LP_4
$\{true\}$	$\mathbf{store}\ x\ v$	$\{\mathsf{LE}(x) = \lvert M \rvert - 1\}$		SP_9
$\{true\}$		$\{\vec{x} = v\}$		SP_{10}
$\{true\}$		$\{\mathsf{LE}_{\mathsf{coh}}(x, t, x) = \lvert M \rvert - 1\}$		SP_{11}
$\{true\}$		$\{M[\mathsf{LE}_{\mathsf{coh}}(x, t, x)] \equiv \langle x := v \rangle\}$		SP_{12}
$\{true\}$	$a := \mathbf{CAS}\ x\ e_1\ e_2$	$\{a \Rightarrow \mathsf{LE}_{\mathsf{coh}}(x, t, x) = \lvert M \rvert - 1\}$		CP_7
$\{true\}$		$\{a \Rightarrow \vec{x} = e_2\}$		CP_8
$\{true\}$		$\{a \Rightarrow [y]_t = \{\vec{y}\}\}$		CP_9
$\{true\}$		$\{a \Rightarrow M[\mathsf{LE}_{\mathsf{coh}}(x, t, y)] \equiv \langle y := \vec{y} \rangle\}$		CP_{10}
$\{\vec{x} = v\}$		$\{\vec{x} = v \vee \vec{x} = e_2\}$		CP_{11}
$\{true\}$	$\mathbf{Crash}$	$\{M[0](x) = \vec{x}\}$		C_1
$\{true\}$		$\{[x]^{\mathsf{P}}, [x]^{\mathsf{A}}_{t}, [x]_t = \{M[0](x)\}\}$		C_2
$\{[x]^{\mathsf{P}} = \{v\}\}$		$\{\vec{x} = v\}$		C_3

Figure 4.9: Selected proof rules for atomic statements executed by thread t. Note t may be equal to t' and x may be equal to y unless explicitly ruled out.

The proof rules shown in Fig. 4.7 and Fig. 4.8, along with, continue to hold in βPx86$_{\mathrm{view}}$. Fig. 4.9 extends Fig. 4.7 and contains additional rules of atomic statement for view-based expressions that have been developed in the context of PIEROGI$_{\mathrm{full}}$. Rule LP_4 states that if the thread view of t for x contains only one element, then after the execution of a **load** instruction to x, the value read is surely the value of the last write at x. This is ensured by the well-formedness condition (see below).

Rules SP_9 - SP_{12} refer to the **store** instruction. Rule SP_9 states that in the post-state the timestamp of the last message in memory with location x ($\mathsf{LE}(x)$), becomes the index of the last message in memory (|M|-1). By rule SP_{10} the last written value at x ($\vec{x}$) becomes equal to v. Since the coherence view of x ($\mathsf{coh}(x)$) becomes equal to $|M|-1$ the expressions $\mathsf{LE}(x)$ and $\mathsf{LE}_{\mathsf{coh}}(x,t,x)$ are equivalent in the post-state. Therefore, as stated in rule SP_{11}, in the post-state $\mathsf{LE}_{\mathsf{coh}}(x,t,x)$ becomes equal to $|M|-1$. Furthermore, in the post-state by rule SP_{12} the message that corresponds to $\mathsf{LE}_{\mathsf{coh}}(x,t,x)$ has location x and value v.

Rules CP_7 - CP_{11} refer to the **CAS** instruction. A returned value true (resp. false) indicates a **CAS** success (resp. failure) Rules CP_7 - CP_{10} describe the conditions that hold in case of a **CAS** success. When a **CAS** succeeds, it stores at x the value of e_2. Similar to the **store** instruction postconditions, in the post state $\mathsf{LE}(x)$ becomes equivalent to the $\mathsf{LE}_{\mathsf{coh}}(x,t,x)$ expression and equal to $|M|-1$ (rule CP_7). Moreover, the last written value on x becomes e_2 (rule CP_8). Most importantly, (rule CP_9), after a successful execution of **CAS**, the thread view of all the locations for t is updated to include only their last written value. By rule CP_{10} the message that corresponds to $\mathsf{LE}_{\mathsf{coh}}(x,t,y)$, for any location y in the memory, has as location y and as value the last written value on y ($\vec{y}$). Finally, rule CS_{11} states that in the post-state the last written value at x either remains the same, indicating a **CAS** failure, or it changes to e_2.

Rules C_1 - C_3 concern the **Crash** transition. Rule C_1 states that in the post-crash state, the initial message of the memory maps its location to its last stored value. This is trivial to show as after a crash, only a single value (namely, the one that persisted prior to the crash) remains observable for each location in the memory. By rule C_2 the *current views* for each location x in the post-crash state contain only the value to which they are mapped in the initial message. Finally, C_3 states that if the persistent view of any location x includes only one value v in the pre-crash state, the last stored value on x ($\vec{x}$) after a crash takes place, will definitely be v.

Fig. 4.10 extends Fig. 4.8 and contains a selection of assertions (middle column) that are proven stable against the corresponding atomic statements (left column) taking into account the constraints mentioned in the right. These proof rules are mostly used for establishing global correctness.

Statement	Stable Assert.	Const.	Ref.
$r := \textbf{load}\, x$	$\{\vec{y} = u\}$		LS_8
	$\{\mathsf{LE}_{\mathsf{coh}}(z, t', y)\}$	$t \neq t'$	LS_9
	$\{\mathsf{LE}(y)\}$		LS_{10}
$a := \textbf{CAS}\ x\ e_1\ e_2$	$\{\vec{x} = v\}$	$\vec{x} \neq e_1$	CS_6
	$\{\mathsf{LE}_{\mathsf{coh}}(z, t', y) = t\}$	$t \neq t'$	CS_7
	$\{\mathsf{LE}(y) = t\}$	$x \neq y \vee \vec{x} \neq e_1$	CS_8
sfence	$\{\vec{x} = v\}$		SFS_4
	$\{\mathsf{LE}(x) = v\}$		SFS_5
	$\{\vec{y} = v\}$	$x \neq y$	WS_{11}
	$\{\mathsf{LE}_{\mathsf{coh}}(z, t', y) = v\}$	$x \neq z \vee t \neq t'$	WS_{12}
	$\{\mathsf{LE}(y) = v\}$	$x \neq y$	WS_{13}
store $x\ v$	$\{\vec{y} = v\}$		OS_4
$\textbf{flush}_{\text{opt}}\ x$	$\{\mathsf{LE}_{\mathsf{coh}}(z, t', y) = v\}$		OS_5
	$\{\mathsf{LE}(y) = v\}$		OS_6

Figure 4.10: Selection of stable assertions for atomic statements executed by thread t. Note x may be equal to y and t may be equal to t' unless explicitly ruled out.

Well-formedness. The final major aspect of our framework is a well-formedness condition that describes the set of reachable states in the Px86$_{\text{view}}$ semantics. The well-formedness condition is common for both PIEROGI$_{\text{simp}}$ and PIEROGI$_{\text{full}}$. The condition is expressed as an invariant of the semantics: it holds initially and is stable under every possible transition of Px86$_{\text{view}}$. In fact, the rules in Figs. 4.7, 4.8, 4.9 and 4.10 are proved with respect to this well-formedness condition.

The majority of the well-formedness constraints are straightforward, e.g. describing the relationship between the views of different components. We provide a brief description of the constraints below.

The most important component of the well-formedness condition is a non-emptiness condition on views. Specifically, it is guaranteed that its current view expression of an address x ($\in$ LOC) contains at least the last write to x. Formally,

$$[x]_t \supseteq \{\vec{x}\} \wedge [x]^{\mathsf{P}} \supseteq \{\vec{x}\} \wedge [x]_t^{\mathsf{A}} \supseteq \{\vec{x}\}$$

For instance, a consequence of this condition is that, in combination with LP_1, we have:

$$\{[y]_t = \{v\}\}\ a := \textbf{load}\, x\ \{[y]_t = \{v\}\} \tag{4.7}$$

Furthermore, the well-formedness condition ensures that all the view components of the thread state (§4.1.2.1) are inside the limits of the memory, i.e. $0 \leq \mathit{view} < |\sigma.M|$.

Finally, the well-formedness condition provides two important properties regarding the coherence (coh) view of the thread state. Firstly,

$$\forall x \in \text{L}\textsc{oc}. t \in \text{T}\textsc{id}. \mathsf{MemLoc}(x, \sigma.\mathbb{T}(t).\mathsf{coh}(x), \sigma.M) = x$$

By this, we mean that the location of the memory message that the coherence view for a location x of a thread t points to is always equal to x.

Secondly,

$$\forall x \in \text{L}\textsc{oc}. t \in \text{T}\textsc{id}. i \in \mathsf{TS}_{\mathsf{t}}(\sigma, x) \Rightarrow \sigma.\mathbb{T}(t).\mathsf{coh}(x) \leq i$$

The above expresses the fact that the coherence view of x for thread t in state σ is the lower bound of the set of *timestamps* that are *observable* for t to read in x in σ.

4.3 Mechanization

Perhaps the greatest strength of our development is an integrated Isabelle/HOL mechanization provides a fully-fledged semi-automated verification tool for $\text{Px86}_{\text{view}}$ programs. This mechanization builds on the existing work on Owicki–Gries for RC11 by Dalvandi *et al.* [39] applying it to the $\text{Px86}_{\text{view}}$ semantics.

The mechanization of $\textsc{Pierogi}_{\text{simp}}$ started by encoding the operational semantics of Cho *et al.* [32] (§4.1), followed by the view-based assertions of $\textsc{Pierogi}_{\text{simp}}$ described in §4.2.1. Then, we proved the correctness of all of the proof rules for the atomic statements and view-based assertions of $\textsc{Pierogi}_{\text{simp}}$, including those described in §4.2.3. These rules can be challenging to prove since they require unfolding of the assertions and examination of the low-level operational semantics and their effect on the views of different system components.

Once proved, the rules provided are highly reusable, and are key to making verification feasible. In particular, when showing the validity of a proof outlines, Isabelle/HOL is able to generate the necessary proof obligations (after some minor interactions), then *automatically* able to find the set of high-level proof rules needed to discharge each proof obligation via the built-in sledgehammer tool [25]. This facility enables a high degree of experimentation and debugging of proof outlines, including the ability to reduce the complexity of assertions once a proof outline has been validated.

The base development of $\textsc{Pierogi}_{\text{simp}}$ (semantics, view-based assertions, and soundness of proof rules) comprise ∼7000 lines of Isabelle/HOL code and took approximately 2 months.

The mechanization of $\textsc{Pierogi}_{\text{full}}$ started by encoding the necessary modifications to the $\text{Px86}_{\text{view}}$ model [32] to reflect the revised version ($\beta\text{Px86}_{\text{view}}$) presented in §4.1. We then adapted the view-based assertions of $\textsc{Pierogi}_{\text{simp}}$ to the new setting and added the *last-entry* assertions of $\textsc{Pierogi}_{\text{full}}$(§4.2.1). Finally, we adapted a selection of the $\textsc{Pierogi}_{\text{simp}}$ proof rules (§4.2.3) to the new context and proved additional proof rules for $\textsc{Pierogi}_{\text{full}}$ (§4.2.3).

The base development of $\textsc{Pierogi}_{\text{full}}$ (semantics, view-based assertions, and soundness of proof rules) comprises ∼10,000 lines of Isabelle/HOL code and took approximately 2.5 months.

4.4 Related Work

The soundness of Pierogi is proven relative to the $\text{Px86}_{\text{view}}$ of Cho *et al.* [32]; there are however other equivalent models in the literature [2, 90, 122]. While the original persistent x86 semantics have explicit asynchronous persist instructions [122], the underlying model assumed in this work is the one due to Cho *et al.* [32], whose persist instructions are synchronous. Nevertheless, Khyzha and Lahav [90] formally proved that the two alternatives are equivalent when reasoning about states after crashes (e.g. using our "crash invariants").

Another program logic for persistent programs is POG [119], which (as with Pierogi) is a descendent of Owicki–Gries [115]. Pierogi goes beyond POG by handling examples that involve **flush**$_{\text{opt}}$ instructions, which cannot be directly verified using POG. Raad *et al.* [119] provide a transformation technique to replace certain patterns of **flush**$_{\text{opt}}$ and **sfence** with **flush**. Specifically, given a program Π that includes **flush**$_{\text{opt}}$ instructions, provided that Π meets certain conditions, this transformation mechanism rewrites Π into an equivalent program Π' that uses **flush** instructions instead of allowing one to use POG. However, there are three limitations to this strategy: 1) the rewriting is an external mechanism that requires stepping outside the POG logic; 2) the rewriting is potentially expensive and must be done for every program that includes **flush**$_{\text{opt}}$; and 3) the transformation technique is incomplete in that not all programs meet the stipulated conditions (e.g. Epoch Persistency 2), and thus cannot be verified using this technique. Pierogi has no such limitations. Moreover, POG has no corresponding mechanization, and developing a mechanization that also efficiently handles the program transformation for **flush**$_{\text{opt}}$ instructions would be non-trivial.

Vindum *et al.* [140] have recently developed a concurrent separation logic for weak persistency called Spirea. This logic is built upon the Perenial [28] and Iris [28] logic framework and has been mechanized in the Coq proof assistant. Spirea is not architecture-specific. Instead, it assumes the more generic explicit epoch persistency model, as proposed by Izraelevitz *et al.* [81]. The assumed consistency model incorporates release-acquire and non-atomic consistency modes, closely resembling the release-acquire synchronization and non-atomic segments of C11. As with Pierogi, the underlying semantics and logic assertions are expressed in terms of view-based expressions, i.e. thread and persistent memory views. To reason about the state after a system crash, the logic utilizes Perennial's *crash Hoare triples.* Denoted as $\{P\}e\{Q\}\{Q_c\}$, a crash Hoare triple requires the crash condition Q_c to hold at every step of execution of e. In the scenario where a program incorporates a recovery procedure, the logic relies on Perennial's *recovery Hoare triples* for reasoning about the state after recovery. Denoted as $\{P\}e\{Q\}\{Q_r\}$, a recovery Hoare triple states that assuming e terminates with a value v, $Q(v)$ holds if no crash-recovery event has occurred. Otherwise, $Q_r(v)$ must hold.

The Owicki–Gries method was first applied to non-SC memory consistency by Lahav *et al.* [97]. One way that their approach, which targets the release/acquire memory model, is different from ours is that they aim to use standard SC-like assertions; in order to retain soundness under a weak memory model, they had to strengthen the standard stability conditions on proof outlines. Dalvandi *et al.* [39, 43] took a different approach when designing their Owicki–Gries logic for the release/acquire fragment of C11: by employing a more expressive, view-based assertion language, they were able to stick with the standard stability requirement. In our work, we follow Dalvandi *et al.*'s approach. However, our assertions are fine-tuned to cope with the other types of view present in Px86$_{\text{view}}$, such as those corresponding to the persistent and the asynchronous views. It is interesting that some of the principles of view-based reasoning apply to different memory models, and future work could look at unifying reasoning across models.

Dalvandi *et al.* [39] have developed a deeper integration of their view-based logic using the Owicki–Gries encoding of Nipkow and Prensa Nieto [111] in Isabelle/HOL. Such an integration would be straightforward for Pierogi$_{\text{simp}}$ too, allowing verification to take place without translating programs into a transition system. This would be much more difficult for POG since Owicki–Gries rules themselves are different from the standard encoding in Isabelle/HOL, in addition to the transformation required for $\mathbf{flush}_{\mathrm{opt}}$ instructions discussed above.

The idea of extending Hoare triples with crash conditions first appeared in the work of Chen *et al.* [31]. However, that work supports neither concurrency nor explicit flushing instructions. Related ideas are found in the works of Ntzik *et al.* [112] and Chajed *et al.* [29]. However, in contrast to $\textsc{Pierogi}_{\mathrm{simp}}$, both of these works 1) assume sequentially consistent memory, as opposed to a weak memory model such as TSO; 2) assume strict persistency (where store and persist orders coincide); and 3) assume there is a synchronous **flush** operation, which is easier to reason about than the asynchronous $\mathbf{flush}_{\mathrm{opt}}$ operation.

Besides program logics, there have been other recent efforts to help programmers reason about persistent programs. For instance, Abdulla *et al.* [2] have proven that state-reachability for persistent x86 is decidable, thus opening the door to automatic verification of persistent programs, and Gorjiara *et al.* [64] have developed a model checker for finding bugs in persistent programs.

Chapter 5

Utilizing PIEROGI$_{\text{simp}}$

In this chapter, we demonstrate the use PIEROGI$_{\text{simp}}$ for verifying several idiomatic persistent x86 programs. We begin with recapping the PIEROGI$_{\text{simp}}$ and its reasoning principles (§5.1). We then verify a selection of programs using PIEROGI$_{\text{simp}}$ in §5.2 and discuss their Isabelle/HOL mechanization (§5.3). The Isabelle/HOL mechanization of the examples can be found at
https://doi.org/10.6084/m9.figshare.18469103.v2.

5.1 Reasoning principles of PIEROGI$_{\text{simp}}$

This section constitutes an informal discussion of the reasoning principles of PIEROGI$_{\text{simp}}$. We aim to provide insight into the process of developing proof outlines and establishing their validity as outlined in Def. 4.2.1. We do this by analyzing in high level a series of running examples.

In this section and the remainder of this chapter, we assume the registers of distinct threads have distinct names. The precondition P in Fig. 5.1 states that both threads may initially only read 0 for both x and y: $\forall t \in \{1, 2\}. [x]_t = [y]_t = \{0\}$

Sequential Reasoning about Consistency using Views. In Fig. 5.1 we present a PIEROGI$_{\text{simp}}$ proof sketch of MP. Recall that in order to account for possible write-read reorderings on Intel-x86 architectures, Px86$_{\text{view}}$ associates each thread t with a coherence view, describing the writes visible to t. To reason about such thread-observable views, recall that PIEROGI$_{\text{simp}}$ supports assertions of the form $[x]_t = S$, stating that t may read any value in the set S for location x. That is, the *thread view* of t for x (see §4.2.1) consists of the writes whose values are those in S. We will now discuss the Local Correctness proof of Fig. 5.1 (see Def. 4.2.1).

In the case of thread 1, we can weaken P (using the standard rule of consequence of Hoare logic – see Cons in §4.2.3) to obtain P_1. Upon executing **store** x 42 (1) we weaken the resulting assertion by dropping the $a = 0$ conjunct; and (2) we update the observable view of thread 1 on x to reflect the new value of x: $[x]_1 = \{42\}$; that is, after executing **store** x 42, the only value observable by thread 1 for x is 42. Similarly, after executing

$$P : \{a = b = 0 \land \forall t \in \{1,2\}.\, [x]_t = [y]_t = \{0\}\}$$

$$\begin{array}{l||l}
P_1 : \{7 \notin [y]_2 \ \land a = 0\} & Q_1 : \{[y]_2 \subseteq \{0,7\} \land (7 \in [y]_2 \Rightarrow \langle y, 7\rangle[x]_2 = \{42\})\} \\
\quad \textbf{store}\ x\ 42; \ //\ \mathsf{SP}_1, \mathsf{Cons} & \quad a := \textbf{load}\, y; \ //\ \mathsf{LP}_2 \\
P_2 : \left\{ [x]_1 = \{42\} \ \land 7 \notin [y]_2 \right\} & Q_2 : \left\{ a \in \{0,7\} \land (a = 7 \Rightarrow [x]_2 = \{42\}) \right\} \\
\quad \textbf{store}\ y\ 7; \ //\ \mathsf{SP}_1, \mathsf{Cons} & \quad b := \textbf{load}\, x; \ //\ \mathsf{LP}_1, \mathsf{Cons} \\
P_3 : \{\mathsf{true}\} & Q_3 : \left\{ a = 7 \Rightarrow \ b = 42 \right\}
\end{array}$$

$$Q : \{a = 7 \Rightarrow b = 42\}$$

Figure 5.1: A PIEROGI$_{\text{simp}}$ proof sketch of message passing (MP), where the //annotation at each step identifies the PIEROGI$_{\text{simp}}$ proof rule (in §4.2.3) applied, and the highlighted assertions capture the effects of the preceding instruction.

store y 7, we could assert $[y]_1 = \{7\}$; however, this is not necessary for establishing the final postcondition Q, and we thus simply weaken the postcondition to true (P_3).

Analogously, in the case of thread 2 we weaken P to obtain Q_1: $[y]_2 = \{0\}$ implies $[y]_2 \subseteq \{0, 7\}$ and $7 \in [y]_2 \Rightarrow \langle y, 7\rangle[x]_2 = \{42\}$. Note that $7 \in [y]_2 \Rightarrow \langle y, 7\rangle[x]_2 = \{42\}$ yields a vacuously true implication as $[y]_2 = \{0\}$ and thus $7 \notin [y]_2$. The $\langle y, 7\rangle[x]_2$ constitutes a *conditional view assertion* [39](see §4.2.1), capturing the essence of message passing by stating how reading a value on one location (y) affects the thread-observable view on a different location (x). More concretely, $\langle y, 7\rangle[x]_2 = \{42\}$ states that if thread 2 executes a load on y and reads value 7, it subsequently may only observe value 42 for x. This is indeed the essence of message passing in MP: once thread 2 reads 7 from y, it may only read 42 for x thereafter. As such, after executing the read instruction $a := \textbf{load}\, y$ (1) we apply the LP_1 rule (in Fig. 4.7) which simply replaces $[y]_2$ with the local register a in which the value of y is read; and (2) we replace the conditional assertion $\langle y, 7\rangle[x]_2 = \{42\}$ with the implication $a = 7 \Rightarrow [x]_2 = \{42\}$, stating that if the value read by thread 2 for y (in a) is 7, then its observable view for x is $\{42\}$. Similarly, upon executing $b := \textbf{load}\, x$ we simply apply LP_1 to replace $[x]_2$ with the local register b in which the value of x is read. Lastly, the final postcondition Q is given by the conjunction of the thread-local postconditions ($P_3 \land Q_3$).

Concurrent Reasoning and Stability. In our description of the PIEROGI$_{\text{simp}}$ proof sketch in Fig. 5.1 thus far we focused on *sequential* (per-thread) reasoning, ignoring how concurrent threads may affect the validity of assertions at each program point. Specifically, as in existing concurrent logics [97, 115, 119], we must ensure that the assertions at each program point are *stable* under concurrent operations. This corresponds to the $\mathsf{Stability}$ requirement of the valid proof outline (see Def. 4.2.1).

For instance, to ensure that P_1 remains stable under the concurrent operation $a := \textbf{load}\, y$, we require that executing $a := \textbf{load}\, y$ on states satisfying the conjunction of P_1 and the precondition of $a := \textbf{load}\, y$ (i.e. Q_1) not invalidate P_1, in that the resulting states continue to satisfy P_1; that is, $\{P_1 \land Q_1\} a := \textbf{load}\, y \{P_1\}$ holds. Similarly, we must ensure that P_1 is stable under $b := \textbf{load}\, x$, i.e. $\{P_1 \land Q_2\} b := \textbf{load}\, x \{P_1\}$ holds. Analogously, we must establish the stability of P_2, P_3, Q_1, Q_2 and Q_3 under concurrent operations. In §4.2.3 we present syntactic rules that simplify the task of checking stability obligations. It is then straightforward to show that the assertions in Fig. 5.1 are stable.

Reasoning about flush Persistency. To reason about the relaxed, buffered persistency of Px86$_{\text{view}}$, Cho *et al.* [32] introduce *persistency views*, determining the possible

$$\{[y]^{\mathsf{P}} = \{0\}\}$$
$$\quad \textbf{store } x\ 1; \ /\!/\ \mathsf{SP}_1$$
$$\left\{ [x]_1 = \{1\} \land [y]^{\mathsf{P}} = \{0\} \right\}$$
$$\quad \textbf{flush } x; \ /\!/\ \mathsf{FP}_1$$
$$\left\{ [x]_1 = \{1\} \land [x]^{\mathsf{P}} = \{1\} \land [y]^{\mathsf{P}} = \{0\} \right\}$$
$$\quad \textbf{store } y\ 1; \ /\!/\ \mathsf{SP}_1$$
$$\left\{ [x]_1 = \{1\} \land [x]^{\mathsf{P}} = \{1\} \land [y]_1 = \{1\} \right\}$$
$$\{\!\{ \lightning : [y]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{1\} \}\!\}$$

$$\{[y]^{\mathsf{P}} = \{0\}\}$$
$$\quad \textbf{store } x\ 1; \ /\!/\ \mathsf{SP}_1$$
$$\left\{ [x]_1 = \{1\} \land [y]^{\mathsf{P}} = \{0\} \right\}$$
$$\quad \textbf{flush}_{\text{opt}}\ x; \ /\!/\ \mathsf{OP}_1$$
$$\left\{ [x]_1{=}\{1\} \land [x]_1^{\mathsf{A}}{=}\{1\} \land [y]^{\mathsf{P}}{=}\{0\} \right\}$$
$$\quad \textbf{sfence}; \ /\!/\ \mathsf{SFP}_1$$
$$\left\{ [x]_1{=}\{1\} \land [x]^{\mathsf{P}}{=}\{1\} \land [y]^{\mathsf{P}}{=}\{0\} \right\}$$
$$\quad \textbf{store } y\ 1; \ /\!/\ \mathsf{SP}_1$$
$$\left\{ [x]_1{=}\{1\} \land [x]^{\mathsf{P}}{=}\{1\} \land [y]_1{=}\{1\} \right\}$$
$$\{\!\{ \lightning : [y]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{1\} \}\!\}$$

Figure 5.2: Proof sketches of Fig. 2.7b (left) and Fig. 2.7d (right)

persisted values for each location; i.e. the values of those writes that may have persisted to memory. Note that the persistency view determines the possible values observable upon recovery from a crash. By contrast, the (per-thread) thread views determine the observable values during normal (non-crashing) executions, and have no bearing on the post-crash values.

Analogously, PIEROGI$_{\text{simp}}$ supports assertions of the form $[x]^{\mathsf{P}} = S$, stating that the *persistent view for* x (see §4.2.1) includes writes whose values are given by S. To see this, consider the PIEROGI$_{\text{simp}}$ proof sketch of Fig. 2.7b in Fig. 5.2 (left). Initially, y holds 0 in persistent memory: $[y]^{\mathsf{P}} = \{0\}$. (Note that the precondition could additionally include $[x]_1 = [y]_1 = \{0\} \land [x]^{\mathsf{P}} = \{0\}$ to denote that initially the thread may only observe 0 for x and y and that x holds 0 in persistent memory; however, this is not needed for the proof and we thus forgo it.)

As before, after executing **store** x 1, the observable value for x is updated, as denoted by $[x]_1 = \{1\}$. Moreover, after executing **flush** x, the persisted value for x, as denoted by $[x]^{\mathsf{P}} = \{1\}$, by committing (persisting) the observable value for x ($[x]_1 = \{1\}$) to memory (see FP_1 in Fig. 4.7). Finally, after executing **store** y 1, the observable value for y is updated, as denoted by $[y]_1 = \{1\}$.

Crash Invariants. Recall that $\lightning\!: y{=}1 \Rightarrow x{=}1$ in Fig. 2.7b denotes a *crash invariant* in that it describes the persistent memory upon recovery from a crash at *any* program point. This is because we have no control over when a crash may occur. To capture such invariants, in PIEROGI$_{\text{simp}}$ we write *quadruples* of the form $\{P\}\ C\ \{Q\}\{\!\{\lightning : I\}\!\}$, where $\{P\}\ C\ \{Q\}$ denotes a Hoare triple and I denotes the crash invariant. If C is a sequential program, I must follow from *every* assertion (including P and Q) in the proof. For instance, in the proof outline of Fig. 5.2 (left) all four assertions imply the invariant $[y]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{1\}$. The above corresponds to the $\mathsf{Persistence}$ requirement of the valid proof outline (see Def. 4.2.1). We discuss the meaning of crash invariants for concurrent programs below.

Reasoning about flush$_{\text{opt}}$ Persistency. Recall that unlike **flush**, **flush**$_{\text{opt}}$ instructions (due to instruction reordering) may behave asynchronously and their effects may not take place immediately after execution. As such, unlike for **flush** x, after executing **flush**$_{\text{opt}}$ x we cannot simply copy the observable view on x to the persistent view on x.

```
$P : \{a = 0 \land \forall o \in \{x, y, z\}, t \in \{1, 2\}.\, [o]_t = [o]^{\mathsf{P}} = \{0\}\}$
$P_1 : \{[y]_2 = \{0\} \land [z]^{\mathsf{P}} = \{0\} \land a = 0\}$
    store $x$ 1; // SP$_1$
$P_2 : \{[y]_2 = \{0\} \land [z]^{\mathsf{P}} = \{0\} \land a = 0 \land [x]_1 = \{1\}\}$
    flush $x$; // FP$_1$, Cons
$P_3 : \{[x]^{\mathsf{P}} = \{1\}\}$
    store $y$ 1; // SP$_1$, Cons
$P_4 : \{[x]^{\mathsf{P}} = \{1\}\}$

||

$\{\mathsf{true}\}$
    $a :=$ load $y$;
$\{\mathsf{true}\}$
    if $(a = 1)$
        $\{a = 1\}$
        store $z$ 1;
$\{\mathsf{true}\}$

$Q : \{[x]^{\mathsf{P}} = \{1\}\}$
$I : \{\!\{ \lightning : [z]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{1\} \}\!\}$
```

Figure 5.3: A PIEROGI$_{\text{simp}}$ proof sketch of Fig. 2.7e

To capture the asynchronous nature of **flush**$_{\text{opt}}$, Cho *et al.* [32] introduce yet another set of views, namely the *thread-local asynchronous view*: the asynchronous view of thread t on x describes the values (writes) that will be persisted at a later time (asynchronously) by t upon executing a barrier instruction. That is, 1) when thread t executes **flush**$_{\text{opt}}$ x, its asynchronous view of x is advanced to at least its observable view of x; and 2) when t executes a barrier (**sfence**, **mfence** or RMW), then its persistent view for each location is advanced to at least its corresponding asynchronous view. We model this in PIEROGI$_{\text{simp}}$ by 1) setting $[x]^{\mathsf{A}}_t$ to be a subset of $[x]_t$ when **flush**$_{\text{opt}}$ x is executed (see OP in Fig. 4.7); and 2) setting $[x]^{\mathsf{P}}$ to be a subset of $[x]^{\mathsf{A}}_t$ (for each location x) when a barrier is executed (see SFP in Fig. 4.7).

This is illustrated in the proof sketch of Fig. 2.7d in Fig. 5.2 (right). In particular, unlike the proof sketch of Fig. 2.7b in Fig. 5.2 (left), after executing **flush**$_{\text{opt}}$ x we cannot simply copy the thread-observable view to the persistent view. Rather, we copy the thread-observable view $[x]_1$ to its asynchronous view and assert $[x]^{\mathsf{A}}_1 = \{1\}$; and upon executing the subsequent **sfence**, we copy the thread-asynchronous view to the persistent view and assert $[x]^{\mathsf{P}} = \{1\}$.

Putting It All Together We next present a PIEROGI$_{\text{simp}}$ proof sketch of Fig. 2.7e in Fig. 5.3.

The program in Fig. 5.3, assuming that the left thread has id 1, is given as follows (see §4.1.1). The formalization of the right thread is omitted but is similar.

$$\Pi \triangleq \left\{ \begin{array}{l} (1, \iota) \mapsto \textbf{store}\ x\ 1\ \textbf{goto}\ 2, (1, 2) \mapsto \textbf{flush}\ x\ \textbf{goto}\ 3, \\ (1, 3) \mapsto \textbf{store}\ y\ 1\ \textbf{goto}\ \zeta, \ldots \end{array} \right\}$$

The annotation of the proof in Fig. 5.3 is given by *ann*, with the mappings of thread 1 as shown below; the mappings of thread 2 are similar.

$$ann \triangleq \{(1, \iota) \mapsto P_1, (1, 2) \mapsto P_2, (1, 3) \mapsto P_3, (1, \zeta) \mapsto P_4, \ldots \quad \}$$

Additionally, we have $in \triangleq a = 0 \land \forall o \in \{x, y, z\}, t \in \{1, 2\}.\, [o]_t = [o]^{\mathsf{P}} = \{0\}$, $fin \triangleq [x]^{\mathsf{P}} = \{1\}$ and $I \triangleq [z]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{1\}$ (see the definition of *proof outline* in §4.2.2)

We now demonstrate that proof outline Fig. 5.3 is valid according to Def. 4.2.1. Both Initialisation and Finalisation clearly hold. Moreover, Persistence holds for thread 1. Note that the crash invariant I follows from the assertions at each program point of thread 1 (i.e. $P_1 \vee P_2 \vee P_3 \vee P_4 \Rightarrow I$). That is, the crash invariant must follow from the assertions at *all* program points of *some* thread (e.g. thread 1 in Fig. 5.3). In the case of sequential programs (e.g. in Fig. 5.2), this amounts to all program points (of the only executing thread). Intuitively, we must ensure that the crash invariant holds at every program point regardless of how the underlying state changes. As the assertions are stable under concurrent operations, it is thus sufficient to ensure that there exists some thread whose assertions at each program point imply the crash invariant.

For Local correctness of thread 1, we must prove (5.1)–(5.3) below; Local correctness of thread 2 is similar.

$$\{P_1\} \quad \textbf{store}\ x\ 1 \quad \{P_2\} \tag{5.1}$$

$$\{P_2\} \quad \textbf{flush}\ x \quad \{P_3\} \tag{5.2}$$

$$\{P_3\} \quad \textbf{store}\ y\ 1 \quad \{P_4\} \tag{5.3}$$

The proof of the left thread is analogous to that in Fig. 5.2 (left); the proof of the right thread is straightforward and applies standard reasoning principles. The final postcondition Q is obtained by weakening the conjunction of per-thread postconditions.

For Stability of P (the precondition of **store** x 1 in thread 1) against thread 2 we must prove:

$$\{P_1\} \quad a := \textbf{load}\, y \quad \{P_1\} \tag{5.4}$$

$$\{P_1 \wedge a = 1\} \quad \textbf{store}\ z\ 1 \quad \{P_1\} \tag{5.5}$$

Stability of other assertions (i.e., P_2–P_4) is similar. We prove (5.1)–(5.5) in §4.2.3.

5.2 Examples ()

In this section, we present a selection of programs that we have verified in Isabelle/HOL. These examples highlight specific aspects of Px86, in particular, the interaction between $\textbf{flush}_{\text{opt}}$ and **sfence**, as well as aspects of our view-based assertion language that simplifies verification.

5.2.1 Optimised Message Passing

We start by considering a variant of Fig. 2.7e (example in Fig. 5.4), which contains two optimizations. First, we notice that the flushing of the write to x in thread 1 can be moved to thread 2 since the write to z is guarded by whether or not thread 2 reads the flag y. Second, it is possible to replace the **flush** with a more optimized $\textbf{flush}_{\text{opt}}$ followed by an **sfence**. We confirm the correctness of these optimizations via the proof outline in Fig. 5.4. The optimized message passing in Fig. 5.4 ensures the same persistent invariant as Fig. 2.7e. However, the way in which this is established differs. In particular, in Fig. 2.7e, the persistent invariant holds due to thread 1, whereas in Fig. 5.4 it holds due to thread 2.

With respect to the persistent invariant, the most important sequence of steps takes place in thread 2 if it reads 1 for y. Note that by the conditional view assertion in

$$\{\forall o \in \{x, y, z\}, t \in \{1, 2\}.\, [o]_t = [o]^{\mathsf{P}} = [o]^{\mathsf{A}}_t = \{0\}\}$$

$$\begin{array}{l@{\quad}\Bigg\|@{\quad}l}
\{[y]_2 = \{0\}\} & \{\ (1 \in [y]_2 \Rightarrow \langle y, 1\rangle[x]_2 = \{1\}) \wedge [y]_2 \subseteq \{0, 1\} \wedge [z]^{\mathsf{P}} = \{0\}\ \} \\
\textbf{store}\ x\ 1; & a := \textbf{load}\, y; \\
\left\{\begin{array}{l} [y]_2 = \{0\} \wedge \\ [x]_1 = \{1\} \end{array}\right\} & \{(a = 1 \Rightarrow [x]_2 = \{1\}) \wedge [z]^{\mathsf{P}} = \{0\}\} \\
\textbf{store}\ y\ 1; & \textbf{if}\ (a \neq 0) \\
\{\mathsf{true}\} & \qquad \{[x]_2 = \{1\} \wedge [z]^{\mathsf{P}} = \{0\}\} \\
 & \qquad \textbf{flush}_{\text{opt}}\ x; \\
 & \qquad \{[x]^{\mathsf{A}}_2 = \{1\} \wedge [z]^{\mathsf{P}} = \{0\}\} \\
 & \qquad \textbf{sfence}; \\
 & \qquad \{[x]^{\mathsf{P}} = \{1\}\} \\
 & \qquad \textbf{store}\ z\ 1; \\
 & \{[z]^{\mathsf{P}} = \{0\} \vee [x]^{\mathsf{P}} = \{1\}\}
\end{array}$$

$$\{[z]^{\mathsf{P}} = \{0\} \vee [x]^{\mathsf{P}} = \{1\}\}$$

$$\{\!\{ \lightning : [z]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{1\} \}\!\}$$

Figure 5.4: Proof outline for optimised message passing

$$\{\forall o \in \{x, y\}, t \in \{1, 2\}.\, [o]_t = \{0\}\}$$

$$\begin{array}{l@{\quad}\Bigg\|@{\quad}l}
\left\{\begin{array}{l} (\hat{a}, \hat{b} = 0 \wedge [y]_1 \subseteq \{0, 1\} \\ \left(\begin{array}{l} \hat{a}, \hat{b} = 0, 1 \wedge [y]_2 = \{1\} \wedge \\ \llbracket y \rrbracket_2) \end{array}\right) \end{array}\right\} &
\left\{\begin{array}{l} (\hat{b}, \hat{a} = 0 \wedge [x]_2 \subseteq \{0, 1\} \\ \left(\begin{array}{l} \hat{b}, \hat{a} = 0, 1 \wedge [x]_1 = \{1\} \wedge \\ \llbracket x \rrbracket_1) \end{array}\right) \end{array}\right\} \\
\langle \textbf{store}\ x\ 1, \hat{a} := \hat{b} + 1 \rangle; & \langle \textbf{store}\ y\ 1, \hat{a} := \hat{b} + 1 \rangle; \\
\left\{\begin{array}{l} \left(\begin{array}{l} \hat{a} = 1 \wedge \hat{b} \in \{0, 2\} \wedge \\ [y]_1 \subseteq \{0, 1\} \end{array}\right) \vee \\ \left(\begin{array}{l} \hat{a}, \hat{b} = 2, 1 \wedge \llbracket y \rrbracket_2 \wedge \\ [y]_2 = \{1\} \wedge \llbracket y \rrbracket^{\mathsf{M}}_1 \end{array}\right) \end{array}\right\} &
\left\{\begin{array}{l} \left(\begin{array}{l} \hat{b} = 1 \wedge \hat{a} \in \{0, 2\} \wedge \\ [x]_2 \subseteq \{0, 1\} \end{array}\right) \vee \\ \left(\begin{array}{l} \hat{a}, \hat{b} = 2, 1 \wedge \llbracket x \rrbracket_1 \wedge \\ [x]_1 = \{1\} \wedge \llbracket x \rrbracket^{\mathsf{M}}_2 \end{array}\right) \end{array}\right\} \\
\textbf{mfence}; & \textbf{mfence}; \\
\left\{\begin{array}{l} \left(\begin{array}{l} \hat{a} = 1 \wedge \hat{b} \in \{0, 2\} \wedge \\ [y]_1 \subseteq \{0, 1\}) \end{array}\right) \vee \\ (\hat{a}, \hat{b} = 2, 1 \wedge [y]_1 = \{1\}) \end{array}\right\} &
\left\{\begin{array}{l} \left(\begin{array}{l} \hat{b} = 1 \wedge \hat{a} \in \{0, 2\} \wedge \\ [x]_2 \subseteq \{0, 1\}) \end{array}\right) \vee \\ (\hat{b}, \hat{a} = 2, 1 \wedge [x]_2 = \{1\}) \end{array}\right\} \\
r_1 := \textbf{load}\, y; & r_2 := \textbf{load}\, x; \\
\left\{\begin{array}{l} \left(\begin{array}{l} \hat{a} = 1 \wedge \hat{b} \in \{0, 2\} \wedge \\ (r_1 \in \{0, 1\}) \end{array}\right) \vee \\ (\hat{a}, \hat{b} = 2, 1 \wedge r1 = 1) \end{array}\right\} &
\left\{\begin{array}{l} \left(\begin{array}{l} \hat{b} = 1 \wedge \hat{a} \in \{0, 2\} \wedge \\ (r_2 \in \{0, 1\}) \end{array}\right) \vee \\ (\hat{b}, \hat{a} = 2, 1 \wedge r2 = 1) \end{array}\right\}
\end{array}$$

$$\{\ (r_1 = 1 \vee r_2 = 1)\ \}$$

Figure 5.5: Proof outline for store buffering

the precondition of $a := \textbf{load}\, y$, thread 2 is guaranteed to read 2 for x after reading 1 for y. Thus, if the test of **if** statement succeeds, then thread 2 must see 1 for x. This view is translated into an asynchronous view after the $\textbf{flush}_{\text{opt}}$ is executed, and then to the persistent view after executing **sfence**. Until this occurs, we can guarantee that $[z]^{\mathsf{P}} = \{0\}$, which trivially guarantees the persistent invariant.

5.2.2 Store, Flush and Optimised Flush Buffering

Initially, we illustrate a typical example of store buffering (SB). Subsequently, we provide two examples that resemble store buffering but involve persist instructions.

Store Buffering. Fig. 5.5 illustrates an example of store buffering. In this scenario, we are not concerned with the content of persistent memory during program execution. Since no explicit persist instructions are issued, we can infer that $[x]^{\mathsf{P}} \subseteq 0, 1$ and $[y]^{\mathsf{P}} \subseteq 0, 1$. Therefore, we skip the persistence part of the valid proof outline (Def. 4.2.1) and apply the standard Owicki–Gries method.

At the end of the program execution depicted in Fig. 5.5, we should observe that either $r_1 = 1$ or $r_2 = 1$. The read values of addresses x and y depend on the order in which the store instructions on x and y are executed. This is because stores are ordered with respect to subsequent **mfence** instructions of the same thread (see Fig. 2.8). According to Cho *et al.*'s semantics, after a thread t executes an **mfence** instruction, t's $\mathsf{v_{rNew}}$ view is updated to a timestamp greater than or equal to the thread's maxcoh view.

To facilitate reasoning about the stores' order, we use auxiliary variables $\hat{a}$ and $\hat{b}$. These auxiliary variables record the order in which the writes to x and y occur; $\hat{a} = 1$ iff the write to x occurs before the write to y, and $\hat{a} = 2$ iff the write to x occurs after the write to y. The reasoning for thread 1 and thread 2 are symmetric. There are two disjuncts for each precondition to consider: one consists of assertions that hold when **store** x 1 is executed before **store** y 1, and the other consists of assertions that hold when **store** y 1 is executed before **store** x 1.

Here, it is important to note that 1) the maxcoh view represents the timestamp of the last message added by t to the memory; 2) the $\mathsf{v_{rNew}}$ view contributes in determining the thread view of x for t, indicating the visible values to be read by t in the case of subsequent external reads (see the LOAD-EXTERNAL rule in Fig. 4.3).

If thread 1 executes **store** x 1 first, the maxcoh view of thread 1 becomes 1. The subsequent **mfence** instruction updates the $\mathsf{v_{rNew}}$ view of thread 1 to 1. At this point, the thread view of y for thread 1 contains either only the value 0 (if thread 2 has not executed **store** y 1 by then) or both the values 0 and 1 (in the opposite case). The subsequent read of y returns either 0 or 1.

We will now analyze the case where thread 1 executes **store** x 1 second. In this case. the maxcoh view of thread 1 becomes 2. After the execution of **store** x 1 three facts can be stated 1) the thread view of thread 2 for y contains only 1 ($[y]_2 = \{1\}$); 2) 1 is the last write on y ($\lceil\!\lceil y \rceil\!\rceil_2$); and 3) the timestamp of the last write of y which is 1 is less than the maxcoh view of thread 1, which is 2. After executing the **mfence** instruction, it is determined that the view of thread 1 for variable y will contain only the value 1 ($[y]_1 = 1$).This is expressed by rule MFP_1. In particular, the **mfence** instruction causes the $\mathsf{v_{rNew}}$ view of thread 1 to be updated to the value of the maxcoh view at that point, which is 2. Since the timestamp of the last write to variable x does not exceed the maxcoh view of thread 1, it is determined that thread 2's view for address x will only contain the last write to y, which is 1. This is due to the fact that any other writes to x are overwritten by the last write to x from the $\mathsf{v_{rNew}}$ view perspective.

Flush Buffering. Our next example is a variation of store buffering (SB) and is used to highlight how writes by different threads on different locations interact with flushes. Here, thread 1 writes to x and flushes y, while thread 2 writes to y then flushes x.[1] This example exhibits similarities to store buffering because the values that are potentially flushed to persistent memory by **flush** x and **flush** y depend on the order of execution

[1] Note that the **flush** operations here are analogous to the **load** instructions in SB.

$$\{\forall o \in \{w, x, y, z\}, t \in \{1, 2\}.\, [o]_t = [o]^{\mathsf{P}} = \{0\}\}$$

$$\begin{array}{l||l}
\left\{\begin{array}{l}(\hat{a}, \hat{b} = 0, 0 \wedge [z]^{\mathsf{P}} = \{0\}) \vee \\ \left(\begin{array}{l}\hat{a}, \hat{b} = 0, 1 \wedge \llbracket y \rrbracket_2 \wedge \\ [y]_2 = \{1\} \wedge [w]^{\mathsf{P}} = \{0\}\end{array}\right)\end{array}\right\} & \left\{\begin{array}{l}(\hat{a}, \hat{b} = 0, 0 \wedge [w]^{\mathsf{P}} = \{0\}) \vee \\ \left(\begin{array}{l}\hat{a}, \hat{b} = 1, 0 \wedge \llbracket x \rrbracket_1 \wedge \\ [x]_1 = \{1\} \wedge [z]^{\mathsf{P}} = \{0\}\end{array}\right)\end{array}\right\} \\
\langle \textbf{store}\ x\ 1, \hat{a} := \hat{b} + 1 \rangle; & \langle \textbf{store}\ y\ 1, \hat{b} := \hat{a} + 1 \rangle; \\
\left\{\begin{array}{l}\left(\begin{array}{l}\hat{a} = 1 \wedge \hat{b} \in \{0, 2\} \wedge \\ ([z]^{\mathsf{P}} = \{0\} \vee [x]^{\mathsf{P}} = \{1\}\end{array}\right) \vee \\ \left(\begin{array}{l}\hat{a}, \hat{b} = 2, 1 \wedge \llbracket y \rrbracket_2 \wedge \\ [y]_2 = \{1\} \wedge \llbracket y \rrbracket_1^{\mathsf{F}} \wedge [w]^{\mathsf{P}} = \{0\}\end{array}\right)\end{array}\right\} & \left\{\begin{array}{l}\left(\begin{array}{l}\hat{b} = 1 \wedge \hat{a} \in \{0, 2\} \wedge \\ ([w]^{\mathsf{P}} = \{0\} \vee [y]^{\mathsf{P}} = \{1\})\end{array}\right) \vee \\ \left(\begin{array}{l}\hat{a}, \hat{b} = 1, 2 \wedge \llbracket x \rrbracket_1 \wedge \\ [x]_1 = \{1\} \wedge \llbracket x \rrbracket_2^{\mathsf{F}} \wedge [z]^{\mathsf{P}} = \{0\}\end{array}\right)\end{array}\right\} \\
\textbf{flush}\ y; & \textbf{flush}\ x; \\
\left\{\begin{array}{l}\left(\begin{array}{l}\hat{a} = 1 \wedge \hat{b} \in \{0, 2\} \wedge \\ ([z]^{\mathsf{P}} = \{0\} \vee [x]^{\mathsf{P}} = \{1\}))\end{array}\right) \vee \\ (\hat{a}, \hat{b} = 2, 1 \wedge [y]^{\mathsf{P}} = \{1\})\end{array}\right\} & \left\{\begin{array}{l}\left(\begin{array}{l}\hat{b} = 1 \wedge \hat{a} \in \{0, 2\} \wedge \\ ([w]^{\mathsf{P}} = \{0\} \vee [y]^{\mathsf{P}} = \{1\})\end{array}\right) \vee \\ (\hat{a}, \hat{b} = 1, 2 \wedge [x]^{\mathsf{P}} = \{1\})\end{array}\right\} \\
\textbf{store}\ w\ 1; & \textbf{store}\ z\ 1; \\
\left\{\begin{array}{l}\left(\begin{array}{l}\hat{a} = 1 \wedge \hat{b} \in \{0, 2\} \wedge \\ ([z]^{\mathsf{P}} = \{0\} \vee [x]^{\mathsf{P}} = \{1\})\end{array}\right) \vee \\ (\hat{a}, \hat{b} = 2, 1 \wedge [y]^{\mathsf{P}} = \{1\})\end{array}\right\} & \left\{\begin{array}{l}\left(\begin{array}{l}\hat{b} = 1 \wedge \hat{a} \in \{0, 2\} \wedge \\ ([w]^{\mathsf{P}} = \{0\} \vee [y]^{\mathsf{P}} = \{1\})\end{array}\right) \vee \\ (\hat{a}, \hat{b} = 1, 2 \wedge [x]^{\mathsf{P}} = \{1\})\end{array}\right\}
\end{array}$$

$$\{\ (\hat{a}, \hat{b} = 1, 2 \wedge [x]^{\mathsf{P}} = \{1\}) \vee (\hat{a}, \hat{b} = 2, 1 \wedge [y]^{\mathsf{P}} = \{1\})\ \}$$

$$\{\!\{ \lightning : [w]^{\mathsf{P}} = \{1\} \wedge [z]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{1\} \vee [y]^{\mathsf{P}} = \{1\} \}\!\}$$

Figure 5.6: Proof outline for flush buffering

of the preceding store instructions.The writes to w and z are used to witness whether the flushes in both threads have occurred. The persistent invariant states that, if both w and z hold 1 in persistent memory, then either x or y has the new value (i.e. 1) in persistent memory. If both threads perform their **flush** operations, then at least one must flush value 1 since a **flush** cannot be reordered with a **store** (see Fig. 2.8).

Although simple to state, the proof is non-trivial since it requires careful analysis of the order in which the stores to x and y occur. In the semantics of Cho *et al.* [32], the **flush** corresponding to the *second* **store** instruction executed synchronises with writes to *all* locations. Thus, for example, if thread 1's store to x is executed after thread 2's store to y, then the subsequent **flush** in thread 1 is guaranteed to flush the new write to y.

The above intuition, as with the store buffering example, requires reasoning about the order in which operations occur. To facilitate this, we use auxiliary variables $\hat{a}$ and $\hat{b}$ to record the order in which the writes to x and y occur; $\hat{a} = 1$ iff the write to x occurs before the write to y, and $\hat{a} = 2$ iff the write to x occurs after the write to y. W.l.o.g., let us now consider the precondition of **flush** y (the reasoning for **flush** x is symmetric). In all post conditions there are two disjuncts to consider.

- The first disjunct describes the case in which thread 1 executes its store before thread 2. In this case the maxcoh of thread 1 is 1 and the maxcoh view of thread 2 is 2. After the execution of the subsequent **flush** y instruction by thread 1 the $\mathsf{v_{pCommit}}$ view of y is equal to 1. Since the $\mathsf{v_{pCommit}}$ view of y dictates the $[y]^{\mathsf{P}}$ view, there is a danger that the thread 1 can terminate having flushed 0 for y. However, from this state, thread 2 is guaranteed to flush 1 for x before setting z to 1, satisfying the persistent invariant, as described by the second disjunct of each assertion in thread 2.

- The second disjunct describes the case in which thread 1 executes its store after thread 2. In this case, the maxcoh of thread 1 is 2 and the maxcoh view of thread 2 is 1. Since the last write on y has a timestamp less than the maxcoh view after the execution of the **store** x 1 instruction, it guaranteed that when the **flush** y takes place the persistent view of y ($[y]^{\mathsf{P}}$) will only contain the value of the last write of y. This is because any other relevant timestamp will be overwritten by the last write of y. In particular after executing **store** x 1 thread 1 is guaranteed to flush 1 for y, and this fact is captured by the conjunct $\llbracket y \rrbracket_2 \wedge [y]_2 = \{1\} \wedge \llbracket y \rrbracket^{\mathsf{F}}_1$, which ensures that 1) thread 2 sees the last write to y; 2) the only value visible for y to thread 2 is 1; and 3) a flush performed by thread 1 is guaranteed to flush the last write to y. Note that by 1) and 2), we are guaranteed that the last write to y has value 1. We use these three facts to deduce that $[y]^{\mathsf{P}} = \{1\}$ in the second disjunct of the postcondition of **flush** y using rule FP_3.

Optimised Flush Buffering. Our next example demonstrates how writes by different threads on different locations interact with optimized flushes. In contrast to the **flush** instruction, $\mathbf{flush}_{\mathrm{opt}}$ is not ordered with respect to **store** instructions on different addresses. The example of Fig 5.7, is a variation of the flush buffering example that uses the optimised flush and sfence instructions instead of the flush instruction. As before, the writes to w and z are used to witness whether the optimized flushes followed by persist barriers in both threads have occurred. Intuitively, the order in which thread 1 executes **store** x 1 and thread 2 executes **store** y 1 does not impose any constraints on the order in which the effects of the subsequently issued optimized flushes take place. This is because the effect of the store instructions can take place in a later point from the execution of the optimized flush instructions if they do not concern the same address (see Fig. 2.8).

Let's consider the proof outline of thread 1. In the initial state all views contains only the value zero, thus the assertion $[w]^{\mathsf{P}} = \{0\} \wedge [y]_1 \subseteq \{0,1\}$ holds. After executing **store** x 1 by rules WS_1 and WS_2 (see Fig. 4.8) both $[w]^{\mathsf{P}}$ and $[y]_1$ remain unchanged. By rule OP (see Fig. 4.7) after the execution of the $\mathbf{flush}_{\mathrm{opt}}$ y instruction we can obtain that the asynchronous view of y for thread 1 becomes equal or a subset of its thread view ($[y]^{\mathsf{A}}_1 \subseteq \{0,1\}$). Furthermore, by rule OS_2, $[w]^{\mathsf{P}}$ remains the same. The **sfence** instruction causes $[y]^{\mathsf{P}}$ to become a subset or equal to $[y]^{\mathsf{A}}_1$ (rule SFP). Finally, by WS_2 after executing the **store** w 1 instruction $[y]^{\mathsf{P}}$ remains unchanged. The proof outline of thread 2 is symmetrical. It is trivial to show that the *crash invariant* holds.

$$
\begin{array}{c}
\{\forall o \in \{w,x,y,z\}, t \in \{1,2\}.\ [o]_t = [o]^{\mathsf{P}} = [o]^{\mathsf{A}}_t = \{0\}\} \\
\begin{array}{l||l}
\{[w]^{\mathsf{P}} = \{0\} \wedge [y]_1 \subseteq \{0,1\}\} & \{[z]^{\mathsf{P}} = \{0\} \wedge [x]_2 \subseteq \{0,1\}\} \\
\mathbf{store}\ x\ 1; & \mathbf{store}\ y\ 1; \\
\{[w]^{\mathsf{P}} = \{0\} \wedge [y]_1 \subseteq \{0,1\}\} & \{[z]^{\mathsf{P}} = \{0\} \wedge [x]_2 \subseteq \{0,1\}\} \\
\mathbf{flush}_{\mathrm{opt}}\ y; & \mathbf{flush}_{\mathrm{opt}}\ x; \\
\{[w]^{\mathsf{P}} = \{0\} \wedge [y]^{\mathsf{A}}_1 \subseteq \{0,1\}\} & \{[z]^{\mathsf{P}} = \{0\} \wedge [x]^{\mathsf{A}}_2 \subseteq \{0,1\}\} \\
\mathbf{sfence}; & \mathbf{sfence}; \\
\{[y]^{\mathsf{P}} \subseteq \{0,1\}\} & \{[x]^{\mathsf{P}} \subseteq \{0,1\}\} \\
\mathbf{store}\ w\ 1; & \mathbf{store}\ z\ 1; \\
\{[y]^{\mathsf{P}} \subseteq \{0,1\}\} & \{[x]^{\mathsf{P}} \subseteq \{0,1\}\}
\end{array} \\
\{\!\{ \lightning : [w]^{\mathsf{P}} = \{1\} \wedge [z]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} \subseteq \{0,1\} \vee [y]^{\mathsf{P}} \subseteq \{0,1\} \}\!\}
\end{array}
$$

Figure 5.7: Proof outline for a flush buffering variation with optimised flush and sfence instructions

5.2.3 Epoch Persistency

In the following two examples, we observe a pattern of epoch persistency. Through these examples, we illustrate how the sequence of optimized flushes and loads can influence the persistency behavior. The only distinction between the two programs showcased in Figs.5.8 and 5.9 lies in the order of operations performed by thread 2. In the case of Fig, 5.8, thread 2 executes first a load operation on x ($a := \textbf{load}\, x$), followed by an optimized flush to x ($\textbf{flush}_{\text{opt}}\ x$). Conversely, in the case of Figure 5.9, thread 2 executes an optimized flush to x ($\textbf{flush}_{\text{opt}}\ x$) first and then proceeds with a load operation on x ($a := \textbf{load}\, x$). In both programs, the write to y is used for witnessing if the **if** statement of thread 2 succeeds. Additionally, the write operation on variable z acts as a mechanism to ascertain the successful execution of the preceding statements within the program before any crash event occurs. As we analyze in the remaining of the section, due to the fact that the **load** introduction is ordered with respect to subsequent $\textbf{flush}_{\text{opt}}$ instructions, program 5.8 provides a stronger guarantee regarding the values that can be observed to persistent memory during its execution.

Epoch Persistency 1. The crash invariant of Fig. 5.8 states that if z and y hold the value 1 in persistent memory then x has the value 2 in persistent memory. In order for thread 2 to read value 2 for x, the **store** of 2 at x must be performed before the **store** of 1 at x. In this case $[x]_2 = \{1, 2\}$. Otherwise, if the last store of x is 1 then $[x]_2 = \{1\}$ and thus the subsequent **if** statement will fail. Unlike the previous example, establishing the persistent invariant for thread 2 requires reasoning about the view of thread 2 for address x (i.e. $[x]_2$) after the execution of the instruction $a := \textbf{load}\, x$. Notice here that $a := \textbf{load}\, x$ is ordered with respect to the later $\textbf{flush}_{\text{opt}}\ x$ instruction. Consequently, any impact of the execution of the **load** on thread view of x for thread 2 ($[x]_2$), will also affect the asynchronous view of x for thread 2 ($[x]_2^{\mathsf{A}}$). Taking into account the ordering of the writes at the address x, we can conclude that if thread 2 reads the value 2, it reads the value of the last write at x. By rule LP_3, if a thread t's view of an address x contains only the last write at this address, and the last value written at this address appears only once at the memory, then if a thread t read this value at x, its view of x (i.e. $[x]_t$) is guaranteed to contain only the last written value at x. Consequently, after reading value 2, thread 2's view of x contains only the value 2 (i.e. $[x]_2 = \{2\}$). The execution of $\textbf{flush}_{\text{opt}}\ x$ ensures $[x]_2^{\mathsf{A}}$ (by rule OP). The OP rule holds because of the way the FLUSHOPT transition alters the $\mathsf{v}_{\mathsf{pAsync}}$ view. Specifically, if $a := \textbf{load}\, x$ returns 2 then, the $\mathsf{v}_{\mathsf{pReady}}$ view of thread 2 is updated to the timestamp of the last write at x (2). The following $\textbf{flush}_{\text{opt}}$ instruction sets the $\mathsf{v}_{\mathsf{pAsync}}$ view of x for thread 2 to a timestamp greater or equal to $\mathsf{v}_{\mathsf{pReady}}$. Subsequently, the asynchronous view of x ($[x]_2^{\mathsf{A}}$), after the execution of the $\textbf{flush}_{\text{opt}}$ instructions contains only the values of messages that correspond to timestamps that are not overwritten by any other timestamp from the $\mathsf{v}_{\mathsf{pAsync}}$'s view perspective. In this case, the only such timestamp is 2.

To conclude, in the case that the **if** statement succeeds (i.e. $\mathsf{v}_{\mathsf{pReady}}$ is 2 in the post state of the $a := \textbf{load}\, x$ instruction), after the execution of the **sfence** it is guaranteed (by rule SFP) that the value of x in persistent memory is 2 (i.e. $[x]^{\mathsf{P}} = \{2\}$). In the case that the **if** statement fails, $[y]^{\mathsf{P}} = \{0\}$ must hold, thus the crash invariant holds trivially.

Epoch Persistency 2. We now consider a second epoch persistency example in Fig. 5.9. The crash invariant of Fig. 5.9 states that if z and y hold the value 1 in persistent memory then x has either the value 1 or 2 in the persistent memory. The first **store** of thread 2,

$$\{(\forall t \in \{1,2\}, o \in \{x,y,z\}.[o]_t = [o]^{\mathsf{P}} = \{0\}) \wedge a = 0\}$$

$$\begin{array}{l||l}
\begin{array}{l}
\left\{ \begin{array}{l} |x,2| = 0 \wedge \\ \left(\begin{array}{l} ([x]_2 = 0 \wedge [x]_1 = 0) \vee \\ ([x]_2 = 1 \wedge [x]_1 = \{0,1\}) \end{array} \right) \end{array} \right\} \\
\textbf{store } x \ 2; \\
\left\{ \begin{array}{l} |x,2| = 1 \wedge \\ \left(\left(\begin{array}{l} [x]_1 = \{2\} \wedge \\ [x]_2 \subseteq \{1,2\} \end{array} \right) \vee \\ [x]_2 \subseteq \{0,1,2\} \end{array} \right) \end{array} \right\}
\end{array}
&
\begin{array}{l}
\{[y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\} \wedge (|x,2| \in \{0,1\})\} \\
\textbf{store } x \ 1; \\
\left\{ \begin{array}{l} \left([x]_2 = 1 \vee \left(\begin{array}{l} [x]_2 = \{1,2\} \wedge |x,2| = 1 \\ \wedge [x]_1 = 2 \end{array} \right) \right) \wedge \\ [y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\} \end{array} \right\} \\
a := \textbf{load } x; \\
\{(a = 2 \Rightarrow [x]_2 = \{2\}) \wedge [y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\}\} \\
\textbf{flush}_{\text{opt}} \ x; \\
\{(a = 2 \Rightarrow [x]_2^{\mathsf{A}} = \{2\}) \wedge [y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\}\} \\
\textbf{if } (a = 2) \\
\quad \{[x]_2^{\mathsf{A}} = \{2\} \wedge [y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\}\} \\
\quad \textbf{store } y \ 1; \\
\{([x]_2^{\mathsf{A}} = \{2\} \vee [y]^{\mathsf{P}} = \{0\}) \wedge [z]^{\mathsf{P}} = \{0\}\} \\
\textbf{sfence}; \\
\{[x]^{\mathsf{P}} = \{2\} \vee [y]^{\mathsf{P}} = \{0\}\} \\
\textbf{store } z \ 1; \\
\{[x]^{\mathsf{P}} = \{2\} \vee [y]^{\mathsf{P}} = \{0\} \vee [z]^{\mathsf{P}} = \{0\}\}
\end{array}
\end{array}$$

$$\{\ [x]^{\mathsf{P}} = \{2\} \vee [y]^{\mathsf{P}} = \{0\} \vee [z]^{\mathsf{P}} = \{0\}\ \}$$

$$\{\!\{ \lightning : [y]^{\mathsf{P}} = \{1\} \wedge [z]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{2\} \}\!\}$$

Figure 5.8: Proof outline for first example of epoch persistency

and the **flush**$_{\text{opt}}$ that follows, are performed at the same address and, therefore cannot be reordered. Reading the value 2 at x implies that the **store** of value 2 at x is performed before the **store** of value 1 at x. Otherwise, thread 2 would only have the option to read the value 1 at x. Given that the load instruction, which determines the order of the stores on variable x, occurs later in the execution, after **store** x 1 we can only infer that $[x]_2 \subseteq \{1,2\}$. After the **flush**$_{\text{opt}}$ is executed, $[x]_2$ is translated into an asynchronous view (by rule OP). The subsequent **load** instruction does not affect the asynchronous view of x for thread 2 since the LOAD-EXTERNAL and LOAD-INTERNAL transition does not modify the $\mathsf{v}_{\mathsf{pAsync}}$ view of x, which determines the asynchronous view set. The execution of **sfence** translates the asynchronous view $[x]_2^{\mathsf{A}} \subseteq \{1,2\}$ to the corresponding persistent view $[x]^{\mathsf{P}} \subseteq \{1,2\}$ (by rule SFP). In case that the **if** statement fails, we are certain that $[y]^{\mathsf{P}} = \{0\}$, thus the crash invariant holds trivially.

5.2.4 CAS-Based Locking

Let us consider now the program of Fig. 5.10. In this example, we use **CAS** as a lock in order to control accesses on x. The crash invariant here, states that if z holds the value 1 in persistent memory then x and y should also obtain the value 1 in persistent memory. In this example, the invariant is established by thread 2.

In order for the invariant to hold, we must ensure that the **flush** instructions of thread 1, are executed before thread 2 executes **store** z 1. The **CAS** instructions at the beginning of the two threads program ensure that the threads are not executing in parallel. In particular, in order for thread 1's **CAS** to succeed the value of the last write on x should be 0. If the value of the last write on x is 2, it means that thread 2's **CAS** is executed and the execution point hasn't reached yet the thread 2's instruction **store** lx 0. In this case the thread 1 **CAS** fails, and its execution stalls. More concretely, by rule CP_4, after the execution of **CAS** in thread 1, we can obtain that either $a_1 = 1 \wedge [\![lx : 1]\!]$ (indicating

```
$\{(\forall t \in \{1,2\}, o \in \{x,y,z\}.[o]_t = [o]^{\mathsf{P}} = \{0\}) \wedge a = 0\}$
    $\{\ \{[x]_2 = \{0\} \vee [x]_2 = \{1\}\}\ \}$
    store $x$ 2;
    $\{\mathsf{true}\}$
||
    $\{[y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\}\}$
    store $x$ 1;
    $\{\ [x]_2 \subseteq \{1,2\} \wedge [y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\}\ \}$
    flush$_{\text{opt}}$ $x$;
    $\{\ [x]_2^{\mathsf{A}} \subseteq \{1,2\} \wedge [y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\}\ \}$
    $a :=$ load $x$;
    $\{[x]_2^{\mathsf{A}} \subseteq \{1,2\} \wedge [y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\}\}$
    if $(a = 2)$
        $\{[x]_2^{\mathsf{A}} \subseteq \{1,2\} \wedge [y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\}\}$
        store $y$ 1;
    $\{([x]_2^{\mathsf{A}} \subseteq \{1,2\} \vee [y]^{\mathsf{P}} = \{0\}) \wedge [z]^{\mathsf{P}} = \{0\}\}$
    sfence;
    $\{([x]^{\mathsf{P}} \subseteq \{1,2\} \vee [y]^{\mathsf{P}} = \{0\}) \wedge [z]^{\mathsf{P}} = \{0\}\}$
    store $z$ 1;
    $\{[x]^{\mathsf{P}} \subseteq \{1,2\} \vee [y]^{\mathsf{P}} = \{0\} \vee [z]^{\mathsf{P}} = \{0\}\}$
$\{\ [x]^{\mathsf{P}} \subseteq \{1,2\} \vee [y]^{\mathsf{P}} = \{0\} \vee [z]^{\mathsf{P}} = \{0\}\ \}$
$\{\!\{ \lightning : [y]^{\mathsf{P}} = \{1\} \wedge [z]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} \subseteq \{1,2\}\}\!\}$
```

Figure 5.9: Proof outline for second example of epoch persistency

that the **CAS** succeed) or $a_1 = 0$. Respectively, in order for thread 2's **CAS** to succeed the value of the last write on x should be 0. If the value of the last write on x is 1, it means that that thread 1's **CAS** is executed and the execution point hasn't reached yet the thread 1's instruction **store** lx 0. In this case the execution of thread 2 stalls. There are two ways for $\llbracket lx : 0 \rrbracket$ to hold for thread 2 before the execution of **CAS**. Either the **CAS** reads the initial value of lx or it reads the value that lx obtains after thread 1 executes the instruction **store** lx 0. In the second case, which is the desirable one, we are sure that before thread 2 executes **CAS**, $[x]^{\mathsf{P}} = [y]^{\mathsf{P}} = [y]_2 = 1$. Those cases are described in the precondition of **CAS** in thread 2.

The first disjunct of the precondition concerns the case in which thread's 2 write on lx is not executed yet, but the value of the last write on lx is 0. From this, it can be inferred that lx obtains its initial value. In this case, we are sure that $1 \notin [x]_2$. The consecutive **CAS** might succeed, although it is certain that thread 2 can not read 1 at x. As a result, the second **if** statement of thread 2 fails, thus $[z]^{\mathsf{P}} \neq \{1\}$, consequently the invariant holds.

The second disjunct of the precondition concerns the case in which thread's 2 **store** lx 0 has been executed. Because the store of 1 at x by thread 1 is ordered before the store of 0 at lx, it is certain that at this point of execution $\llbracket x \rrbracket_1 \wedge [x]_1 = \{1\}$ holds. By rule CP_4 if the consecutive **CAS** succeeds, thread 1's view of x is transferred to thread 2. As a result the **if** statement that follows succeeds and **flush** z persists the value 1 at z. Because either $[x]^{\mathsf{P}} = \{1\} \wedge [y]^{\mathsf{P}} = \{1\}$ or $[z]^{\mathsf{P}} = \{1\}$ for every state of thread 2's program, the invariant holds.

The third disjunct of the precondition concerns the case in which thread's 1 **CAS** has succeeded and thus $\llbracket x : 1 \rrbracket$. In this case thread, 2's **CAS** can not succeed and the invariant holds trivially.

```
$\{\forall v \in \{lx, x, y, z\}, t \in \{1,2,3\}.\ [v]_t = \{0\} \wedge [v]^{\mathsf{P}} = \{0\}\}$

$\{(\llbracket lx : 0 \rrbracket \wedge a_1 = 0) \vee (\llbracket lx : 2 \rrbracket \wedge a_1 = 0)\}$
$a_1$ := CAS $lx$ 0 1;
$\{(a_1 = 1 \wedge \llbracket lx : 1 \rrbracket) \vee a_1 = 0\}$
if $(a_1 = 1)$
$\{\llbracket lx : 1 \rrbracket\}$
    store $x$ 1;
    $\{\llbracket lx : 1 \rrbracket \wedge [x]_1 = \{1\} \wedge \llbracket x \rrbracket_1\}$
    store $y$ 1;
    $\{\llbracket lx : 1 \rrbracket \wedge [x]_1 = \{1\} \wedge \llbracket x \rrbracket_1 \wedge [y]_1 = \{1\}\}$
    flush $x$;
    $\{\llbracket lx : 1 \rrbracket \wedge [x]^{\mathsf{P}} = \{1\} \wedge \llbracket x \rrbracket_1 \wedge [y]_1 = \{1\} \wedge [x]_1 = \{1\}\}$
    flush $y$;
    $\{\llbracket lx : 1 \rrbracket \wedge [x]^{\mathsf{P}} = \{1\} \wedge \llbracket x \rrbracket_1 \wedge [y]^{\mathsf{P}} = \{1\} \wedge [x]_1 = \{1\}\}$
    store $lx$ 0;
$\{([x]^{\mathsf{P}} = \{1\} \wedge [y]^{\mathsf{P}} = \{1\}) \vee a_1 = 0\}$

||

$\left\{\begin{array}{l} (\llbracket lx : 0 \rrbracket \wedge 1 \notin [x]_2 \wedge [z]^{\mathsf{P}} = \{0\} \wedge a_2 = a_3 = 0) \vee \\ (\llbracket lx : 0 \rrbracket \wedge [x]_1 = [x]^{\mathsf{P}} = [y]^{\mathsf{P}} = \{1\} \wedge \llbracket x \rrbracket_1 \wedge [z]^{\mathsf{P}} = \{0\} \wedge a_2 = a_3 = 0) \vee \\ (\llbracket lx : 1 \rrbracket \wedge [z]^{\mathsf{P}} = \{0\} \wedge a_2 = a_3 = 0) \end{array}\right\}$
$a_2$ := CAS $lx$ 0 2;
$\left\{\begin{array}{l} (\llbracket lx : 2 \rrbracket \wedge a_2 = 1 \wedge 1 \notin [x]_2 \wedge [z]^{\mathsf{P}} = \{0\}) \vee \\ (\llbracket lx : 2 \rrbracket \wedge [x]^{\mathsf{P}} = [y]^{\mathsf{P}} = \{1\} \wedge [z]^{\mathsf{P}} = \{0\} \wedge a_2 = 1 \wedge [x]_2 = \{1\}) \vee \\ (a_2 = 0 \wedge [z]^{\mathsf{P}} = \{0\}) \end{array}\right\}$
if $(a_2 = 1)$
    $\left\{\begin{array}{l} (\llbracket lx : 2 \rrbracket \wedge 1 \notin [x]_2 \wedge [z]^{\mathsf{P}} = \{0\}) \vee \\ (\llbracket lx : 2 \rrbracket \wedge [x]^{\mathsf{P}} = \{1\} \wedge [y]^{\mathsf{P}} = \{1\} \wedge \\ [z]^{\mathsf{P}} = \{0\} \wedge [x]_2 = \{1\}) \end{array}\right\}$
    $a_3$ := load $y$;
    $\left\{\begin{array}{l} (a_3 = 1 \Rightarrow \llbracket lx : 2 \rrbracket \wedge [x]^{\mathsf{P}} = \{1\} \wedge [y]^{\mathsf{P}} = \{1\} \wedge [x]_2 = \{1\}) \wedge \\ \llbracket lx : 2 \rrbracket \wedge [z]^{\mathsf{P}} = \{0\} \end{array}\right\}$
    if $(a_3 = 1)$
        $\{[x]^{\mathsf{P}} = \{1\} \wedge [y]^{\mathsf{P}} = \{1\} \wedge \llbracket lx : 2 \rrbracket\}$
        store $z$ 1;
        $\{[x]^{\mathsf{P}} = \{1\} \wedge [y]^{\mathsf{P}} = \{1\} \wedge \llbracket lx : 2 \rrbracket \wedge [z]_1 = \{1\}\}$
        flush $z$;
    $\left\{\begin{array}{l} ([x]^{\mathsf{P}} = \{1\} \wedge [y]^{\mathsf{P}} = \{1\} \wedge \llbracket lx : 2 \rrbracket \\ \wedge [z]^{\mathsf{P}} = \{1\}) \\ \vee ([z]^{\mathsf{P}} = \{0\} \wedge \llbracket lx : 2 \rrbracket) \end{array}\right\}$
    store $lx$ 0;
$\{([x]^{\mathsf{P}} = \{1\} \wedge [y]^{\mathsf{P}} = \{1\} \wedge [z]^{\mathsf{P}} = \{1\}) \vee ([z]^{\mathsf{P}} = \{0\})\}$

$\{\ [z]^{\mathsf{P}} = \{0\} \vee ([x]^{\mathsf{P}} = \{1\} \wedge [y]^{\mathsf{P}} = \{1\})\ \}$
$\{\!\{ \lightning : [z]^{\mathsf{P}} = \{1\} \Rightarrow ([x]^{\mathsf{P}} = \{1\} \wedge [y]^{\mathsf{P}} = \{1\}) \}\!\}$
```

Figure 5.10: CAS-based locking

5.3 Mechanization

The mechanization of the examples presented in this chapter is built on top of the PIEROGI$_{\text{simp}}$ framework. With the base development in place, each example comprises 200–400 lines of code (including the encoding of the program, the annotations, and the proofs of validity). The validity proofs of these examples took approximately 1 month of full-time work.

Chapter 6

dTML under Px86

In this chapter, we focus on the *correctness* and *verification* of of transactional memory algorithms with respect to a realistic memory model, namely Px86. In particular, we adapt the *durable* Transactional Mutex Lock (dTML$_{\mathsf{SC}}$) algorithm (see Chapter 3), which itself is a durable extension of the Transactional Mutex Lock [38] with logging mechanisms that support recoverability, and show that it is *durably opaque.* This work brings together two research areas, the world of *hardware* weak memory models [32,119,120,123] and the world of correctness conditions for concurrent objects and transactional memory in the context of persistent memory (see §2.4.2.1). Unlike these prior works, as the name implies, dTML$_{\mathrm{Px86}}$ assumes Intel's x86 persistency and consistency model (Px86) [78], which extends the x86 TSO model [131] with a persistency semantics [32,90,120,122,123].

In our initial effort to validate dTML$_{\mathrm{Px86}}$, we attempted to use PIEROGI$_{\mathrm{simp}}$, as detailed in Chapter 4 ([24]). However, using this logic directly in our current work is not possible for two reasons.

(1) Like prior works on verifying Px86 programs [32, 119], PIEROGI$_{\mathrm{simp}}$ [24] has only focussed on reasoning about the behaviour *up to the first crash* of the program. To fully establish the correctness of dTML$_{\mathrm{Px86}}$, it is critical to also reason about the program after restarting the system.

(2) The PIEROGI$_{\mathrm{simp}}$ assertions are inadequate for reasoning about certain phenomena that occur in dTML$_{\mathrm{Px86}}$. In particular, we must often reason about memory patterns by considering the sequence in which writes occur, which previously defined assertions do not cover.

To this end, we shifted to using the PIEROGI$_{\mathrm{full}}$ logic (Chapter 4).

To exploit the efficiency possibilities of Px86, we make use of *optimized flush* instructions (**flush**$_{\mathrm{opt}}$) throughout dTML$_{\mathrm{Px86}}$. These instructions improve performance by asynchronously tagging writes that are to be persisted when an **sfence** or another persist barrier instruction is later encountered (see example Fig. 2.7c). Optimized flush instructions are however difficult to reason about in the context of Px86$_{\mathrm{view}}$. In fact, earlier Owicki-Gries logics [119] only provided partial support for **flush**$_{\mathrm{opt}}$ instructions and required programs with **flush**$_{\mathrm{opt}}$ instructions to be transformed into a program with synchronized **flush** instructions only. This transformation technique was known to be incomplete [119]. Full support for $\textbf{flush}_{\text{opt}}$ could only be provided after the development of the view-based semantics for Px86 [32] and their corresponding Owicki-Gries logic [24]. Our proofs for $\text{dTML}_{\text{Px86}}$ represent the first large-scale proofs of correctness for a realistic program that uses $\textbf{flush}_{\text{opt}}$.

Our correctness proof of $\text{dTML}_{\text{Px86}}$ uses *forward simulation* to establish a refinement between the abstract operational specification dTMS2 (see §3.4.3). This, to our knowledge, is the first operational proof of refinement for the Px86. Other works have used refinement to verify durable linearizability directly under the declarative Px86 model [53,118]. Unlike our work, these prior works are not accompanied by any mechanization. [41] have considered operational refinement proofs of transactional memory algorithms under the RC11 memory model. These proofs have a different set of complexities (e.g., relaxed and release-acquire accesses), but do not require consideration of durability or recovery as we do in $\text{dTML}_{\text{Px86}}$.

To summarise, the main contributions of this work are as follows

(1) We develop a durable transactional memory $\text{dTML}_{\text{Px86}}$ that guarantees durable opacity under $\text{Px86}_{\text{view}}$. Our algorithm makes use of optimized flush instructions for improved efficiency, increasing the verification challenge.

(2) To take advantage of our operational reasoning technique, we apply a simulation-based proof to show the correctness of $\text{dTML}_{\text{Px86}}$ by refinement. The proof proceeds via a long-established technique of establishing a forward simulation between the implementation and an abstract specification [23,44,52]. In the context of transactional memory, we prove that $\text{dTML}_{\text{Px86}}$ is a refinement of an operational model dTMS2, whose traces are guaranteed to be durably opaque (see Theorem 3.4.1).

(3) We mechanise our entire development ranging from the semantics, logic (including the soundness of the atomic Hoare triples), and all proofs pertaining to $\text{dTML}_{\text{Px86}}$, including proofs of the invariant and simulation.

This chapter is organized as follows. In §6.1, we provide some background and further motivation for our work. In §6.2, we present our $\text{dTML}_{\text{Px86}}$ implementation, and in §6.3, the $\text{dTML}_{\text{Px86}}$ model. The invariants of $\text{dTML}_{\text{Px86}}$ are demonstrated in §6.4. In §6.5 we present the durable opacity proof of $\text{dTML}_{\text{Px86}}$. Finally in §6.6 we discuss the mechanization effort and in §6.7 the related work.

The Isabelle/HOL formalization of the $\text{dTML}_{\text{Px86}}$ correctness proof can be found at https://doi.org/10.6084/m9.figshare.25037312.v2.

6.1 Motivation

In this work, we are aiming to verify an algorithm that exceeds the complexity of the litmus test examples verified in Chapter 4 and follows the more complicated compared to sequential consistency (Chapter 3), but realistic Px86 model. The proposed transactional memory implementation must not only ensure correct thread synchronization and persistence under the Px86 memory model but also provide transactional correctness. Below we summarize the challenges introduced by the aforementioned aspects.

Px86 Semantics. The Px86 model introduces several challenges as analysed in §2.2.2 related to the weak ordering of the Px86$_{\text{view}}$ instructions (see Fig. 2.8). As a consequence of the allowed reorderings, the issuing order of a program's instructions deviates from the order in which the effects of the instructions impact the volatile memory. Additionally, these two aforementioned orders differ from the order in which the effects of the instructions impact the persistent memory. Examples of the weak behaviors exhibited by the Px86 model can be found in Fig. 2.7.

The Px86 model describes the semantics of programs running in systems that display Intel-x86 hardware, including an NVM technology such as Optane DC. The logic that we use to verify dTML$_{\text{Px86}}$, PIEROGI$_{\text{full}}$, covers programs that run on App Direct mode platforms that support `ADR`. For brevity we make the assumption that each cache line only holds one location, eliminating the need to reason about other locations on the same cache line.

Transactional Memory Correctness. There is extended literature regarding transactional memory correctness. In Chapter 2, we analyze some of the most prominent correctness conditions. A popular condition here is *opacity* [67], which ensures that there exists a total order across *all* transactions so that the committed transactions are strictly serialized and the aborted transactions are consistent with the serialization order. While the above provides semantics for transaction consistency; under persistent memory, we also require a further guarantee of *failure atomicity.* To this end, we follow the notion of *durable opacity* (Def. 3.1.1), where all transactions committed before a crash are persistent (after the crash), and in addition, the effects of any partially executed transactions are not visible after the crash.

Implementation Challenges Under Px86. There are two main challenges when developing durable TM algorithms under weak memory models such as Px86. **1)** The first challenge concerns the thread *synchronization.* This difficulty is introduced by the fact that in the relaxed memory context, a read of a shared location may not return the location's last written value. For **2)** The second challenge concerns *durability.* Without placing correctly explicit flush instructions in the algorithm and the careful design of a recovery mechanism, there is no guarantee on which values are visible in memory after a system crash.

To tackle the initial challenge under TSO we must use instructions with strong ordering guarantees (e.g. **CAS**) at key points within our TM implementation algorithm, preventing threads from reading stale values. To tackle the second challenge, we strategically position **flush**$_{\text{opt}}$ along with the **sfence** instructions in a way that does not compromise the algorithm's efficiency. We also design a recovery process that enables the state to be reset to a consistent state after a crash.

6.2 The dTML$_{\text{Px86}}$ Algorithm

Pseudocode for dTML$_{\text{Px86}}$ is given in Fig. 6.1. In order to handle the relaxed behaviors introduced by Px86, we introduce several extensions to the original TML implementation [38]. Specifically, the lines highlighted blue ensure correct thread synchronization

```
TMBegin
$Bp$ : do $\mathsf{loc}_t$ := load glb;
$B1$ : until $even(\mathsf{loc}_t)$;
       return $ok$;

TMRead($x$)
  $Rp$ : $r_t$ := load $x$;
  $R1$ : if $even(\mathsf{loc}_t) \wedge \neg\mathsf{hasRead}_t$ then
  $R2$ :    $\mathsf{hasRead}_t$ := CAS glb $\mathsf{loc}_t$ $\mathsf{loc}_t$;
  $R3$ :    if $\mathsf{hasRead}_t$ then
              return $r_t$;
            else return $abort$;
  $R4$ : $c_t$ := load glb;
  $R5$ : if $c_t = \mathsf{loc}_t$ then
            return $r_t$;
         else return $abort$;

TMCommit
  $Cp$ : if $odd(\mathsf{loc}_t)$ then
  $C1$ :    sfence;
  $C2$ :    $log$.empty();
  $C3$ :    ⟨store glb ($\mathsf{loc}_t + 1$),
            writer := $None$⟩
         return $commit$;
```

```
TMWrite($x, v$)
  $Wp$ : if $even(\mathsf{loc}_t)$ then
  $W1$ :    $\mathsf{hasWritten}_t$ := CAS glb $\mathsf{loc}_t$ ($\mathsf{loc}_t + 1$);
  $W2$ :    if $\mathsf{hasWritten}_t$ then
  $W3$ :       ⟨$\mathsf{loc}_t := \mathsf{loc}_t + 1$, writer := $t$⟩
            else return $aborted$
  $W4$ : if $\neg log$.contains($x$) then
  $W5$ :    $c_t$ := load $x$;
  $W6$ :    $log$.update($x, c_t$);
  $W7$ : store $x$ $v$;
  $W8$ : $\mathbf{flush}_{\mathrm{opt}}$ $x$;
         return $ok$;

TMRecover
  $Rec1$  : while $\neg log$.isEmpty()
  $Rec2$  :    $c_{syst} := log$.getKey();
  $Rec3$  :    store $c_{syst}$ $log$.getVal($c_{syst}$);
  $Rec4$  :    $\mathbf{flush}_{\mathrm{opt}}$ $c_{syst}$;
  $Rec5$  :    sfence;
  $Rec6$  :    $log$.update($c_{syst}, \bot$);
  $Rec7$  : $c_{syst}$ := load glb;
  $Rec8$  : if $even(c_{syst})$ then
  $Rec9$  :    ⟨store glb $c_{syst} + 2$,
                recGlb := $c_{syst} + 2$⟩
  $Rec10$ : else ⟨store glb ($c_{syst} + 1$),
                recGlb := $c_{syst} + 1$⟩
```

Figure 6.1: Durable Transactional Mutex Lock

under relaxed memory, while the lines highlighted green are required to ensure correctness under persistency. The variables highlighted grey are auxiliary. All the local variables apart from the auxiliary ones are modeled as registers. To distinguish them from global variables, we index the registers with the *id* of the transaction that they belong to. As before, we assume that thread identifiers coincide with the transaction identifiers. As in the previous chapters, we will use the term *internal* read for a read that a transaction performs to a location that previously wrote itself, and *external* read for a read that a transaction performs to a location that has been written by another transaction. We also use the term *live* transaction, for a transaction that has not been completed (has not returned from a `TMCommit` operation).

We assume that all locations, the registers for every transaction, the global variable glb, and the auxiliary variable recGlb, are initialized to zero. Furthermore, the auxiliary variable writer is initialized to *None*.

The basic TML algorithm is presented in §3.3.2. We explain the extensions we developed for adapting TML to $\mathrm{Px86}_{\mathrm{view}}$ in stages, starting with the extensions that concern correct synchronization.

6.2.1 Correct Synchronization Under Px86

Under Px86, in the presence of multiple writes to a location, a read may return a value that is not the last written value. A writing transaction must successfully perform a **CAS** at line $W1$, which guarantees that it reads the last written value of glb. However, in the standard TML algorithm [38, 44] (which assumes sequentially consistent memory), this synchronization is not present in read-only transactions. Thus, a read-only transaction may complete with a stale value of glb!

To address this, we follow a similar approach to [41] in the RC11 memory model, and introduce a **CAS** in the `TMRead` operation ($R2$), mimicking a fetch-and-add-zero to ensure that the last value of glb is read. Note that this **CAS** only needs to performed if the the corresponding transaction has not previously performed a read or a write, thus at line $R1$, we bypass $R2$ when loc_t is odd or $\mathsf{hasRead}_t$ holds. If this **CAS** succeeds the executing transaction can immediately return the read value, and if this **CAS** fails, the transaction can immediately abort ($R3$).

Although our solution to weak memory synchronization is similar to the RC11 memory model [41], there are subtle differences in the way our solution guarantees correctness of reads. Unlike RC11 memory model which requires a "release" synchronization on the read corresponding to Rp, in TSO, is sufficient to perform a standard read. To explain the issue, consider the program in Fig. 6.1 without lines $R1$-$R3$. An execution of this program can reach a state with the following memory sequence:

$$\langle M_0, \langle \mathsf{glb} := 1\rangle, \langle x := 1\rangle, \langle \mathsf{glb} := 2\rangle, \langle \mathsf{glb} := 3\rangle, \langle x := 2\rangle, \langle \mathsf{glb} := 4\rangle\rangle$$

after executing two transactions writing 1 then 2 to the location x. Now suppose transaction t starts, then reads $\mathsf{glb} := 2$ (i.e., $\mathsf{loc}_t = 2$), allowing it to complete `TMBegin`, then performs a `TMRead`(x) operation. The Px86 semantics allow it to read from the stale write $\langle x := 1\rangle$, then commit the transaction, violating opacity.

The program in Fig. 6.1 with lines $R1$-$R3$ can also reach the state above. Again suppose transaction t starts, then reads $\mathsf{glb} := 2$ (i.e., $\mathsf{loc}_t = 2$), allowing it to complete `TMBegin`, then executes a `TMRead`(x) operation reading the stale write $\langle x := 1\rangle$. However, now the transaction proceeds to line $R2$ and since loc_t is not the last written value of glb, the **CAS** fails, thus the transaction aborts.

6.2.2 Read-only transactions in Px86.

Like [41], we observe other behaviours of dTML$_{\text{Px86}}$ that would not be present under SC memory without affecting durable opacity. In particular, a read-only transaction t is not immediately invalidated when glb is updated by another writing transaction, provided t continues to read from transactional locations that are consistent with the an older (stale) value of glb. This read-only transaction would be able to successfully commit if it *never* reads a value for x that is more recent than its copy of glb. In case a read-only transaction reads a value of a location, x at Rp, that is more recent than it's local copy of glb, the load of glb at $R4$ would also read a more recent copy of glb and the transaction would subsequently abort. This particular synchronization property is much simpler to guarantee that in the RC11 model [41], which requires careful management of release-acquire annotations.

6.2.3 Ensuring durability.

Durability of TML under a stronger sequentially consistent memory model has been studied in previous work [23] (Chapter 3). The main idea there was to introduce a durably linearizable [81] persistent undo log that records the previous values of locations that have been overwritten by incomplete writing transactions. The log is reset to empty when the writing transaction commits. If a crash occurs when a incomplete writing transaction t is in flight, the subsequent recovery operation sets the state to the last consistent state by undoing the writes of t using the undo log.

In dTML$_{\text{Px86}}$, we use the same recovery mechanism but optimize the algorithm using **flush**$_{\text{opt}}$ instructions. This is in contrast to dTML$_{\mathsf{SC}}$ (Fig. 3.4), which performs a **flush** after every write to memory within the `TMWrite` operation. The precise semantics of **flush**$_{\text{opt}}$ was only defined in recent works [32, 122] and is known to be a challenge to verify. For instance, the POG logic [119] requires a transformation to rewrite **flush**$_{\text{opt}}$ - **sfence** sequences with **flush** instructions and the transformation is known to be incomplete. Our work uses PIEROGI$_{\text{full}}$ that deals directly with **flush**$_{\text{opt}}$ instructions via special assertions that capture the values that persist when an **sfence** instruction is later executed.

6.3 dTML$_{\text{Px86}}$ Model

We build a transition system model for dTML$_{\text{Px86}}$. In this model, we must clarify possible histories of the algorithm, which in turn requires us to clarify the invocation and response events. We assume that the algorithm is executed by a *most-general client* [52] that calls the operations of dTML$_{\text{Px86}}$.

dTML$_{Px86}$ Executions and Histories. As with the dTML$_{\mathsf{SC}}$ model, we assume a *program counter* pc_t (initially *NotStarted*) that is used to model the control flow of transaction t. When t is in flight, but not executing any operation we have $pc_t = Ready$. Similarly, $pc_t = Aborted$ and $pc_t = Committed$ iff t has aborted or committed, respectively. Otherwise, pc_t is a line number corresponding to the instruction of the operation t is executing.

We assume each operation $op \in \{\texttt{TMBegin}, \texttt{TMRead}(x), \texttt{TMWrite}(x, v), \texttt{TMCommit}\}$ generates an event $inv_t(op)$ when op starts executing and $res_t(op)$, when op completes.

Ensuring Well-Formed Histories. As described in §3.1, to ensure well-formedness of histories, we must ensure that transaction identifiers are not reused. Additionally, a live (i.e., in-flight) transaction before a crash must not continue its execution after the crash. To this end, we implicitly assume a *persistent transaction manager* that allocates new transaction identifiers. As with the dTML$_{\mathsf{SC}}$ model, we use program counters to concisely characterize this assumption. First note that we assume program counter values of all threads except the system thread are unchanged after a CRASH transition (see Fig. 4.2), thus any transaction t with $pc_t = NotStarted$ can be executed after a crash. To ensure that in-flight transactions are not resumed, we assume that recovery starts by setting pc_t to *Aborted* for every transaction t such that $pc_t \notin \{NotStarted, Aborted, Committed\}$ (cf. `TMCrashRecovery` in Fig. 3.5) .

Modelling log Operations. The final source of complexity is the *durably linearizable* [81] log, *log*. The *log* and its associated operations are precisely modeled in accordance with the description provided in §3.4.2; therefore, we omit its description here. An actual implementation of *log* may synchronize threads, e.g., with **mfence** operations, which affects the persistency and thread views of the variables of dTML$_{\text{Px86}}$. Our proof makes no such assumptions about *log*. Namely, we assume the *weakest possible* ordering guarantees. Thus, an implementation of *log* that performs thread synchronization using fences would not affect the soundness of our result.

6.4 Invariants of dTML$_{\mathbf{Px86}}$ ()

This section describes the key invariants of dTML$_{\text{Px86}}$ and mechanisms for proving their correctness. These will be used in the simulation proof in §6.5. Our work builds on the PIEROGI$_{\text{full}}$ logic which uses view-based expressions derived from the *view* components of the thread state. To elaborate, we use the PIEROGI$_{\text{full}}$ proof rules (see §4.2) combined with the Owicki-Gries style proof method demonstrated to establish the correctness of assertions. In §6.4.1 we present the crash invariant and in §6.4.2 we present the dTML$_{\text{Px86}}$ program annotation.

6.4.1 dTML$_{\mathbf{Px86}}$ Crash Invariant

To prove the correctness of dTML$_{\text{Px86}}$ implementation, we construct a multithreaded program $\Pi_{\text{dTML}_{\text{Px86}}}$ based on the model that is introduced in section §6.3. Program $\Pi_{\text{dTML}_{\text{Px86}}}$ includes all dTML$_{\text{Px86}}$ operations, invocation events, response events and the system crash event. With the exception of the system thread, which is only capable of executing the `TMRecover` operation, any thread t in TID is free to perform any number of operations (excluding the recovery operation) as long as the resulting execution history conforms to the control flow and well-formedness constraints.

In this section, we analyze the crash invariant I of dTML$_{\text{Px86}}$. We have shown that the crash invariant holds at any program transition of $\Pi_{\text{dTML}_{\text{Px86}}}$. The corresponding proofs are mechanized in Isabelle/HOL. The crash invariant constitutes a collection of properties that the dTML$_{\text{Px86}}$ implementation preserves. These properties are used to prove that dTML$_{\text{Px86}}$ is durably opaque. The most important ones are as follows.

Memory properties. The first three properties describe memory patterns that occur during the execution of dTML$_{\text{Px86}}$. In the properties below, assume that $i, j \in \mathbf{dom}(M)$ such that $i < j$ are arbitrarily chosen.

Property 1. The values of glb are monotonically increasing, i.e.,

$$\forall v_i, v_j.\ M[i] \equiv \langle \mathsf{glb} := v_i \rangle \wedge M[j] \equiv \langle \mathsf{glb} := v_j \rangle \implies v_i \leq v_j$$

Property 1 states that the values of `glb` are monotonically increasing. In contrast to prior work [23], the recovery process does not reset `glb` in the event of a crash. This is necessary in order to avoid `TMRead` operations returning stale values (values that were in persistent memory, but subsequently modified) after a crash. To demonstrate this phenomenon, consider the program in Fig. 6.1 that resets glb to zero (**store** glb 0) after *Rec*6 instead of executing lines *Rec*7 − *Rec*10. An execution of this program can reach a state with the following memory sequence:

$$\langle \{\mathsf{glb} \mapsto 2, x \mapsto 5, \_ \mapsto 0\}, \langle x := 3\rangle, \langle \mathsf{glb} := 0\rangle, \langle \mathsf{glb} := 1\rangle, \langle y := 1\rangle, \langle \mathsf{glb} := 2\rangle\rangle$$

which is reached from the initial state post-crash after a **(1)** a writing transaction updates x to 3 then commits (so glb = 2), **(2)** another writing transaction writes updates x to 5 (so $\mathit{log}(x) = 3$), **(3)** a crash occurs (resulting in the intial state above), **(4)** the modified recovery operation described above executes (appending $\langle x := 3\rangle$ then $\langle \mathsf{glb} := 0\rangle$ to the memory), **(5)** a third writing transaction that updates y to 1 commits successfully.

Now assume that another transaction t starts, then reads 2 for glb from the *initial message*, allowing it to complete `TMBegin`, then performs a `TMRead`(x) operation. In this case, according to Px86 semantics, the initial value of x (i.e., 5) is still observable at *Rp*. The test at $R1$ succeeds and the **CAS** instruction at $R2$ can still succeed, since the last value of glb is 2. As a result, t can successfully complete the `TMRead` operation and subsequently commit, violating durable opacity.

Property 2. If there exists a write between two writes to glb such that the value of glb is unchanged, then the location of any intermediate write between these two writes must be on glb, i.e.,

$$(\exists v.\ M[i] \equiv \langle \mathsf{glb} := v\rangle \wedge M[j] \equiv \langle \mathsf{glb} := v\rangle) \quad\Longrightarrow\quad \forall k \in [i, j].\ M[k].\mathsf{loc} = \mathsf{glb}$$

Property 2 holds since this memory pattern can only occur when two or more transactions that have not performed any read or write yet, perform a `TMRead` by successfully executing their **CAS** instruction at $R2$. The first of these reading transactions introduces a write to glb that immediately follows either **(1)** the initial message, or **(2)** a write to glb by a writing transaction at $C3$, or **(3)** a message added by the `TMRecover` process at *Rec*9 or *Rec*10.

Property 3. Between a memory message on glb with even value and another memory message on a location different from glb, there exists a message with location on glb with odd value, i.e.,

$$i > 0 \wedge M[i].\mathsf{loc} = \mathsf{glb} \wedge \mathit{even}(M[i].\mathsf{val}) \wedge M[j].\mathsf{loc} \neq \mathsf{glb} \implies \exists k \in (i, j).\ M[x].\mathsf{loc} = \mathsf{glb} \wedge \mathit{odd}(M[x].\mathsf{val})$$

Property 3 describes the memory pattern that occurs when a transaction performs successfully a `TMWrite`. Note that excluding the initial message and the messages added from the recovery process, the only way that messages with a location different from glb are added to the memory is by executing $W7$. Prior to this point of execution, the writing transaction performs a successful **CAS** at $W1$. The execution of $W1$ adds a message to memory with location glb and odd value.

The next property uses $\mathsf{maxcoh}_t \triangleq \lambda\sigma.\ \bigsqcup_x (\sigma.\mathbb{T}(t)).\mathsf{coh}(x)$ and $\mathsf{vrnew}_t \triangleq \lambda\sigma.\ (\sigma.\mathbb{T}(t)).\mathsf{v}_{\mathsf{rNew}}$.

Property 4. When a `TMRecover` process is not in progress, for any transaction that is not a writing transaction, the coherence view for all the locations in memory is less than or equal to its $\mathsf{v}_{\mathsf{rNew}}$ view, i.e., $\forall t \in \textsc{Tid}.\ \neg\mathsf{Recovering} \wedge \mathsf{writer} \neq t \implies \mathsf{maxcoh}_t \leq \mathsf{vrnew}_t$,.

Property 4 describes partially the thread state of not writing transactions while a recovery process is not in progress. Specifically, it states that in any given state, for any not-writing transaction t, the vrnew_t timestamp is greater or equal to the maximum among the coherence view timestamps. This holds because the only cases in which $\mathsf{coh}_t\ x > \mathsf{vrnew}_t$, is when t is executing a write on x or an internal read to x. Both cases are precluded for not writing transactions.

Properties about tracked locations and *log*. We now describe a set of properties describing the memory locations that are tracked by Px86 and *log*. Note that we assume that all locations in $\textsc{Loc}$ different from glb can be transactionally written and read.

Property 5. The domain of the persistent *log* does not contain the location glb, i.e.,

$\forall x \in \mathbf{dom}(\mathit{log}).\ x \neq \mathsf{glb}$.

Property 6. For all locations $x \neq \mathsf{glb}$ that is not in *log*, the persistent view includes only their last written value, i.e., $\forall x \in \textsc{Loc}.\ x \neq \mathsf{glb} \wedge x \notin \mathbf{dom}(\mathit{log}) \implies [x]^{\mathsf{P}} = \{\vec{x}\}$.

Properties about glb and recGlb. Next, we have three properties over glb and the auxiliary variable recGlb.

Property 7. When a writing transaction t is live the last value of glb in the memory must be odd, i.e., $\mathsf{writer} \neq \mathit{None} \implies \mathit{odd}(\overrightarrow{\mathsf{glb}})$.

Property 7 holds due to the successful execution of $W1$. Note that the implication does not hold in the other direction because, in our model, we reset the auxiliary variable writer to *None* during a crash, yet the last value of glb after a crash may be odd. One could have defined a stronger invariant: $\neg\mathsf{Recovering} \implies (\mathsf{writer} \neq \mathit{None} \Leftrightarrow \mathit{odd}(\overrightarrow{\mathsf{glb}}))$, however, we have not needed this strengthening in our proofs.

Property 8. With the exception of the initial message, the value of glb is greater than or equal to recGlb, i.e., $\forall i \in \mathbf{dom}(M).\ 0 < i \wedge M[i].\mathsf{loc} = \mathsf{glb} \implies M[i].\mathsf{val} \geq \mathsf{recGlb}$.

Property 9. After a transaction t successfully executes `TMBegin`, the value of loc_t must be less than or equal to $(\overrightarrow{\mathsf{glb}})$. Moreover, after a successful `TMWrite` and/or `TMRead` operation has taken place (i.e., $\mathsf{hasRead}_t \vee \mathsf{hasWritten}_t$ holds), the value of recGlb is less than or equal to loc_t, i.e.,

$$\begin{aligned}\forall t \in \textsc{Tid}.\ &(\mathit{pc}_t \notin \{\mathit{NotStarted}, \mathit{Bp}, B1, B2, \mathit{Aborted}, \mathit{Committed}\} \implies \mathsf{loc}_t \leq \overrightarrow{\mathsf{glb}}) \wedge \\ &((\mathsf{hasRead}_t \vee \mathsf{hasWritten}_t) \implies \mathsf{recGlb} \leq \mathsf{loc}_t)\end{aligned}$$

Properties About Recovery. Finally, we have a set of properties about the state immediately after a crash (before recovery has begun) and after recovery has finished.

Property 10. When a `TMRecover` process is in progress, all the transactions are either *NotStarted*, *Aborted* or *Committed*, i.e.,

$\mathsf{Recovering} \implies (\forall t.\ \mathit{pc}_t \in \{\mathit{NotStarted}, \mathit{Aborted}, \mathit{Committed}\})$.

Property 11. When a `TMRecover` process is not in progress (i.e., has completed), the value of glb in the initial message is less than the value of the auxiliary variable recGlb, which in turn is at most the final value of the value of recGlb, and the value of $\mathit{even}(\mathsf{recGlb})$ is even, i.e.,

$$\neg\mathsf{Recovering} \implies M[0](\mathsf{glb}) < \mathsf{recGlb} \wedge \mathsf{recGlb} \leq \overrightarrow{\mathsf{glb}} \wedge \mathit{even}(\mathsf{recGlb}).$$

6.4.2 dTML$_{\mathbf{Px86}}$ Program Annotation

We now enumerate the local properties of each thread by adding program annotations at each atomic step. The program annotation is formed by view-based expressions (see §4.2.1). The assertions of dTML$_{\mathsf{SC}}$ can be classified into three categories: **(1)** transactions that have not yet performed a read or a write (green assertions), **(2)** *read-only* transactions (pink assertions), and **(3)** *writing* transactions (blue assertions). The assertions highlighted yellow in Figs. 6.2, 6.4 and 6.5 capture the effects of the preceding instruction.

We define an assertion ready_t, which holds when an in-flight transaction is in an idle state (i.e., not executing any transactional operation):

$$\begin{aligned}\mathsf{ready}_t = {} & \Big(\neg\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge \mathit{even}(\mathsf{loc}_t) \wedge \mathsf{writer} \neq t \wedge \big(\mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \implies \forall y.[y]_t = \{\vec{y}\}\big)\Big) \\ & \vee \big(\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge \mathit{even}(\mathsf{loc}_t) \wedge \mathsf{writer} \neq t \wedge (\forall y.\ y \neq \mathsf{glb} \implies \mathsf{read}_{\mathsf{pre}}(t, y))\big) \\ & \vee \begin{pmatrix}\mathsf{hasWritten}_t \wedge \mathit{odd}(\mathsf{loc}_t) \wedge \mathsf{writer} = t \wedge \mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \wedge \\ (\forall y.[y]_t = \{\vec{y}\}) \wedge (\forall y \in \mathbf{dom}(\mathit{log}).\ [y]_t^{\mathsf{A}} = \{\vec{y}\})\end{pmatrix}\end{aligned}$$

The first disjunct captures two local conditions of t: that the local snapshot of glb is even and the writer is not t, as well as a visibility guarantee that if t's the local snapshot of glb is consistent with the last write to glb, then the thread's view of each location y is the last write to y. The visibility guarantee ensures that the transaction t can be serialized after the last writing transaction in case t successfully performs its reads and commits.

The second disjunct covers read-only transactions as described in §6.2 using the predicate $\mathsf{read}_{\mathsf{pre}}(t, y)$ below. We let $\mathsf{coh}_t(x) \triangleq \lambda\sigma.\ (\sigma.\mathbb{T}(t)).\mathsf{coh}(x)$.

$$\mathsf{read}_{\mathsf{pre}}(t, y) = \mathsf{coh}_t\ \mathsf{glb} > 0 \wedge M[\mathsf{coh}_t\ \mathsf{glb}] \equiv \langle \mathsf{glb} := \mathsf{loc}_t \rangle$$

Predicate $\mathsf{read}_{\mathsf{pre}}(t, y)$ is established a successful CAS-SUCCESS transition (see Fig. 4.3). Namely, the successful **CAS** transition at $R2$ in Fig. 6.4 shifts the coherence view of glb to the length of the memory in the pre-state, which is greater than zero since the memory includes always the initial message. Furthermore, the second conjunct of $\mathsf{read}_{\mathsf{pre}}$ holds because the successful **CAS** transition appends the message $\langle \mathsf{glb} := \mathsf{loc}_t \rangle$ to the end of the memory.

The third disjunct of ready is straightforward since a writing transaction takes the lock (by making glb odd). The only additional guarantee required is that the t's asynchronous view of each location in log is maximal. This guarantees that when the transaction later performs an **sfence** at $C1$, all of the writes performed by the transaction are persisted.

We define an assertion ready_t, which holds when an in-flight transaction is in an idle state (i.e., not executing any transactional operation):

$$
\begin{aligned}
&\mathsf{ready}_t = \\
&\Big(\neg\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge \mathit{even}(\mathsf{loc}_t) \wedge \mathsf{writer} \neq t \wedge \big(\mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \implies \forall y. [y]_t = \{\vec{y}\}\big)\Big) \\
&\vee\ (\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge \mathit{even}(\mathsf{loc}_t) \wedge \mathsf{writer} \neq t \wedge (\forall y.\ y \neq \mathsf{glb} \implies \mathsf{read}_{\mathsf{pre}}(t, y))) \\
&\vee\ \Big(\mathsf{hasWritten}_t \wedge \mathit{odd}(\mathsf{loc}_t) \wedge \mathsf{writer} = t \wedge \mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \wedge (\forall y. [y]_t = \{\vec{y}\}) \wedge (\forall y \in \mathbf{dom}(\mathit{log}).\ [y]_t^{\mathsf{A}} = \{\vec{y}\})\Big)
\end{aligned}
$$

The first disjunct captures two local conditions of t: that the local snapshot of glb is even and the writer is not t, as well as a visibility guarantee that if t's the local snapshot of glb is consistent with the last write to glb, then the thread's view of each location y is the last write to y. The visibility guarantee ensures that the transaction t can be serialized after the last writing transaction in case t successfully performs its reads and commits.

The second disjunct covers read-only transactions as described in §6.2 using the predicate $\mathsf{read}_{\mathsf{pre}}(t, y)$ below. We let $\mathsf{coh}_t(x) \triangleq \lambda\sigma.\ (\sigma.\mathbb{T}(t)).\mathsf{coh}(x)$.

$$\mathsf{read}_{\mathsf{pre}}(t, y) = \mathsf{coh}_t\ \mathsf{glb} > 0 \wedge M[\mathsf{coh}_t\ \mathsf{glb}] \equiv \langle \mathsf{glb} := \mathsf{loc}_t \rangle$$

Predicate $\mathsf{read}_{\mathsf{pre}}(t, y)$ is established a successful **CAS** transition (see Fig. 4.3). Namely, the successful **CAS** transition at $R2$ in Fig. 6.4 shifts the coherence view of glb to the length of the memory in the pre-state, which is greater than zero since the memory includes always the initial message. Furthermore, the second conjunct of $\mathsf{read}_{\mathsf{pre}}$ holds because the successful **CAS** transition appends the message $\langle \mathsf{glb} := \mathsf{loc}_t \rangle$ to the end of the memory.

The third disjunct of ready is straightforward since a writing transaction takes the lock (by making glb odd). The only additional guarantee required is that the t's asynchronous view of each location in log is maximal. This guarantees that when the transaction later performs an **sfence** at $C1$, all of the writes performed by the transaction are persisted.

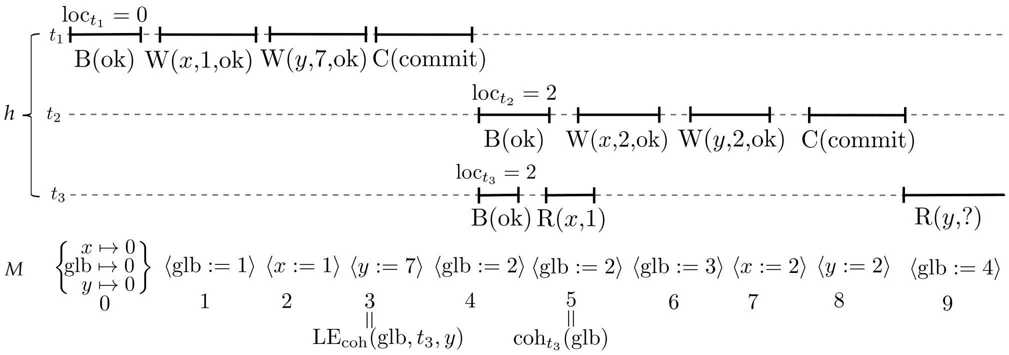
The TMBegin annotation. We start with discussing the annotation of the `TMBegin` operation. In the initial state, all the registers are initialized to zero, therefore both the $\mathsf{hasWritten}_t$ and $\mathsf{hasRead}_t$ registers are set to zero ($false$). The implication at $PB1$ states that if the value read for glb is even and is consistent with the last write of glb, then t's thread view for every location contains only its last value. The program annotations for `TMRead` and `TMWrite` guarantee that a subsequent read or write operation can only succeed if loc_t remains consistent with the last value of glb after the execution of Bp. $PB1$ is adequate to establish ready_t, in particular its first disjunct.

The TMRead annotation. Perhaps the most important part of the `TMRead` annotation (Fig. 6.4) lies in explicitly identifying the specific loaded value obtained at Rp that leads to a successful `TMRead` operation. Supposing that a transaction t executes a `TMRead` operation, there are three distinct cases.

- If t has not performed a read or a write yet, then the ready assertion specifies that in case loc_t is equal to the last value of glb and the thread view for all the locations

```
TMBegin
$PBp$ : $\{\neg\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge \mathsf{writer} \neq t\}$
  $Bp$ : do $\mathsf{loc}_t :=$ load glb
$PB1$ : $\left\{\begin{pmatrix}\neg\mathsf{hasRead}_t \wedge \neg\mathsf{hasWritten}_t \wedge \mathsf{writer} \neq t \\ \wedge\ (even(\mathsf{loc}_t) \implies (\mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \implies (\forall y.\ [y]_t = \{\vec{y}\})))\end{pmatrix}\right\}$
  $B1$ : until $even(\mathsf{loc}_t)$;
         return $ok$;  $\{\mathsf{ready}_t\}$
```

Figure 6.2: `TMBegin` annotation



$\mathrm{loc}_{t_1} = 0$
t_1
B(ok)
W(x,1,ok)
W(y,7,ok)
C(commit)
h
$\mathrm{loc}_{t_2} = 2$
t_2
B(ok)
W(x,2,ok)
W(y,2,ok)
C(commit)
$\mathrm{loc}_{t_3} = 2$
t_3
B(ok)
R(x,1)
R(y,?)
M
$\left\{\begin{matrix} x \mapsto 0 \\ \mathrm{glb} \mapsto 0 \\ y \mapsto 0 \end{matrix}\right\}$
$\langle \mathrm{glb} := 1\rangle$
$\langle x := 1\rangle$
$\langle y := 7\rangle$
$\langle \mathrm{glb} := 2\rangle$
$\langle \mathrm{glb} := 2\rangle$
$\langle \mathrm{glb} := 3\rangle$
$\langle x := 2\rangle$
$\langle y := 2\rangle$
$\langle \mathrm{glb} := 4\rangle$
0
1
2
3
4
5
6
7
8
9
$\mathrm{LE}_{\mathrm{coh}}(\mathrm{glb}, t_3, y)$
$\mathrm{coh}_{t_3}(\mathrm{glb})$

Figure 6.3: Example execution for read-only transactions

in memory contains only their last writes. In such case, in the post-state of Rp it is guaranteed that the loaded value corresponds to the last write at x ($r_t = \vec{x}$). In case that loc_t is not equal to the last value of glb, the **CAS** instruction at $R2$ fails and thus t aborts.

- If t is a read-only transaction, the valid value to be read for x is deterministic and corresponds to the value of the memory message with timestamp $\mathsf{LE}_{\mathsf{coh}}(\mathsf{glb}, t, x)$. To establish this, we prove an additional lemma that demonstrates the following: If $\mathsf{read}_{\mathsf{pre}}(t, x)$, Property 1, Property 3 and Property 8 hold in the pre-state, then, upon executing $r_t := \mathbf{load}\, x$, loc_t is visible to t on location glb if the loaded value matches the value of the message $M[\mathsf{LE}_{\mathsf{coh}}(\mathsf{glb}, t, x)]$ (i.e., $\mathsf{loc}_t \in [\mathsf{glb}]_t \implies M[\mathsf{LE}_{\mathsf{coh}}(\mathsf{glb}, t, x)] \equiv \langle x, r_t\rangle$).

- If t is a writing transaction, its thread views of all the locations in memory consist solely of their most recent writes. This is due to the fact that t had previously executed a successful **CAS** instruction at $W1$, which effectively constrains the visibility of t to its last writes for all locations. Consequently, in this scenario, the `TMRead` operation consistently succeeds, and $r_t = \vec{x}$.

We now discuss in more detail, the correctness proof of read-only transactions, which is the most challenging aspect of the proof. We use the example in Fig. 6.3 with an abstract history h comprising three transactions t_1-t_3. Transactions t_1 and t_2 can not be reordered due to the *real-time* order constraint of durable opacity (see Def. 3.1.1). Moreover, since the first read of transaction t_3 has returned t_1 for x, the only valid sequential history corresponds to the ordering ($t_1 \prec t_3 \prec t_2$). Thus, the second read in transaction t_3 must either return 7 for y, or abort.

In the implementation, we must identify the timestamp of the write that a read-only transaction reads from and does not lead to an abort. To this end, let $\mathsf{vrnew}_t(x) \triangleq$

$\lambda\sigma.\ (\sigma.\mathbb{T}(t)).\mathsf{v_{rNew}}$ In the example in Fig. 6.3, we have $\mathsf{LE_{coh}}(\mathsf{glb}, t_3, y) = 3$ since t_3's coherence view of glb is memory index 5, and the last write to y before index 5 is at index 3. Note that we instantiate this to $\mathsf{LE_{coh}}(\mathsf{glb}, t_3, y)$ in the second disjunct of $PR1$ in Fig. 6.4.

Provided that t_3 reads from the memory at index 3, the message at index 5 will still be observable to t_3. Therefore, it can read this message at $R4$ so that the check at $R5$ does not fail. We can prove that the second read of t_3 can only succeed if it reads from index 3 by contradiction.

Case 1: t_3 reads a message with timestamp greater than $\mathsf{LE_{coh}}(\mathsf{glb}, t_3, y)$ at Rp. In Fig. 6.3, the only such message is at index 8. Using Property 3, in the post-state of Rp, there exists a timestamp ts between $\mathsf{coh}_{t_3}(\mathsf{glb})$ (i.e., 5) and $\mathsf{coh}_{t_3}(y)$ (i.e., 8), such that $M[ts] \equiv \langle \mathsf{glb} := v\rangle$ and $odd(v)$. In our example, $ts = 6$.

Every observable timestamp for glb can be shown to be greater than or equal to ts (i.e. 6) and thus greater than $\mathsf{coh}_{t_3}(\mathsf{glb})$ (i.e., 5) by using Property 4. Transaction t_3 must read a value for glb at $R4$ that is different from loc_{t_3}. By the third conjunct of the pink disjunct of ready_{t_3}, we have $even(\mathsf{loc}_{t_3})$. Moreover by Property 1, each value of glb after ts is at least v. Since $odd(v)$, we have $v \neq \mathsf{loc}_{t_3}$, thus t_3 cannot observe loc_{t_3} for glb.

Case 2: t_3 reads a message with timestamp less than $\mathsf{LE_{coh}}(\mathsf{glb}, t_3, y)$ at Rp.

In Fig. 6.3, such a message is the initial message (with timestamp 0). By Property 4, vrnew_t must be at least $\mathsf{coh}_{t_3}(\mathsf{glb})$ (i.e., timestamp 5). However, $\mathsf{LE_{coh}}(\mathsf{glb}, t_3, y)$ overwrites this earlier message, and hence the earlier timestamp is no longer visible to t_3.

The TMWrite annotation. Figure 6.5 depicts the `TMWrite` annotation. The check performed at Wp determines whether a transaction t has previously executed a write operation. If the number of locations loc_t written by t is even, it means that t has not performed any writes yet. Consequently, $PW1$ asserts that $\mathsf{hasWritten}_t$ is $false$ and $\mathsf{writer} \neq t$. Additionally, $PW1$ ensures that if t is a writer, the asynchronous view for t of all locations in log, except for location x, is maximal. This guarantees that if t becomes a writing transaction and executes an sfence at $C1$, all of its writes will persist. Location x is excluded because, between the write operation at x ($W7$) and the asynchronous flush of the new write ($W8$), the asynchronous view of x contains both its old value and the newly written value ($\vec{x}$). Finally, $PW1$ states that the address to be written (x) is not equal to glb, which is necessary to establish Property 5.

Next, t attempts to acquire the single global versioned lock glb by executing **CAS** at $W1$. A successful **CAS** operation sets the $\mathsf{hasWritten}_t$ register to $true$, indicating that t has become a writing transaction. As stated in $PW2$, in this case, the last written value at glb is set to loc_t incremented by one, and the thread view of t for all memory locations is updated to include only their last written values. On the other hand, if **CAS** fails, it indicates the presence of another concurrent writing transaction, causing t to abort.

The subsequent execution of $W3$ increments loc_t by one. Therefore, according to $PW4$, loc_t becomes equal to the last value of glb. Additionally, the auxiliary variable writer is set to t.

Lines $W4 - W9$ encompass the following operations: updating the log ($W4 - W6$), performing the write at x ($W7$), and subsequently asynchronously flushing it ($W8$).

The corresponding assertions remain unchanged, except for the final condition of $PW6$. This condition states that x is going to be updated to its last written value (c_t) in log. Establishing the condition ready, particularly its third disjunct, from $PW8$ is straightforward. It should be noted that after the execution of $W8$, the asynchronous view of x contains only its last written value (as per the OP rule in Fig. 4.7). The combination of the above condition with $PW8$ is sufficient to establish that the asynchronous view of t for all locations in the domain of log contains only their last written value $((\forall y \in \mathbf{dom}(log).\ x \neq y \implies [y]_t^{\mathsf{A}} = \vec{y})$.

The TMCommit annotation. Fig. 6.6 illustrates the `TMCommit` annotation. Transactions that have not performed any read or write and read-only transactions commit without any further check. In the case of a writing transaction t, according to the assertion ready, loc_t is odd, $\mathsf{hasWritten}_t$ is true, $\mathsf{writer} = t$, the thread view of t for any location y includes only the last stored value at y, and the asynchronous view of t for any location y that belongs to the domain of log includes only the last stored value at y. It is worth noting that the locations that belong to the domain of log, are the only locations that have been updated by a writing transaction t. As seen at the postcondition of $C1$, after the execution of **sfence** the asynchronous views of t for the aforementioned locations become equal to their persistent views. Having the above stated prior to emptying the log (i.e. at $PC2$) is sufficient for establishing locally Property 6. Property 6, guarantees that during the execution and commit of read-only transactions, and after writing transactions commit, the value can be observed in persistent memory for any location x apart from glb is deterministic and equal to the last written value on x.

The TMRecover annotation. Fig. 6.7 illustrates the `TMRecover` annotation. The `TMRecover` annotation serves three purposes. **(1)** It provides sufficient information about the memory state after a crash event and during the `TMRecover` process, in order to establish that Property 1 and Property 2 locally hold. The above is enabled by the assertion: $\forall ts \in \mathbf{dom}(M).ts > 0 \implies M[ts].\mathsf{loc} \neq \mathsf{glb}$, which ensures that during recovery, all the memory messages, apart from the initial one represent writes to locations different from glb. For showing this during copying and emptying the log we use the Property 5. **(2)** It guarantees the consistency of memory upon completion of the recovery process. This is accomplished by Property 6 in combination with rules $\mathsf{C1}$ and $\mathsf{C3}$ (see Figure 4.9). By applying Property 6 and the aforementioned rules, any location y within the initial message is mapped to its persisted value $\vec{y}$, which represents the last value written to y by a committed transaction prior to the system crash. Moreover, the recovery process sequentially restores all the locations recorded in the log. The `TMRecover` annotation guarantees that the recovered values correspond to those stored in the log. **(3)** It guarantees that by the completion of the recovery process, the last written value in glb is even and greater than its initial value.

Theorem 6.4.1 (). *The proof outline for $dTML_{Px86}$ is valid.*

```
$\texttt{TMRead}(x)$
$PRp : \{\mathsf{ready}_t\}$
  $Rp : r_t \mathrel{:=} \mathbf{load}\, x;$
$PR1 : \left\{ \begin{array}{l} \left(\neg\mathsf{hasRead}_t \land \neg\mathsf{hasWritten}_t \land \mathit{even}(\mathsf{loc}_t) \land \mathsf{writer} \neq t \land (\mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \implies r_t = \vec{x}\,\})\right) \\ \lor \left(\begin{array}{l} \mathsf{hasRead}_t \land \neg\mathsf{hasWritten}_t \land \mathit{even}(\mathsf{loc}_t) \land \mathsf{writer} \neq t \land \\ (\mathsf{loc}_t \in [\mathsf{glb}]_t \implies M[\mathsf{LE}_{\mathsf{coh}}(\mathsf{glb}, t, x)] \equiv \langle x, r_t\rangle \land (\forall y.\ y \neq \mathtt{glb} \implies \mathsf{read}_{\mathsf{pre}}(t, y))) \end{array}\right) \\ \lor \left(\begin{array}{l} \mathsf{hasWritten}_t \land \mathit{odd}(\mathsf{loc}_t) \land \mathsf{writer} = t \land \mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \land \\ r_t = \vec{x} \land (\forall y.\ [y]_t = \{\vec{y}\}) \land (\forall y \in \mathbf{dom}(\mathit{log}).\ [y]_t^{\mathsf{A}} = \{\vec{y}\}) \end{array}\right) \end{array} \right\}$
  $R1 : \mathbf{if}\ \mathit{even}(\mathsf{loc}_t) \land \neg\mathsf{hasRead}_t\ \mathbf{then}$
$PR2 : \left\{ \neg\mathsf{hasRead}_t \land \neg\mathsf{hasWritten}_t \land \mathit{even}(\mathsf{loc}_t) \land \mathsf{writer} \neq t \land (\mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \implies r_t = \vec{x}) \right\}$
  $R2 :$    $\mathsf{hasRead}_t := \mathbf{CAS}\ \mathsf{glb}\ \mathsf{loc}_t\ \mathsf{loc}_t$
$PR3 :$    $\{\mathsf{hasRead}_t \Rightarrow \mathsf{ready}_t\}$
  $R3 :$    $\mathbf{if}\ \mathsf{hasRead}_t\ \mathbf{then}$
$PRa :$    $\{\mathsf{ready}_t\}$
  $Rr :$      $\mathbf{return}\ r_t$  $\{\mathsf{ready}_t\}$
           $\mathbf{else\ return}\ \mathit{abort}$  $\{\mathsf{true}\}$
$PR4 : \left\{ \begin{array}{l} \left(\begin{array}{l} \mathsf{hasRead}_t \land \neg\mathsf{hasWritten}_t \land \mathit{even}(\mathsf{loc}_t) \land \mathsf{writer} \neq t \land x \neq \mathsf{glb} \land \\ (\mathsf{loc}_t \in [\mathsf{glb}]_t \implies M[\mathsf{LE}_{\mathsf{coh}}(\mathsf{glb}, t, x)] \equiv \langle x, r_t\rangle \land (\forall y.\ y \neq \mathtt{glb} \implies \mathsf{read}_{\mathsf{pre}}(t, y))) \end{array}\right) \\ \lor \left(\begin{array}{l} \mathsf{hasWritten}_t \land \mathit{odd}(\mathsf{loc}_t) \land \mathsf{writer} = t \land \mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \land x \neq \mathsf{glb} \land \\ r_t = \vec{x} \land (\forall y.\ [y]_t = \{\vec{y}\}) \land (\forall y \in \mathbf{dom}(\mathit{log}).\ [y]_t^{\mathsf{A}} = \{\vec{y}\}) \end{array}\right) \end{array} \right\}$
  $R4 : c_t \mathrel{:=} \mathbf{load}\, \mathsf{glb};$
$PR5 : \left\{ \begin{array}{l} \left(\begin{array}{l} \mathsf{hasRead}_t \land \neg\mathsf{hasWritten}_t \land \mathit{even}(\mathsf{loc}_t) \land \mathsf{writer} \neq t \land x \neq \mathsf{glb} \land \\ (\, c_t = \mathsf{loc}_t \implies M[\mathsf{LE}_{\mathsf{coh}}(\mathsf{glb}, t, x)] \equiv \langle x, r_t\rangle \land (\forall y.\ y \neq \mathsf{glb} \implies \mathsf{read}_{\mathsf{pre}}(t, y))) \end{array}\right) \\ \lor \left(\begin{array}{l} \mathsf{hasWritten}_t \land \mathit{odd}(\mathsf{loc}_t) \land \mathsf{writer} = t \land x \neq \mathsf{glb} \land \mathsf{loc}_t = \overrightarrow{\mathsf{glb}} \land \\ r_t = \vec{x} \land c_t = \mathsf{loc}_t \land (\forall y \in \mathbf{dom}(\mathit{log}).\ [y]_t^{\mathsf{A}} = \{\vec{y}\}) \end{array}\right) \end{array} \right\}$
  $R5 : \mathbf{if}\ c_t = \mathsf{loc}_t\ \mathbf{then}$
$PRb : \{\mathsf{ready}_t\}$
  $Rr :$    $\mathbf{return}\ r_t$  $\{\mathsf{ready}_t\}$
  $Ab : \mathbf{else\ return}\ \mathit{abort};\{\mathsf{true}\}$
```

Figure 6.4: TMRead annotation

```
$\texttt{TMWrite}(x, v)$
$PWp : \{\mathsf{ready}_t\}$
  $Wp :$ if $even(\mathsf{loc}_t)$ then
$PW1 : \left\{ \begin{array}{l} even(\mathsf{loc}_t) \land \neg\mathsf{hasWritten}_t \land \mathsf{writer} \neq t \\ \land\ (\mathsf{writer} = t \implies (\forall y \in \mathbf{dom}(log).\ x \neq y \implies [y]_t^{\mathsf{A}} = \{\vec{y}\})) \land x \neq \mathsf{glb} \end{array} \right\}$
  $W1 :$    $\mathsf{hasWritten}_t := \mathbf{CAS}\ \mathsf{glb}\ \mathsf{loc}_t\ (\mathsf{loc}_t + 1);$
$PW2 :$    $\left\{ \begin{array}{l} (\mathsf{hasWritten}_t \implies (\forall y.\ [y]_t = \{\vec{y}\}) \land Succ(\mathsf{loc}_t) = \vec{\mathsf{glb}}) \\ \land(\mathsf{writer} = t \implies (\forall y \in \mathbf{dom}(log).\ x \neq y \implies [y]_t^{\mathsf{A}} = \{\vec{y}\})) \land x \neq \mathsf{glb} \end{array} \right\}$
  $W2 :$    if $\mathsf{hasWritten}_t$ then
$PW3 :$    $\left\{ \begin{array}{l} \mathsf{hasWritten}_t \land (\forall y.\ [y]_t = \{\vec{y}\}) \land Succ(\mathsf{loc}_t) = \vec{\mathsf{glb}} \\ \land(\mathsf{writer} = t \implies (\forall y \in \mathbf{dom}(log).\ x \neq y \implies [y]_t^{\mathsf{A}} = \{\vec{y}\})) \land x \neq \mathsf{glb} \end{array} \right\}$
  $W3 :$       $\langle x_t := loc_t + 1,\ \mathsf{writer} := t \rangle$
           else return $abort$; $\{\mathsf{true}\}$
$PW4 : \left\{ \begin{array}{l} odd(\mathsf{loc}_t) \land \mathsf{loc}_t = \vec{\mathsf{glb}} \land (\forall y \in \mathbf{dom}(log).\ x \neq y \implies [y]_t^{\mathsf{A}} = \{\vec{y}\}) \land \mathsf{writer} = t \\ \land\mathsf{hasWritten}_t \land (\forall y.\ [y]_t = \{\vec{y}\}) \land x \neq \mathsf{glb} \end{array} \right\}$
  $W4 :$ if $\neg log.\mathbf{contains}(x)$ then
$PW5 : \left\{ \begin{array}{l} odd(\mathsf{loc}_t) \land \mathsf{loc}_t = \vec{\mathsf{glb}} \land (\forall y \in \mathbf{dom}(log).\ x \neq y \implies [y]_t^{\mathsf{A}} = \{\vec{y}\}) \land \mathsf{writer} = t \\ \land\mathsf{hasWritten}_t \land (\forall y.\ [y]_t = \{\vec{y}\}) \land x \neq \mathsf{glb} \end{array} \right\}$
  $W5 :$    $c_t := \mathbf{load}\ x;$
$PW6 :$    $\left\{ \begin{array}{l} odd(\mathsf{loc}_t) \land \mathsf{loc}_t = \vec{\mathsf{glb}} \land (\forall y \in \mathbf{dom}(log).\ x \neq y \implies [y]_t^{\mathsf{A}} = \{\vec{y}\}) \land \mathsf{writer} = t \\ \land\mathsf{hasWritten}_t \land (\forall y.\ [y]_t = \{\vec{y}\}) \land x \neq \mathsf{glb} \land c_t = \vec{x} \end{array} \right\}$
  $W6 :$    $log.\mathbf{update}(x, c_t);$
$PW7 : \left\{ \begin{array}{l} odd(\mathsf{loc}_t) \land \mathsf{loc}_t = \vec{\mathsf{glb}} \land (\forall y \in \mathbf{dom}(log).\ x \neq y \implies [y]_t^{\mathsf{A}} = \{\vec{y}\}) \land \mathsf{writer} = t \\ \land\mathsf{hasWritten}_t \land (\forall y.\ [y]_t = \{\vec{y}\}) \end{array} \right\}$
  $W7 :$ store $x\ v$;
$PW8 : \left\{ \begin{array}{l} odd(\mathsf{loc}_t) \land \mathsf{loc}_t = \vec{\mathsf{glb}} \land (\forall y \in \mathbf{dom}(log).\ x \neq y \implies [y]_t^{\mathsf{A}} = \{\vec{y}\}) \land \mathsf{writer} = t \\ \land\mathsf{hasWritten}_t \land (\forall y.\ [y]_t = \{\vec{y}\}) \end{array} \right\}$
  $W8 :$ $\mathbf{flush}_{\text{opt}}\ x;$
$PWr : \{\mathsf{ready}_t\}$
  $Wr :$ return $ok$;   $\{\mathsf{ready}_t\}$
```

Figure 6.5: TMWrite annotation

```
$\texttt{TMCommit}$
$PCp : \{\mathsf{ready}_t\}$
  $Cp :$ if $odd(\mathsf{loc}_t)$ then
$PC1 : \left\{ \begin{array}{l} \mathsf{hasWritten}_t \land \mathsf{writer} = t \land (\forall y.\ [y]_t = \{\vec{y}\}) \land \mathsf{loc}_t = \vec{\mathsf{glb}} \\ \land(\forall y \in \mathbf{dom}(log).[y]_t^{\mathsf{A}} = \{\vec{y}\}) \end{array} \right\}$
  $C1 :$ sfence;
$PC2 : \left\{ \begin{array}{l} \mathsf{hasWritten}_t \land \mathsf{writer} = t \land (\forall y.\ [y]_t = \{\vec{y}\}) \land \mathsf{loc}_t = \vec{\mathsf{glb}} \\ \land(\forall y \in \mathbf{dom}(log).[y]^{\mathsf{P}} = \{\vec{y}\}) \end{array} \right\}$
  $C2 :$   $log.\mathbf{empty}();$
$PC3 : \{\mathsf{hasWritten}_t \land \mathsf{writer} = t \land (\forall y.\ [y]_t = \{\vec{y}\}) \land \mathsf{loc}_t = \vec{\mathsf{glb}}\}$
  $C3 :$   $\langle \mathbf{store}\ \mathsf{glb}\ (\mathsf{loc}_t + 1),$
           $\mathsf{writer} := None \rangle \{\mathsf{true}\}$
  $Cr :$ return $commit$;   $\{\mathsf{true}\}$
```

Figure 6.6: TMCommit annotation

```
TMRecover
$PRec1 : \left\{\begin{array}{l} \mathsf{writer} = \bot \wedge (\forall y.\ [y]_{syst} = \{\vec{y}\}) \\ \wedge(\forall ts \in \mathbf{dom}(M).ts > 0 \implies M[ts].\mathsf{loc} \neq \mathsf{glb}) \end{array}\right\}$
  $Rec1$ :  while $\neg log.\mathbf{isEmpty}()$
$PRec2 : \left\{\begin{array}{l} \mathbf{dom}(log) \neq \{\} \wedge \mathsf{writer} = \bot \wedge (\forall y.\ [y]_{syst} = \{\vec{y}\}) \\ \wedge(\forall ts \in \mathbf{dom}(M).ts > 0 \implies M[ts].\mathsf{loc} \neq \mathsf{glb}) \end{array}\right\}$
  $Rec2$ :    $c_{syst} := log.\mathbf{getKey}();$
$PRec3 : \left\{\begin{array}{l} c_{syst} \in \mathbf{dom}(log) \wedge \mathsf{writer} = \bot \wedge (\forall y.\ [y]_{syst} = \{\vec{y}\}) \\ \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \implies M[ts].\mathsf{loc} \neq \mathsf{glb}) \end{array}\right\}$
  $Rec3$ :    store $c_{syst}$ $log.\mathbf{getVal}(c_{syst});$
$PRec4 : \left\{\begin{array}{l} c_{syst} \in \mathbf{dom}(log) \wedge \mathsf{writer} = \bot \wedge (\forall y.\ [y]_{syst} = \{\vec{y}\}) \\ \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \implies M[ts].\mathsf{loc} \neq \mathsf{glb}) \end{array}\right\}$
  $Rec4$ :    $\mathbf{flush}_{\mathrm{opt}}\ c_{syst};$
$PRec5 : \left\{\begin{array}{l} c_{syst} \in \mathbf{dom}(log) \wedge \mathsf{writer} = \bot \wedge (\forall y.\ [y]_{syst} = \{\vec{y}\}) \\ \wedge\ [c_{syst}]^{\mathsf{A}}_{syst} = \{\vec{c_{syst}}\} \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \implies M[ts].\mathsf{loc} \neq \mathsf{glb}) \end{array}\right\}$
  $Rec5$ :    sfence;
$PRec6 : \left\{\begin{array}{l} c_{syst} \in \mathbf{dom}(log) \wedge \mathsf{writer} = \bot \wedge (\forall y.\ [y]_{syst} = \{\vec{y}\}) \\ \wedge\ [c_{syst}]^{\mathsf{P}} = \{\vec{c_{syst}}\} \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \implies M[ts].\mathsf{loc} \neq \mathsf{glb}) \end{array}\right\}$
  $Rec6$ :    $log.\mathbf{update}(c_{syst}, \bot);$
$PRec7 : \left\{\begin{array}{l} \mathbf{dom}(log) = \{\} \wedge \mathsf{writer} = \bot \wedge (\forall y.\ [y]_{syst} = \{\vec{y}\}) \\ \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \implies M[ts].\mathsf{loc} \neq \mathsf{glb}) \end{array}\right\}$
  $Rec7$ : $c_{syst}$ := load glb;
$PRec8 : \left\{\begin{array}{l} c_{syst} = M[0](\mathsf{glb}) \wedge \mathbf{dom}(log) = \{\} \wedge \mathsf{writer} = \bot \\ \wedge(\forall y.\ [y]_{syst} = \{\vec{y}\}) \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \implies M[ts].\mathsf{loc} \neq \mathsf{glb}) \end{array}\right\}$
  $Rec8$ : if $even(c_{syst})$ then
$PRec9 : \left\{\begin{array}{l} even(c_{syst}) \wedge \mathbf{dom}(log) = \{\} \wedge \mathsf{writer} = \bot \wedge c_{syst} = M[0](\mathsf{glb}) \\ \wedge(\forall y.\ [y]_{syst} = \{\vec{y}\}) \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \implies M[ts].\mathsf{loc} \neq \mathsf{glb}) \end{array}\right\}$
  $Rec9$ :    $\langle$store glb $c_{syst} + 2$,
              recGlb $:= c_{syst} + 2\rangle \{pc_{syst} = Rec_{complete}\}$
$PRec9 : \left\{\begin{array}{l} odd(c_{syst}) \wedge \mathbf{dom}(log) = \{\} \wedge \mathsf{writer} = \bot \wedge c_{syst} = M[0](\mathsf{glb}) \\ \wedge(\forall y.\ [y]_{syst} = \{\vec{y}\}) \wedge\ (\forall ts \in \mathbf{dom}(M).ts > 0 \implies M[ts].\mathsf{loc} \neq \mathsf{glb}) \end{array}\right\}$
 $Rec10$ : else $\langle$store glb $(c_{syst} + 1)$,
                recGlb $:= c_{syst} + 1\rangle \{pc_{syst} = Rec_{complete}\}$
```

Figure 6.7: TMRecover annotation

6.5 Durable Opacity of dTML$_{\mathbf{Px86}}$ ()

To show that dTML$_{\text{Px86}}$ is durably opaque, we establish a forward simulation between the dTML$_{\text{Px86}}$ transition system and the dTMS2 specification (see §3.4.3). The dTMS2 specification is shown to imply durable opacity (Theorem 3.4.1).

It is well known that a forward simulation is a sound proof technique for refinement. As in proofs of linearizability [52], refinement must guarantee trace inclusion, i.e. that every externally observable behavior of the concrete system (i.e. dTML$_{\text{Px86}}$) is an externally observable behavior of the abstract system (i.e. dTMS2). Here, the external steps (i.e., transitions) correspond to invocations and responses of transactional operations as well as the system crashes. We now give a more general definition of forward simulation than the one presented in Chapter 3 that does not apply only to input/output automata but to any transition system.

Definition 6.5.1 (Forward simulation)**.** For an abstract system A and a concrete system C, a relation R between the states of A and C is a *forward simulation* iff each of the following holds.

Initialisation. For any initial state cs_0 of C, there exists an initial state as_0 of A such that $R(as_0, cs_0)$.

External step correspondence. For any external transition from cs to cs' in C and any state as of A such that $R(as, cs)$, there exists an external transition from as to as' such that $R(as, cs)$.

Internal step correspondence. For any internal transition from cs to cs' of C and any state as of A such that $R(as, cs)$, either:

- $R(as, cs')$, or (stuttering step)
- there is an internal transition of A from as to as' such that $R(as', cs')$. (non-stuttering step)

We start with identifying the *linearization points* of its dTML$_{\text{Px86}}$ operation. Operation `TMBegin` linearizes at $B1$ provided loc_t is even. For transactions that have not performed any `TMRead` or `TMWrite`, the linearization point of `TMRead` is a successful **CAS** at $R2$. For any other type of transaction, `TMRead` linearizes at $R5$ provided the value read from glb is loc_t. Operation `TMWrite` linearizes when the memory is updated at $W7$. Operation `TMCommit` has two linearization points depending on whether the transaction has successfully executed a `TMWrite` operation. For a writing transaction (i.e., when loc_t is odd), `TMCommit` linearizes at $C2$. Otherwise, `TMCommit` linearizes at Cp.

The forward simulation relation (R) for our dTMS2-dTML$_{\text{Px86}}$ refinement obtains the same form as of the forward simulation relation presented in Chapter 3. Specifically, it splits into two relations: a global relation *globalR* and a transactional relation *txnR*. The global relation describes how the shared states of the two transition systems are related, while the transactional relation specifies the relation between the state of each transaction in the concrete and abstract transition system. The simulation itself is

$R(cs, as) = globalR(cs, as) \wedge \forall t \in \textsc{Tid}.\ txnR(cs, as, t)$

Our simulation relation assumes the following auxiliary definitions, where cs is the concrete state and as is the abstract state. As in Chapter 3, $\mathit{intHalf}(n) \triangleq \lfloor \frac{n}{2} \rfloor$, returns the integer part of n divided by 2.

$$
\begin{aligned}
\mathit{writes}(cs, as) &\triangleq \textbf{if } cs.\mathsf{writer} = t \wedge pc_t \neq C3 \textbf{ then } as.\mathsf{wrSet}_t \textbf{ else } \emptyset \\
\mathit{logicalGlb}(cs) &\triangleq \textbf{if } cs.\mathsf{writer} = t \wedge pc_t = C3 \\
&\qquad \textbf{then } cs.\overrightarrow{\mathsf{glb}} - cs.\mathsf{recGlb} + 1 \textbf{ else } cs.\overrightarrow{\mathsf{glb}} - cs.\mathsf{recGlb} \\
\mathit{wrCount}(cs) &\triangleq \mathit{intHalf}(\mathit{logicalGlb}(cs)) \\
\mathit{inFlight}(t, cs) &\triangleq t \in \textsc{Tid} \wedge pc_t \notin \{\mathit{NotStarted}, \mathit{Aborted}, \mathit{Committed}\}
\end{aligned}
$$

Function *writes* returns the abstract wrSet_t of a writing transaction. Note that the abstract wrSet_t is empty after the writing transaction has cleared its log and hence linearized `TMCommit` at $C2$. Function *logicalGlb* is used to determine the logical value of glb (since initialization or the last recovery) and as in the refinement proof of $\mathrm{dTML}_{\mathsf{SC}}$ compensates for the fact that a committing writing transaction has linearized by not yet incremented glb at $C3$. Function $\mathit{wrCount}(cs)$ returns the number of committed writing transactions in the concrete state, taking into account the fact that each writing transaction updates glb twice. Finally, *inFlight* is used to determine whether the given transaction t in state cs is *live* (has been started but has not been committed or aborted).

6.5.1 Global Relation of the $\mathbf{dTML_{Px86}}$-dTMS2 Simulation Relation

The global relation *globalR* comprises conditions (6.1)-(6.4) below. The definition relies on $\mathsf{LE}(t, x)$ which returns the timestamp of the last write to x before timestamp t.

$$
\begin{aligned}
&\mathit{globalR}(cs, as) = \\
&\quad \neg\mathsf{Recovering} \Rightarrow (\forall x.\ x \neq \mathsf{glb} \Rightarrow \vec{x} = (\mathit{last}(as.L) \oplus \mathit{writes}(cs, as))(x) \wedge && (6.1)\\
&\quad \neg\mathsf{Recovering} \Rightarrow (\mathit{wrCount}(cs) = |as.L| - 1) \wedge && (6.2)\\
&\quad \forall x.\ x \neq \mathsf{glb} \Rightarrow \mathit{last}(as.L)(x) = \textbf{if } x \notin \textbf{dom}(cs.\mathit{log})\textbf{then } \vec{x} \textbf{ else } (cs.\mathit{log})(x) \wedge && (6.3)\\
&\quad \forall i.\ \forall v.\ cs.M[i] \equiv \langle \mathsf{glb} := v \rangle \Rightarrow \forall x.\ \forall w.\ x \neq \mathsf{glb} \wedge cs.M[\mathsf{LE}(i, x)] \equiv \langle x := w \rangle \Rightarrow && (6.4)\\
&\qquad\qquad as.L[\mathit{intHalf}(v - cs.\mathsf{recGlb})](x) = w
\end{aligned}
$$

The first two conditions assume that the recovery process is not in progress.

Condition (6.1) states that, for each location x different from glb, the last written value for x in $\mathrm{dTML}_{\mathrm{Px86}}$ is the value of x in the last memory snapshot of dTMS2 overwritten by the write set of an in-flight writing transaction (if there is any).

Condition (6.2) states that the number of memory snapshots in dTMS2 memory since initialization or the last crash is equal to $\mathit{wrCount}(cs)$.

Condition (6.3) states that, for each location x different from glb, the value of x in the last memory snapshot of dTMS2 is the last written value for x in $\mathrm{dTML}_{\mathrm{Px86}}$ whenever x is not in *log* and is the corresponding value in *log*, otherwise.

Condition (6.4) states that whenever dTML$_{\mathrm{Px86}}$'s memory index i contains a write to glb with value v, for any location x different from glb, the value of the last write to x before index i is precisely the value of x in the abstract memory snapshot indexed by $\mathit{intHalf}(v - cs.\mathsf{recGlb})$.

The *globalR* relation and program annotations work together to match abstract and concrete values during *external* reads. For example, suppose a read-only transaction $\mathtt{TMRead}(x)$ in dTML$_{\mathrm{Px86}}$ reads x for the first time, returning a value v. The first step of the refinement proof requires identifying the timestamp of the dTML$_{\mathsf{SC}}$ memory with location x and value v. The remaining step is to check that the corresponding element satisfies the validity constraint, *validIdx*, imposed by dTMS2 to ensure that dTMS2 can indeed, take the corresponding abstract step.

6.5.2 Transactional Relation of the dTML$_{\mathrm{Px86}}$-dTMS2 Simulation Relation

The transaction relation (*txnR*) part of the forward simulation comprises the conditions (6.5)-(6.10) below as well as the condition that we describe at the end of the section

$$\forall t.\ \mathit{inFlight}(t, cs) \wedge \neg cs.\mathsf{hasWritten}_t \wedge \neg cs.\mathsf{hasRead}_t \Rightarrow as.\mathsf{rdSet}_t = \emptyset \tag{6.5}$$

$$\forall t.\ \mathit{inFlight}(t, cs) \wedge cs.\mathsf{hasRead}_t \Rightarrow as.\mathsf{rdSet}_t \neq \emptyset \tag{6.6}$$

$$\forall t.\ \mathit{inFlight}(t, cs) \wedge (\neg cs.\mathsf{hasWritten}_t \vee \mathit{even}(cs.\mathsf{loc}_t)) \Rightarrow as.\mathsf{wrSet}_t = \emptyset \tag{6.7}$$

$$\forall t.\ \mathit{inFlight}(t, cs) \wedge \mathit{odd}(cs.\mathsf{loc}_t) \wedge cs.pc_t \notin \{Bp, B1, W4 - W7\} \Rightarrow as.\mathsf{wrSet}_t \neq \emptyset \tag{6.8}$$

$$\forall t.\ \mathit{inFlight}(t, cs) \wedge cs.\mathsf{writer} = t \wedge cs.pc_t \notin \{W4 - W7\} \Rightarrow as.\mathsf{wrSet}_t \neq \emptyset \tag{6.9}$$

$$\forall t. \forall x \in \mathbf{dom}(as.\mathsf{wrSet}_t).\ cs.\mathsf{writer} = t \Rightarrow (as.\mathsf{wrSet}_t)(x) = cs.\vec{x} \tag{6.10}$$

The first five conditions relate the dTML$_{\mathrm{Px86}}$ state of an *inFlight* transaction t with the wrSet_t and rdSet_t variables of the corresponding dTMS2 state. By (6.5), if $cs.\mathsf{hasRead}_t$ and $cs.\mathsf{hasWritten}_t$ are false then $as.\mathsf{rdSet}_t$ is empty. By (6.6) if $cs.\mathsf{hasRead}_t$ is *true* then $as.\mathsf{rdSet}_t$ is not empty. By (6.7) if $cs.\mathsf{hasWritten}_t$ is $false$ or the value of $cs.\mathsf{loc}_t$ is *even* then the $as.\mathsf{wrSet}_t$ is empty. By (6.8) if the value of $cs.\mathsf{loc}_t$ is *odd* and the program counter of t is not Bp or $B1$, where loc_t has not yet obtained a valid starting value and is also not in $W4 - W7$, where t has not performed yet any write even though loc_t is *odd*, then $cs.\mathsf{wrSet}_t$ can not be empty. Finally by (6.9), if t is a writing transaction ($as.\mathsf{writer} = t$) and its program counter is not in $W4-W7$, then $as.\mathsf{wrSet}_t$ is also not empty. Determining when the write set of a transaction is empty is particularly important, as dTMS2 imposes different ordering constraints to the read-only transactions from the writing transactions in `TMCommit`. Condition (6.10) resolves *internal* reads by specifying that the dTMS2 write set ($as.\mathsf{wrSet}_t$) for a transaction t should include the most recent value written by dTML$_{\mathrm{Px86}}$ for each location in the write set domain.

We now provide a description of the final condition of *txnR*. We summarise its main purposes above.

1) Maps abstract with concrete pc values. Firstly, as with the correctness proof of dTML$_{\mathsf{SC}}$, the final condition maps the dTML$_{\mathrm{Px86}}$ program counter values to their dTMS2 counterparts. The mapping is depicted in Table 6.1. The steps in which the concrete *pc* values are mapped to the same abstract *pc* value in the pre and post-state are stuttering steps, while the steps in which the concrete *pc* values are mapped to different abstract *pc* values in the pre and post state are non-stuttering steps.

$cs.pc:$	$NotStarted$	$Aborted$	$Ready$	$Committed$
$as.pc:$	$NotStarted$	$Aborted$	$Ready$	$Committed$
$cs.pc:$	$Bp, B1$	Br	$Rp - R2$ $R3 \wedge \neg cs.\mathsf{hasRead}_t$, $R4, R5$	$R3 \wedge cs.\mathsf{hasRead}_t$, Rr
$as.pc:$	$BeginPending$	$BeginResponding$	$ReadPending(x)$	$ReadResponding(v)$
$cs.pc:$	$Wp - W7$	$W8, Wr$	$Cp, C1, C2$	$C3, Cr$
$as.pc:$	$WritePending(x, v)$	$WriteResponding$	$CommitPending$	$CommitResponding$

Table 6.1: Mapping of $\text{dTML}_{\text{Px86}}$ to dTMS2 pc values.

2) Maps the returned read values to their abstract counterparts. Secondly, the final condition ensures that the value returned by a $\text{dTML}_{\text{Px86}}$'s successful read on x ($\mathtt{TMRead}(x)$) is the same as the value returned in the corresponding $\mathsf{non\text{-}stuttering}$ step of dTMS2. We give an overview of the proof for identifying the corresponding values above.

Case: Read-only transaction. The first read ($\mathtt{TMRead}(x)$) of a read-only transaction t succeeds if $cs.\mathsf{loc}_t$ obtains the maximum value of $cs.\mathsf{glb}$. Otherwise, the **CAS** instruction at $R2$ fails. Based on the precondition of $R2$ ($P2$) (see Fig. 6.4), if the **CAS** instruction succeeds, we can deduce that $cs.\vec{\mathsf{glb}} = cs.\mathsf{loc}_t$ and $cs.\mathsf{loc}_t$ is even. Additionally, we can infer that there is no message with a timestamp greater than or equal to the timestamp corresponding to the last write of $cs.\mathsf{glb}$ with a location different from glb. This is because, according to Property 3, if such a write existed, the value of the last write of glb would be odd. Therefore, the timestamp of the message read for x precedes the timestamp of the message of the last write of $cs.\mathsf{glb}$ and must have the form $\mathsf{LE}(i, x)$, where $cs.M[i] \equiv \langle cs.\mathsf{glb} := cs.\mathsf{loc}_t \rangle$. By instantiating Condition (6.4), we can infer that the value read for x corresponds to the abstract value $as.L[intHalf(cs.\mathsf{loc}_t - cs.\mathsf{recGlb})](x)$.

For any subsequent read operation ($\mathtt{TMRead}(x)$) performed by a read-only transaction t, we can derive the index of the memory snapshot of dTMS2 that contains the returned write directly by the $\mathtt{TMRead}$ program annotation. Specifically, the $\mathtt{TMRead}$ program annotation (assertion $P5$ in Fig. 6.4) imposes that the only value that can be successfully returned by $\text{dTML}_{\text{Px86}}$ corresponds to the concrete memory message with timestamp $\mathsf{LE}_{\mathsf{coh}}(cs.\mathsf{glb}, t, x)$. By expanding the definition of $\mathsf{LE}_{\mathsf{coh}}$, we obtain $\mathsf{LE}_{\mathsf{coh}}(cs.\mathsf{glb}, t, x) = M[\mathsf{LE}(\mathsf{coh}_t\ cs.\mathsf{glb}, x)]$. Given this, and by using condition (6.4), we can determine that the index of the memory snapshot of dTMS2 containing this write is $as.L[intHalf(cs.\mathsf{loc}_t - cs.\mathsf{recGlb})]$.

Case: Writing transaction. According to the $\mathtt{TMRead}$ program annotation (assertion $P5$ in Fig. 6.4), a read operation on x ($\mathtt{TMRead}(x)$) of a writing transaction t can only return the last value written on x ($cs.\vec{x}$). In case the read is *external* by utilizing condition (6.1) we can deduce that the corresponding abstract value is equal to $last(as.L)(x)$. In case the read is *internal* by condition (6.10) the corresponding abstract value is equal to $(as.\mathsf{wrSet}_t)(x)$.

3) Guarantees that the ordering constraints of dTMS2 are met. Thirdly, the final condition incorporates Conditions (6.11) and (6.12), which are sufficient for demonstrating that the ordering (validity) constraints of dTMS2 are met. These conditions guarantee that for any read-only transaction t, when $\mathtt{TMRead}$ and $\mathtt{TMCommit}$ linearize, there exists a timestamp (i.e., $n = intHalf(cs.\mathsf{loc}_t - cs.\mathsf{recGlb})$) such that $validIdx(t, n)$

holds. Furthermore, they ensure that for any writing transaction t that performs an external read, when TMRead linearizes, $\mathit{validIdx}(t, \mathit{intHalf}(cs.\mathsf{loc}_t - cs.\mathsf{recGlb}))$ again holds. Lastly, Condition (6.11) helps in determining that for any writing transaction t, when TMCommit linearizes, $as.\mathsf{rdSet}_t \subseteq \mathrm{last}(as.L)$.

$$\begin{aligned}
&\forall t.(\mathit{inFlight}(t, cs) \wedge \\
&\quad (cs.\mathsf{pc}_t \in \{\mathit{Ready}, Wp, Wr, Rp, Rr, Cp\} \wedge \mathsf{loc}_t = \vec{\mathsf{glb}}) \\
&\quad \vee cs.\mathsf{pc}_t \in \{W1 - W8, R1 - R5, C1 - C2\}) \\
&\quad \Rightarrow as.\mathsf{rdSet}_t \subseteq as.M[\mathit{intHalf}(cs.\mathsf{loc}_t - \mathsf{recGlb})]
\end{aligned} \tag{6.11}$$

$$\begin{aligned}
&\forall t.(\mathit{inFlight}(t, cs) \wedge \\
&\quad (cs.\mathsf{pc}_t \in \{\mathit{Ready}, Wp, Wr, Rp, Rr, Cp\} \wedge \mathsf{loc}_t = \vec{\mathsf{glb}}) \\
&\quad \vee cs.\mathsf{pc}_t \in \{W1 - W8, R1 - R5, C1 - C2\}) \\
&\quad \Rightarrow as.\mathsf{beginIdx}_t \leq \mathit{intHalf}(cs.\mathsf{loc}_t - \mathsf{recGlb})
\end{aligned} \tag{6.12}$$

4) Ensures that the abstract memory is consistent with the concrete memory after the TMRecover process of dTML$_{Px86}$ takes place. Lastly, the final condition ensures that after a TMRecover operation completes, the dTML$_{\mathrm{Px86}}$ memory list is consistent with the dTMS2 memory. Specifically, it ensures that immediately after a crash, the length of the dTMS2 memory list is 1, the transaction that executes the TMRecover operation is *syst* and the value of each location x that is read and cleared from the *cs.log* in each recovery loop is equal to the corresponding value of x the memory snapshot of dTMS2.

6.5.3 Step Correspondence Between dTML$_{\mathbf{Px86}}$ and dTMS2

We now describe precisely the step correspondence between dTML$_{\mathrm{Px86}}$ and dTMS2. For this we use a step corresponding function *sc*, of the form $sc(cs, t, \alpha) = \beta$, where cs is a concrete state, t is a transaction identifier, α is an internal transition of dTML$_{\mathrm{Px86}}$ and β is the internal transition of dTMS2 that corresponds to α in case α indicates a non-stuttering step. In case α indicates a stuttering step *sc* returns $\perp$.

- A **begin operation** takes effect at $B1$. The *pc* mapping provided by *txnR* (Fig. 6.1) guarantees that the corresponding abstract transition is $\mathtt{doBegin}_t$. Thus if $\alpha = B1$ then $sc(cs, t, \alpha) = \mathtt{doBegin}_t$.

- A **read operation** takes effect at $R2$ for the first read of a read-only transaction, provided that the corresponding **CAS** instruction succeeds, and at $R5$ for a writing transaction or subsequent reads of a read-only transaction provided that $cs.\mathsf{loc}_t$ is equal to $cs.\mathsf{glb}$. The concrete to abstract mapping of program counter values (Fig. 6.1) guarantees that the read address coincides for both TMS2 and dTML$_{\mathrm{Px86}}$ and that in both cases, the corresponding abstract transition is $\mathsf{doRead}_t(x)$.

  In all cases, excluding internal reads, by the final condition, we have that the successfully loaded value corresponds to the $as.L[\mathit{intHalf}(cs.\mathsf{loc}_t - cs.\mathsf{recGlb})](x)$ value of the abstract memory. Combining, the TMRead annotation (see Fig. 6.4), Condition (6.2) and Condition (6.12) we can deduce that at both $R2$ and $R5$

$as.\mathsf{beginIdx}_t \leq \mathit{intHalf}(cs.\mathsf{loc}_t - cs.\mathsf{recGlb}) < |as.L|$. Furthermore, by Condition (6.11) we have that $as.\mathsf{rdSet}_t \subseteq as.L[\mathit{intHalf}(cs.\mathsf{loc}_t - cs.\mathsf{recGlb})]$.
Therefore $\mathit{validIdx}(t, \mathit{intHalf}\, cs.\mathsf{loc}_t - cs.\mathsf{recGlb})$ holds. Having this, if $\alpha = R2$ (resp.$\alpha = R5$) and the corresponding **CAS** instruction succeeds (resp. $cs.\mathsf{glb} = \mathsf{loc}_t$) then $sc(cs, t, \alpha) = \mathtt{doRead}_t(x) \wedge \mathit{validIdx}(t, \mathit{intHalf}\, cs.\mathsf{loc}_t - cs.\mathsf{recGlb})$.

- A **write operation** takes effect when a transaction t executes $W7$. The pc mapping provided by $txnR$ (Fig. 6.1) guarantees that the the corresponding abstract action is $\mathtt{doWrite}_t(x, v)$. Thus if $\alpha = W7$ then $sc(cs, t, \alpha) = \mathtt{doWrite}_t(x, v)$. As before the pc mapping provided by $txnR$ ensures that $as.pc_t$ obtains the correct value.

- A **commit operation** is executed at Cp for a read-only transaction and at $C2$ for a writing transaction. In both cases ($\alpha = Cp$ or $\alpha = C2$), we have $sc(cs, t, \alpha) = \mathtt{doCommit}_t$. Moreover, similar to the case of a read operation, we can deduce that $\mathit{validIdx}(t, \mathit{intHalf}\, cs.\mathsf{loc}_t - cs.\mathsf{recGlb})$ holds in both situations.

  Conditions (6.5)-(6.9) are used for determining if t is a read-only transaction ($as.\mathsf{wrSet}_t = \emptyset$). If so, the precondition of dTMS2 requires that $\exists n.\mathit{validIdx}(t, n)$

  For writing transactions ($as.\mathsf{wrSet}_t \neq \emptyset$) the precondition of dTMS2 requires that $\mathsf{rdSet}_t \subseteq \mathit{last}(as.L)$. Using the final condition of $txnR$ in combination with Condition (6.2) and the TMCommit annotation at $PC2$ (see Fig. 6.6) we can obtain that $as.L[\mathit{intHalf}(cs.\overrightarrow{\mathsf{glb}} - cs.\mathsf{recGlb})] = \mathit{last}(as.L)$ and thus $\mathsf{rdSet}_t \subseteq \mathit{last}(as.L)$.

  Finally, the $txnR$ program-counter mapping ensures that $as.\mathsf{pc}_t$ has the correct value as usual.

  In all other cases $sc(cs, t, \alpha) = \bot$.

Theorem 6.5.1 (). *SimR is a forward simulation is a between dTMS2 and dTML$_{Px86}$.*

6.6 Mechanization Effort

The refinement proof has been fully mechanized in Isabelle-HOL. This mechanization builds on the PIEROGI$_{\text{full}}$ framework [24]. The first step focused on demonstrating the crash invariant of dTML$_{\text{Px86}}$ (§6.4.1). The second step concerned showing the validity of the proof outline for dTML$_{\text{Px86}}$ (Theorem 6.4.1). These steps required approximately 2.5 months of full-time work.

Finally, the third step involved proving that the dTML$_{\text{Px86}}$ implementation refines the dTMS2 specification (Theorem 6.5.1). Specifically, we established the simulation relation for each step of the dTML$_{\text{Px86}}$ transition system, resulting in a total of 47 sub-proofs. This last step required approximately 2 months.

As with the development of PIEROGI, throughout this work we use the built-in sledgehammer tool to enable finding relevant proof rules needed to discharge proof obligations. The core development of PIEROGI$_{\text{full}}$, including semantics, view-based expressions, and the soundness of proof rules, consists of approximately 10,000 lines of Isabelle/HOL code. With this foundation in place, the proof of the crash invariant, and the validity of the proof outline for dTML$_{\text{Px86}}$ encompass approximately 20,000 lines of Isabelle/HOL code, including the dTML$_{\text{Px86}}$ encoding and annotations. The refinement proof consists of approximately 10.000 lines.

6.7 Related Work

It is already known that TMS1 [50], which is a weaker condition than opacity [86] is sufficient for contextual refinement under SC [14]; However, the connection between durable opacity and contextual refinement remains unexplored. This becomes particularly relevant in the case of persistent transactional memory implementations like $\mathrm{dTML}_{\mathrm{Px86}}$, which are primarily intended for use as libraries. Two key concerns arise in this context. Firstly, it is important to determine the client-side guarantees that are necessary under weak persistent memory models and to assess whether dTMS2 alone is sufficient to provide these guarantees. The second concern revolves around establishing verification techniques for validating client programs under Px86 (or other relaxed persistency models).

Relevant work in the context of C11 has been conducted by Dalvandi *et al.* [42]. The paper demonstrates that TMS2 is insufficient for providing any client guarantees under the relaxed memory model of C11. To this end, a more adequate specification has been proposed, which constitutes an adaptation of TMS2. Furthermore, a logic has been developed for verifying client programs. In the context of persistent memory, Khyzha *et al.* [91] have introduced a correctness criterion that ensures contextual refinement. However, this criterion overlooks the complexities arising from weak persistency as it assumes the persistent sequential consistency model [90]. A more recent work for verifying persistent transactional libraries has been developed by Stefanesco *et al.* [136]. Their work introduces a declarative framework that offers flexibility for specifying persistent libraries and facilitates modular verification. The framework addresses implementations under weak persistency models such as Px86. However, it is not mechanized. We believe an operational approach may also be beneficial for verifying persistent TM implementations.

Chapter 7

Conclusions

Overall, this thesis has addressed several key aspects related to the formalization and verification of persistent transactional memory algorithms, providing insights into the challenges and solutions in defining and proving correctness guarantees in a persistent memory setting. In this thesis, we primarily explore two research questions. The first question delves into the concept of correctness in persistent transactional memory algorithms: What constitutes correctness in this context, and how can it be precisely defined and formalized? We start exploring this research topic by reviewing several transactional memory correctness criteria targeting volatile transactional memory algorithms(§2.3). We later present a collection of correctness criteria for durable concurrent objects (§2.4). Having gained ideas from both research areas, we suggest and formalize *durable opacity* [23] (§3.1), an adaptation of opacity that accounts for durability and system crashes.

The second question concerns the verification of persistent transactional memory algorithms: What methods can be used to effectively verify these algorithms, and to what degree are the existing verification techniques, originally designed for volatile transactional memory algorithms, applicable in this new context? We approach the verification problem gradually, initially building our proofs assuming a simplified memory model, persistent SC, which does not fully capture the complexities of realistic persistent memory architectures. However, it provided us with the opportunity to gain an initial insight into the verification challenges that arise from persistency. Subsequently, we integrate these proofs into a more complex and realistic model, Px86.

In our initial effort to verify a transactional memory algorithm, we attempt to show that an adaptation of the TML algorithm to persistent SC, $\text{dTML}_{\mathsf{SC}}$, adheres to durable opacity [38] (Chapter 3). The proposed verification method comprises two steps. The initial step involves developing a program invariant for $\text{dTML}_{\mathsf{SC}}$ and showing that it is locally correct and interference-free. This is achieved by applying the Owicki–Gries method [114] in its classic form. The subsequent step comprises using *forward simulation* [103, 109] and the $\text{dTML}_{\mathsf{SC}}$ program invariant to establish a refinement of $\text{dTML}_{\mathsf{SC}}$ with respect to an abstract operational specification that implies durable opacity (called dTMS2 [23]).

Later on, we try to verify a transactional memory algorithm assuming the more complicated Px86 model. We start this process by establishing a program logic that consists of assertions that are able to describe the program behaviors induced by the asynchronous nature of the Px86 weak memory model as well as the state of persistent memory. The proposed logic [24] (Chapter 4) leverages two works: Px86$_{\text{view}}$ [32], a view-based operational semantics of x86 persistency and the C11 Owicki–Gries logic [39, 42, 48] to reason about view-based operational semantics, which we adapt to Px86$_{\text{view}}$. We initially use our logic, PIEROGI, to verify several litmus tests (Chapter 5). During this process, we have realized that the local correctness and interference freedom validity requirements of the Owicki–Gries method are insufficient for asserting the validity of persistent memory states. To this end, we enhance the Owicki–Gries validity proof outline with an extra requirement, named *persistence*, which imposes showing that an invariant that describes the state of the persistent memory (*crash invariant*) holds until the program's initial crash (Def. 4.2.1).

In our last work, we attempt to apply our logic to verify an adaptation of TML to Px86, dTML$_{\text{Px86}}$ (Chapter 6). We have then realized the importance of expressing the crash and recovery process in an operational way and enabling verification beyond the initial crash of a program. As a result, we have updated the definition of *crash invariant* to represent a set of properties that hold for all the program transitions, including the crash transition and the algorithm's recovery process (Def. 4.2.2). The above enables us to enhance PIEROGI with assertions that describe memory patterns that span through pre- and post-crash program states. We name the enhanced version of our logic PIEROGI$_{\text{full}}$(Chapter 4).

The verification process of dTML$_{\text{Px86}}$ consists of the following steps. We first develop a program annotation and crash invariant for dTML$_{\text{Px86}}$ using PIEROGI$_{\text{full}}$ assertions. Subsequently, we show that the Owicki–Gries validity proof outline with the updated requirement of *peristence* holds. Afterwards, we use the above and the forward simulation technique to establish a refinement of dTML$_{\text{Px86}}$ with respect to dTMS2.

All the proofs outlined above have been mechanized in Isabelle/HOL.

In summary, this thesis demonstrates that established verification methods and concepts for volatile memory algorithms, including view-based semantics, the Owicki–Gries method, and the forward simulation technique for showing refinement, can be used for verifying persistent memory algorithms. These methods/concepts can be applied either directly or require minimal modifications, and are proved to be equally effective in this domain.

7.1 Discussion

In this section, we discuss the difficulties we have encountered throughout the development as well as the motivation behind our design and verification decisions.

In Chapter 3, our focus has been on the formalization of *durable opacity* and the development of an example algorithm and verification technique. Durable opacity extends opacity to address full-system crash events, similar to how durable linearizability [81] extends the notion of linearizability [73]. While deliberating on the appropriate correctness condition for our persistent transactional memory implementation, we primarily considered two other candidates. The first was persistent serializability as defined in [123], and the second was an adaptation of recoverability [20] for persistent memory. Our version of persistent recoverability revolves around the idea of persisting the writes of a transaction only if a subsequent transaction reads at least one of them. Nevertheless, we opted to design a durable opaque persistent STM implementation primarily because of opacity's strong guarantees. Driven by the potential ramifications of inconsistent reads, which can result in irrecoverable errors like divide-by-zero exceptions or segmentation faults, opacity has emerged as a widely preferred option for TM implementations [7,27,65,105]. Opacity mandates that both aborted and live transactions (even if they eventually abort) must only observe consistent values according to the transactions that have been successfully committed thus far. A possible disadvantage of this criterion is that it can lead to unnecessary aborts.

A relaxed version of durable opacity is *buffered durable opacity*, which can be defined in a similar fashion to buffered durable linearizability [81]. Informally, buffered durable opacity entails persisting only a subhistory p_i of each era between two consecutive crashes (h_i) of a concurrent history h. The persisted subhistory must be formed in a way such that if a transaction t_b is included in p_i and there exists a transaction t_a in h_i that happens before t_b ($t_a \prec_{h_i} t_b$), then t_a is also included in p_i. Taking into account proposed *buffered durable linearizable* implementations [60,110,145], we expect that implementing a *buffered durable opaque* STM will lead to a latency reduction. This expectation is based on the fact that such an implementation would need fewer explicit persist instructions/ persist barriers since it could resume from previous consistent states instead of eagerly persisting transactions. An implementation that enables relaxed durability in the context of persistent hardware transactional memory, allowing a transaction to commit before becoming persistent, is proposed in [61].

Later on, we focus on designing a durably opaque software transactional memory implementation, dTML$_{\mathsf{SC}}$. Our point of departure is the transactional mutex lock (TML) algorithm, presented in [38]. TML is a "lightweight" STM algorithm with minimal storage and instrumentation overheads. As mentioned in [38], TML has the capability to operate with just one word of global metadata, one word of per-thread metadata (assuming the nesting mechanism is disregarded; otherwise, it requires two words), and low instrumentation per access. While TML features a simple design that promotes low latency, it exhibits a drawback in terms of scalability. It scales effectively only in scenarios where critical sections rarely involve write operations. Regardless, we think that TML is a strong candidate for demonstrating persistent STM correctness because, besides its simplicity, it constitutes a *real-world* STM implementation.

We then proceeded to demonstrate a correctness proof technique. We begin with constructing an operational characterization of durable opacity, dTMS2. dTMS2 is derived from TMS2 [50] which has already been proven to imply opacity. Adapting TMS2 to the persistency setting surprisingly only required minimal changes. As a high-level specification that implies durable opacity, dTMS2 has the potential to be reused in data refinement proofs for a broader range of algorithms.

Our verification method constitutes showing that dTML$_{\mathsf{SC}}$ refines dTMS2 by establishing the existence of a *forward simulation* [103, 109]. Our Isabelle/HOL mechanization is based on the existing TML-TMS2 refinement proof mechanization presented in [44]. Although the modifications to the original proof state were relatively minor, numerous lemmas required revising and reproving. Given that an opacity proof has already been developed for an STM implementation, a modularized proof approach might be able to reduce the mechanization effort. An example proof of this kind is given in [22]. The demonstrated method separates the proof of durability for memory accesses from the proof of opacity. Firstly, the persistent STM implementation is encoded exactly as the original STM implementation but with calls to an external (abstract) library for relegating all memory operations (read/writes). Then, a linearizability proof is established between the abstract library (containing a single memory layer) and a concrete library incorporating both volatile and persistent memory. This proof ensures the durability of memory accesses. Combining this proof with the existing opacity proof is adequate to show that the persistent version of the STM implementation is durably opaque.

In Chapter 4, we present an Owicki–Gries program logic for Intel's x86 persistency and consistency model (Px86), PIEROGI. We based our logic on the view-based operational semantics by Cho *et al.* [32]. The expressiveness of these semantics allowed us to define various assertions that efficiently describe the Px86 state, including an assertion (*persistent view*) that describes the values that can be observed for any location of the persistent memory as well as an assertion (*asynchronous view*) that captures directly the effect of Intel's optimised flush instruction. The chapter presents two versions of our logic, $\text{PIEROGI}_{\text{simp}}$ and $\text{PIEROGI}_{\text{full}}$. The key distinction between them lies in the requirement of persistence for the validity of a proof outline. $\text{PIEROGI}_{\text{simp}}$ expresses the persistence requirement as an invariant, which is formed only by persistent view assertions and holds up until the first crash of a program. On the contrary, $\text{PIEROGI}_{\text{full}}$ expresses the persistence requirement as an invariant which is formed by any view-based assertion of $\text{PIEROGI}_{\text{full}}$ and holds thought the execution of a program including its crash and recovery events.

In Chapter 5, we utilize $\text{PIEROGI}_{\text{full}}$ for reasoning about several litmus tests discussed in previous works by Raad *et al.* [119, 122]. In this manner, we illustrate how $\text{PIEROGI}_{\text{full}}$ facilitates the verification and mechanization of programs that were previously unverifiable.

Finally, in Chapter 6, we demonstrate the verification of a durably opaque software transactional memory implementation under Px86. Our TM algorithm, $\text{dTML}_{\text{Px86}}$, is an adaptation of $\text{dTML}_{\mathsf{SC}}$ but with additional synchronization mechanisms to cope with Px86. While developing $\text{dTML}_{\text{Px86}}$, we considered numerous design alternatives. For instance, we were considering moving the **CAS** instruction of line $R2$, to line Bp. In this way, a transaction t could have retried loading the most recent value of glb into loc_t until it succeeds. This would have allowed the transaction to avoid aborting at a later stage. However, while this design may have resulted in fewer aborts, it would likely lead to a considerable increase in overall latency. Another design alternative we were considering is to have a **flush** instruction instead of the $\textbf{flush}_{\text{opt}}$; **sfence** sequence in $Rec4$ and $Rec5$. We expect the **flush** instruction in this case to be equally or more efficient than the current solution.

In the subsequent sections of Chapter 6, we describe the application of the proof technique presented in Chapter 3 for the $\text{dTML}_{\text{Px86}}$ implementation. Establishing correctness under the Px86 memory model posed much greater challenges in comparison to assuming the persistent SC model. A modularised approach, such as the adaptation of the technique demonstrated in [22] to the relaxed setting of Px86, could have potentially reduced the mechanisation effort.

An interesting remark regarding the correctness proofs of $\text{dTML}_{\mathsf{SC}}$ and $\text{dTML}_{\text{Px86}}$ is that the linearization points differ for the two algorithms. This discrepancy remains evident even for TM operations (`TMBegin`, `TMCommit`) that are nearly identical in both implementations. The reason for this lies in the design of the dTMS2 specification, which permits an operation to abort at any point, even after it has linearized. The above allows for a wider range of options when determining the placement of linearization points.

7.2 Future Work

Possible extensions of the work presented in this thesis can be classified into five categories:

1) **Formalization of additional correctness conditions for persistent STMs:** An interesting subject for future work would be to formalize buffered durable opacity and other correctness conditions for persistent software transactional memory, as well as explore their performance implications.e buffered durable opacity and other correctness conditions for persistent software transactional memory, as well as explore their performance implications.

2) **Implementation and benchmarking:** A valuable extension of our work is implementing and benchmarking dTML$_{\mathrm{Px86}}$ and its variations as well as comparing its performance with other persistent STM algorithms (e.g. Onefile [125], RomulusLog [37], Trinity [124], and p-orec [150]).

3) **Extension of the Px86$_{\mathbf{view}}$ logic and application of the outlined methodology to other persistent weak memory models:** The methodology demonstrated for developing a view-based Owicki–Gries logic for Px86 can also be applied to construct a program logic for other weak persistent memory models such as the PArmv8 model. A good starting point for this could be the PArmv8 view-based semantics presented in [32]. Moreover, PIEROGI is capable of being expanded to incorporate reasoning about additional Px86 model features, including non-temporal writes and read/write operations on various memory types such as uncachable and write-through. This can be implemented by extending the Cho *et al.* semantics [32] to cover the additional features and developing in the same manner additional view-based expressions that effectively describe the impact of the additional features to the program state.

4) **Development of alternative mechanizations:** As demonstrated for a view-based Owicki–Gries RC11-RAR logic [39], our mechanization can be integrated to the Nipkow and Nieto's Owicki–Gries framework [111] in the Isabelle/HOL theorem prover. The Nipkow and Nieto's framework, initially developed for programs under the sequentially consistent memory model and later extended to support the RC11-RAR weak memory model [43], provides a WHILE-language for writing concurrent programs. It also allows embedding assertions directly into a program itself. The advantage of this approach is that it allows for the writing of annotated programs using a more familiar pseudocode syntax. Additionally, Nipkow and Nieto's framework supports the automatic generation of all standard Owicki–Gries local-correctness and interference-freedom proof obligations.

5) **Exploring the connection between durable opacity and contextual refinement:** A possible extension of our work is exploring the client-side guarantees that are required under the Px86 model and implementing/verifying dTML$_{\mathrm{Px86}}$ as a library.

Bibliography

[1] "Intel 64 and ia-32 architectures software developer's manual (combined volumes)," https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf/, 2019.

[2] P. A. Abdulla, M. F. Atig, A. Bouajjani, K. N. Kumar, and P. Saivasan, "Deciding reachability under persistent x86-tso," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–32, 2021.

[3] K. Abrahamson, "Modal logic of concurrent nondeterministic programs," in *Semantics of Concurrent Computation: Proceedings of the International Symposium, Evian, France, July 2–4, 1979.* Springer, 2005, pp. 21–33.

[4] M. K. Aguilera and S. Frølund, "Strict linearizability and the power of aborting," *Technical Report HPL-2003-241*, 2003.

[5] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: Definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[6] M. Alshboul, P. Ramrakhyani, W. Wang, J. Tuck, and Y. Solihin, "Bbb: Simplifying persistent programming using battery-backed buffers," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* IEEE, 2021, pp. 111–124.

[7] A. S. Anand, R. Shyamasundar, and S. Peri, "Opacity proof for capr+ algorithm," in *Proceedings of the 17th International Conference on Distributed Computing and Networking*, 2016, pp. 1–4.

[8] K. R. Apt, F. S. de Boer, and E. Olderog, *Verification of Sequential and Concurrent Programs*, ser. Texts in Computer Science. Springer, 2009.

[9] A. ARM, "Architecture reference manual-armv8, for armv8-a architecture profile," *ARM Limited, Dec*, 2017.

[10] A. Armstrong and B. Dongol, "Modularising opacity verification for hybrid transactional memory," in *FORTE.* Springer, 2017, pp. 33–49.

[11] A. Armstrong, B. Dongol, and S. Doherty, "Proving opacity via linearizability: a sound and complete method," in *FORTE.* Springer, 2017, pp. 50–66.

[12] K. Arun, D. Mishra, and B. Panda, "Empirical analysis of architectural primitives for nvram consistency," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC).* IEEE, 2021, pp. 172–181.

[13] H. Attiya, O. Ben-Baruch, and D. Hendler, “Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory,” in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, 2018, pp. 7–16.

[14] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky, “Safety of live transactions in transactional memory: Tms is necessary and sufficient,” in *Distributed Computing: 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings 28.* Springer, 2014, pp. 376–390.

[15] H. Avni, E. Levy, and A. Mendelson, “Hardware transactions in nonvolatile memory,” in *Distributed Computing: 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings.* Springer, 2015, pp. 617–630.

[16] H. Barringer, “The use of temporal logic in the compositional specification of concurrent systems,” in *Temporal logics and their applications*, 1987, pp. 53–90.

[17] H. A. Beadle, W. Cai, H. Wen, and M. L. Scott, “Nonblocking persistent software transactional memory,” in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 429–430.

[18] O. Ben-Baruch, D. Hendler, and M. Rusanovsky, “Upper and lower bounds on the space complexity of detectable objects,” in *Proceedings of the 39th Symposium on Principles of Distributed Computing*, 2020, pp. 11–20.

[19] N. Ben-David, G. E. Blelloch, M. Friedman, and Y. Wei, “Delay-free concurrency on faulty persistent memory,” in *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, 2019, pp. 253–264.

[20] P. A. Bernstein, V. Hadzilacos, N. Goodman *et al.*, *Concurrency control and recovery in database systems.* Addison-wesley Reading, 1987, vol. 370.

[21] R. Berryhill, W. Golab, and M. Tripunitara, “Robust shared objects for non-volatile main memory,” in *19th International Conference on Principles of Distributed Systems (OPODIS 2015).* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[22] E. Bila, J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, and H. Wehrheim, “Modularising verification of durable opacity,” *Logical Methods in Computer Science*, vol. 18, 2022.

[23] E. Bila, S. Doherty, B. Dongol, J. Derrick, G. Schellhorn, and H. Wehrheim, “Defining and verifying durable opacity: Correctness for persistent software transactional memory,” in *FORTE*, ser. Lecture Notes in Computer Science, A. Gotsman and A. Sokolova, Eds., vol. 12136. Springer, 2020, pp. 39–58. [Online]. Available: https://doi.org/10.1007/978-3-030-50086-3_3

[24] E. V. Bila, B. Dongol, O. Lahav, A. Raad, and J. Wickerson, “View-based owicki-gries reasoning for persistent x86-tso,” in *ESOP*, ser. Lecture Notes in Computer Science, I. Sergey, Ed., vol. 13240. Springer, 2022, pp. 234–261. [Online]. Available: https://doi.org/10.1007/978-3-030-99336-8_9

[25] S. Böhme and T. Nipkow, “Sledgehammer: Judgement day,” in *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, ser. LNCS, J. Giesl and R. Hähnle, Eds., vol. 6173. Springer, 2010, pp. 107–121.

[26] S. Brookes and P. W. O'Hearn, "Concurrent separation logic," *ACM SIGLOG News*, vol. 3, no. 3, pp. 47–65, 2016.

[27] T. Brown and S. Ravi, "On the cost of concurrency in hybrid transactional memory," *arXiv preprint arXiv:1907.02669*, 2019.

[28] T. Chajed, "Verifying a concurrent, crash-safe file system with sequential reasoning," Ph.D. dissertation, Massachusetts Institute of Technology, 2022.

[29] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich, "Verifying concurrent, crash-safe systems with perennial," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 243–258.

[30] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 433–452, 2014.

[31] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using crash hoare logic for certifying the fscq file system," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 18–37.

[32] K. Cho, S. H. Lee, A. Raad, and J. Kang, "Revamping hardware persistency models: view-based and axiomatic persistency models for Intel-x86 and Armv8," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 16–31.

[33] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach," in *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1983, pp. 117–126.

[34] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 105–118, 2011.

[35] R. Colvin, S. Doherty, and L. Groves, "Verifying concurrent data structures by simulation," *Electronic Notes in Theoretical Computer Science*, vol. 137, no. 2, pp. 93–110, 2005.

[36] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 133–146.

[37] A. Correia, P. Felber, and P. Ramalhete, "Romulus: Efficient algorithms for persistent transactional memory," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018, pp. 271–282.

[38] L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear, "Transactional mutex locks," in *Euro-Par 2010-Parallel Processing: 16th International Euro-Par Conference, Ischia, Italy, August 31-September 3, 2010, Proceedings, Part II 16*. Springer, 2010, pp. 2–13.

[39] S. Dalvandi, S. Doherty, B. Dongol, and H. Wehrheim, "Owicki-gries reasoning for C11 RAR," in *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, ser. LIPIcs, R. Hirschfeld and T. Pape, Eds., vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 11:1–11:26.

[40] S. Dalvandi, S. Doherty, B. Dongol, and H. Wehrheim, "Owicki-Gries reasoning for C11 RAR (artifact)," *Dagstuhl Artifacts Ser.*, vol. 6, no. 2, pp. 15:1–15:2, 2020.

[41] S. Dalvandi and B. Dongol, "Implementing and verifying release-acquire transactional memory in C11," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, pp. 1817–1844, 2022. [Online]. Available: https://doi.org/10.1145/3563352

[42] S. Dalvandi and B. Dongol, "Implementing and verifying release-acquire transactional memory in c11," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 1817–1844, 2022.

[43] S. Dalvandi, B. Dongol, S. Doherty, and H. Wehrheim, "Integrating Owicki-Gries for C11-style memory models into Isabelle/HOL," *Journal of Automated Reasoning*, vol. 66, no. 1, pp. 141–171, 2022.

[44] J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, O. Travkin, and H. Wehrheim, "Mechanized proofs of opacity: a comparison of two techniques," *Formal Aspects of Computing*, vol. 30, pp. 597–625, 2018.

[45] J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, and H. Wehrheim, "Verifying correctness of persistent concurrent data structures: a sound and complete method," *Formal Aspects of Computing*, vol. 33, no. 4-5, pp. 547–573, 2021.

[46] J. Derrick, B. Dongol, G. Schellhorn, O. Travkin, and H. Wehrheim, "Verifying opacity of a transactional mutex lock," in *FM 2015: Formal Methods: 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings 20.* Springer, 2015, pp. 161–177.

[47] S. Doherty, B. Dongol, J. Derrick, G. Schellhorn, and H. Wehrheim, "Proving opacity of a pessimistic stm," in *20th International Conference on Principles of Distributed Systems (OPODIS 2016).* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[48] S. Doherty, B. Dongol, H. Wehrheim, and J. Derrick, "Verifying C11 programs operationally," in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, J. K. Hollingsworth and I. Keidar, Eds. ACM, 2019, pp. 355–365.

[49] S. Doherty, L. Groves, V. Luchangco, and M. Moir, "Formal verification of a practical lock-free queue algorithm," in *FORTE.* Springer, 2004, pp. 97–114.

[50] S. Doherty, L. Groves, V. Luchangco, and M. Moir, "Towards formally specifying and verifying transactional memory," *Formal Aspects of Computing*, vol. 25, pp. 769–799, 2013.

[51] B. Dongol and J. Derrick, "Verifying linearisability: A comparative survey," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, pp. 1–43, 2015.

[52] B. Dongol and J. Derrick, "Verifying linearisability: A comparative survey," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 19:1–19:43, 2015. [Online]. Available: https://doi.org/10.1145/2796550

[53] E. D'Osualdo, A. Raad, and V. Vafeiadis, "The path to durable linearizability," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, pp. 748–774, 2023. [Online]. Available: https://doi.org/10.1145/3571219

[54] D. Dziuma, P. Fatourou, and E. Kanellou, "Consistency for transactional memory computing," *Transactional Memory. Foundations, Algorithms, Tools, and Applications: COST Action Euro-TM IC1001*, pp. 3–31, 2015.

[55] P. Ekemark, Y. Yao, A. Ros, K. Sagonas, and S. Kaxiras, "Tsoper: Efficient coherence-based strict persistency," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 125–138.

[56] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 237–246.

[57] P. Felber, V. Gramoli, and R. Guerraoui, "Elastic transactions," in *Distributed Computing: 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings 23*. Springer, 2009, pp. 93–107.

[58] X. Feng, R. Ferreira, and Z. Shao, "On the relationship between concurrent separation logic and assume-guarantee reasoning," in *ESOP*. Springer, 2007, pp. 173–188.

[59] Y. Fridman, Y. Snir, M. Rusanovsky, K. Zvi, H. Levin, D. Hendler, H. Attiya, and G. Oren, "Assessing the use cases of persistent memory in high-performance scientific computing," in *2021 IEEE/ACM 11th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE, 2021, pp. 11–20.

[60] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank, "A persistent lock-free queue for non-volatile memory," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 28–40, 2018.

[61] E. Giles, K. Doshi, and P. Varman, "Hardware transactional persistent memory," in *Proceedings of the International Symposium on Memory Systems*, 2018, pp. 190–205.

[62] E. R. Giles, K. Doshi, and P. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–14.

[63] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single machine graph analytics on massive datasets using intel optane dc persistent memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 8.

[64] H. Gorjiara, G. H. Xu, and B. Demsky, "Jaaru: Efficiently model checking persistent memory programs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 415–428.

[65] J. E. Gottschlich, M. Vachharajani, and J. G. Siek, “An efficient software transactional memory using commit-time invalidation,” in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010, pp. 101–110.

[66] J. Gu, Q. Yu, X. Wang, Z. Wang, B. Zang, H. Guan, and H. Chen, “Pisces: A scalable and efficient persistent transactional memory.” in *USENIX Annual Technical Conference*, 2019, pp. 913–928.

[67] R. Guerraoui and M. Kapalka, “On the correctness of transactional memory,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 175–184.

[68] R. Guerraoui and M. Kapałka, “Principles of transactional memory,” *Synthesis Lectures on Distributed Computing*, vol. 1, no. 1, pp. 1–193, 2010.

[69] R. Guerraoui and R. R. Levy, “Robust emulations of shared memory in a crash-recovery model,” in *24th International Conference on Distributed Computing Systems, 2004. Proceedings.* IEEE, 2004, pp. 400–407.

[70] T. Harris, J. Larus, and R. Rajwar, “Transactional memory,” *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–263, 2010.

[71] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit, “A lazy concurrent list-based set algorithm,” in *OPODIS.* Springer, 2006, pp. 3–16.

[72] M. Herlihy, “A methodology for implementing highly concurrent data structures,” in *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, 1990, pp. 197–206.

[73] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[74] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[75] P. W. Hutto and M. Ahamad, “Slow memory: Weakening consistency to enhance concurrency in distributed shared memories,” in *Proceedings., 10th International Conference on Distributed Computing Systems.* IEEE Computer Society, 1990, pp. 302–303.

[76] D. Imbs and M. Raynal, “Virtual world consistency: A condition for stm systems (with a versatile protocol with invisible read operations),” *Theoretical Computer Science*, vol. 444, pp. 113–127, 2012.

[77] Intel Corporation, “Persistent Memory Programming,” 2015. [Online]. Available: https://pmem.io/

[78] Intel Corporation, “Intel 64 and IA-32 Architectures Optimization Reference Manual,” 2021. [Online]. Available: https://software.intel.com/content/dam/develop/external/us/en/documents-tps/64-ia-32-architectures-optimization-manual.pdf

[79] J. Izraelevitz, T. Kelly, and A. Kolli, “Failure-atomic persistent memory updates via justdo logging,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 427–442, 2016.

[80] J. Izraelevitz, H. Mendes, and M. L. Scott, "Brief announcement: Preserving happens-before in persistent memory," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, 2016, pp. 157–159.

[81] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model," in *International Symposium on Distributed Computing*. Springer, 2016, pp. 313–327.

[82] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[83] J. Jeong, J. Hong, S. Maeng, C. Jung, and Y. Kwon, "Unbounded hardware transactional memory for a hybrid dram/nvm memory system," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 525–538.

[84] J. Jeong and C. Jung, "Pmem-spec: persistent memory speculation (strict persistency can trump relaxed persistency)," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 517–529.

[85] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 660–671.

[86] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: Durable hardware transactional memory," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 452–465.

[87] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 361–372.

[88] J. Kaiser, H.-H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis, "Strong logic for weak memory: Reasoning about release-acquire consistency in Iris," in *ECOOP*, 2017.

[89] J.-O. Kaiser, H.-H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis, "Strong logic for weak memory: Reasoning about release-acquire consistency in iris," in *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[90] A. Khyzha and O. Lahav, "Taming x86-tso persistency," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–29, 2021.

[91] A. Khyzha and O. Lahav, "Abstraction for crash-resilient objects," in *ESOP*. Springer International Publishing Cham, 2022, pp. 262–289.

[92] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 399–411.

[93] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, T. F. Wenisch, and S. Computing, "Persistency programming 101," in *Non-Volatile Memories Workshop*, 2015.

[94] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[95] R. M. Krishnan, J. Kim, A. Mathew, X. Fu, A. Demeri, C. Min, and S. Kannan, "Durable transactional memory can scale with timestone," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 335–349.

[96] O. Lahav and V. Vafeiadis, "Owicki-gries reasoning for weak memory models," in *Automata, Languages, and Programming: 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II 42*. Springer, 2015, pp. 311–323.

[97] O. Lahav and V. Vafeiadis, "Owicki-Gries reasoning for weak memory models," in *Automata, Languages, and Programming*, M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 311–323.

[98] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers c-28*, vol. 9, pp. 690–691, 1979.

[99] M. Lesani, V. Luchangco, and M. Moir, "Putting opacity in its place," in *Workshop on the theory of transactional memory*, 2012, pp. 137–151.

[100] N. Li and W. Golab, "Detectable sequential specifications for recoverable shared objects," in *35th International Symposium on Distributed Computing (DISC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[101] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 329–343, 2017.

[102] D. E. Lowell and P. M. Chen, "Free transactions with rio vista," *ACM SIGOPS Operating Systems Review*, vol. 31, no. 5, pp. 92–101, 1997.

[103] N. Lynch and F. Vaandrager, "Forward and backward simulations," *Information and Computation*, vol. 121, no. 2, pp. 214–233, 1995.

[104] N. A. Lynch and M. R. Tuttle, "Hierarchical correctness proofs for distributed algorithms," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, 1987, pp. 137–151.

[105] A. Matveev and N. Shavit, "Reduced hardware norec: A safe and scalable hybrid transactional memory," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 59–71, 2015.

[106] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson, "Atomic in-place updates for non-volatile main memories with kamino-tx," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 499–512.

[107] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996, pp. 267–275.

[108] A. Mokkedem and D. Méry, "On using temporal logic for refinement and compositional verification of concurrent systems," *Theoretical Computer Science*, vol. 140, no. 1, pp. 95–138, 1995.

[109] O. Müller, "I/o automata and beyond: Temporal logic and abstraction in isabelle," in *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs' 98 Canberra, Australia September 27–October 1, 1998 Proceedings 11*. Springer, 1998, pp. 331–348.

[110] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey III, D. R. Chakrabarti, and M. L. Scott, "Dalí: A periodically persistent hash map," in *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[111] T. Nipkow and L. P. Nieto, "Owicki/gries in isabelle/hol," in *Fundamental Approaches to Software Engineering: Second International Conference, FASE'99, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999. Proceedings 2*. Springer, 1999, pp. 188–203.

[112] G. Ntzik, P. da Rocha Pinto, and P. Gardner, "Fault-tolerant resource reasoning," in *Programming Languages and Systems: 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30-December 2, 2015, Proceedings 13*. Springer, 2015, pp. 169–188.

[113] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley db." in *USENIX Annual Technical Conference, FREENIX Track*, 1999, pp. 183–191.

[114] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs i," *Acta informatica*, vol. 6, no. 4, pp. 319–340, 1976.

[115] S. S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Informatica*, vol. 6, pp. 319–340, 1976.

[116] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM (JACM)*, vol. 26, no. 4, pp. 631–653, 1979.

[117] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 265–276, 2014.

[118] A. Raad, M. Doko, L. Rozic, O. Lahav, and V. Vafeiadis, "On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 68:1–68:31, 2019. [Online]. Available: https://doi.org/10.1145/3290381

[119] A. Raad, O. Lahav, and V. Vafeiadis, "Persistent Owicki-Gries reasoning: a program logic for reasoning about persistent programs on Intel-x86," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 151:1–151:28, 2020.

[120] A. Raad, L. Maranget, and V. Vafeiadis, "Extending intel-x86 consistency and persistency: formalising the semantics of intel-x86 memory types and non-temporal stores," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–31, 2022. [Online]. Available: https://doi.org/10.1145/3498683

[121] A. Raad and V. Vafeiadis, "Persistence semantics for weak memory: Integrating epoch persistency with the tso memory model," *POPL*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[122] A. Raad, J. Wickerson, G. Neiger, and V. Vafeiadis, "Persistency semantics of the intel-x86 architecture," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 11:1–11:31, 2020.

[123] A. Raad, J. Wickerson, and V. Vafeiadis, "Weak persistency semantics from the ground up: formalising the persistency semantics of armv8 and transactional models," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 135:1–135:27, 2019. [Online]. Available: https://doi.org/10.1145/3360561

[124] P. Ramalhete, A. Correia, and P. Felber, "Efficient algorithms for persistent transactional memory," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 1–15.

[125] P. Ramalhete, A. Correia, P. Felber, and N. Cohen, "Onefile: A wait-free persistent transactional memory," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE, 2019, pp. 151–163.

[126] M. Raynal, G. Thia-Kime, and M. Ahamad, "From serializable to causal transactions for collaborative applications," in *EUROMICRO 97. Proceedings of the 23rd EUROMICRO Conference: New Frontiers of Information Technology (Cat. No. 97TB100167).* IEEE, 1997, pp. 314–321.

[127] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 672–685.

[128] A. M. Rudoff, "Deprecating the pcommit instruction," https://www.intel.com/content/www/us/en/developer/articles/technical/deprecate-pcommit-instruction.html/.

[129] S. Scargall, *Programming persistent memory: A comprehensive guide for developers.* Springer Nature, 2020.

[130] R. Sears and E. Brewer, "Stasis: Flexible transactional storage," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 29–44.

[131] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-tso: a rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, 2010. [Online]. Available: https://doi.org/10.1145/1785414.1785443

[132] N. Shavit and D. Touitou, "Software transactional memory," in *PODC*, 1995, pp. 204–213.

[133] K. Siek and P. T. Wojciechowski, “Atomic rmi: A distributed transactional memory framework,” *International Journal of Parallel Programming*, vol. 44, pp. 598–619, 2016.

[134] K. Siek and P. T. Wojciechowski, “Last-use opacity: a strong safety property for transactional memory with prerelease support,” *Distributed Computing*, vol. 35, no. 3, pp. 265–301, 2022.

[135] C. SPARC International, Inc, *The SPARC architecture manual: version 8.* Prentice-Hall, Inc., 1992.

[136] L. Stefanesco, A. Raad, and V. Vafeiadis, “Specifying and verifying persistent libraries,” *CoRR*, vol. abs/2306.01614, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.01614

[137] L. Sun, Y. Lu, and J. Shu, “Dp2: Reducing transaction overhead with differential and dual persistency in persistent memory,” in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, 2015, pp. 1–8.

[138] A. Turon, V. Vafeiadis, and D. Dreyer, “Gps: Navigating weak memory with ghosts, protocols, and separation,” in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 691–707.

[139] V. Vafeiadis and M. Parkinson, “A marriage of rely/guarantee and separation logic,” in *CONCUR 2007–Concurrency Theory: 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007. Proceedings 18.* Springer, 2007, pp. 256–271.

[140] S. F. Vindum and L. Birkedal, “Spirea: A mechanized concurrent separation logic for weak persistent memory,” 2022.

[141] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 91–104, 2011.

[142] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen, “Persistent transactional memory,” *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 58–61, 2014.

[143] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, “Characterizing and modeling non-volatile memory systems,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 2020, pp. 496–508.

[144] Y. Wei, N. Ben-David, M. Friedman, G. E. Blelloch, and E. Petrank, “Flit: a library for simple and efficient persistent algorithms,” in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 309–321.

[145] H. Wen, W. Cai, M. Du, L. Jenkins, B. Valpey, and M. L. Scott, “A fast, general system for buffered persistent data structures,” in *50th International Conference on Parallel Processing*, 2021, pp. 1–11.

[146] D. Wright, M. Batty, and B. Dongol, “Owicki-gries reasoning for c11 programs with relaxed dependencies,” in *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings.* Springer, 2021, pp. 237–254.

[147] L. Xiang, X. Zhao, J. Rao, S. Jiang, and H. Jiang, “Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 488–505.

[148] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, “An empirical guide to the behavior and use of scalable persistent memory.” in *FAST*, vol. 20, 2020, pp. 169–182.

[149] V. Yashina, “Intel® optane™ persistent memory – memory mode decision guide,” https://www.intel.com/content/www/us/en/developer/articles/guide/intel-optane-persistent-memory-decision-guide.html/.

[150] P. Zardoshti, T. Zhou, Y. Liu, and M. Spear, “Optimizing persistent memory transactions,” in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE, 2019, pp. 219–231.