Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Software Transactional Memory for Non-Volatile Memory

Author:
Rini Banerjee

Supervisor:
Dr Azalea Raad

Second Marker:
Prof. Paul Kelly

June 22, 2022

Abstract

Software transactional memory (STM) is a popular programming language abstraction that uses high-level blocks of code known as transactions to make correct concurrent programming easier. The formal semantics of STM have been explored extensively in the literature via the development of transactional consistency models, the vast majority of which make the implicit assumption that the underlying memory is volatile (e.g. RAM). This may no longer be a realistic assumption going forward, since emerging non-volatile memory (NVM) technologies are expected to supplant volatile memory in the near future due to their high speed, byte-addressability and durability. Transactional consistency models that also account for persistency guarantees have been largely unexplored to date, with the only existing model in the literature satisfying these criteria being unrealistic in terms of performance due to its strong consistency guarantees under a model known as serialisability. This motivates the need for a mathematically precise STM model that can provide both consistency *and* persistency guarantees while maintaining good performance.

To fill this gap, we present a formal transactional persistency model that extends the well-known snapshot isolation (SI) model with durability guarantees. Specifically, we present a declarative model for SI in the context of non-volatile memory, which we formulate using execution graphs. We also present a corresponding reference implementation of this model over the Intel-x86 architecture and prove that this implementation is sound. Finally, we present a more general version of our implementation that has greater practical use. Our work demonstrates the correctness of our first implementation since we have shown that every possible execution of this implementation must satisfy the axioms of our declarative model via the soundness proof. This work also improves upon the state of the art by providing better performance due to our choice of using SI over serialisability.

Acknowledgements

I would like to thank my supervisor, Azalea Raad, for being such a fantastic guiding force for me throughout this project. I am immensely grateful to her for the hours of technical discussions, whiteboard diagrams and Teams calls. She has been a mentor to me in every sense of the word.

I would also like to thank Jackie Bell for being a wonderful personal tutor. She has supported me through and through these last few years and I feel very lucky to have had someone so genuinely kind as my personal tutor.

Thank you to all the friends I have made at Imperial over the last four years. You have been my support system through every deadline, application and stress-point, and the sense of togetherness we have managed to sustain through it all – even through a pandemic! – is really special.

Finally, thank you to my parents for your immense love and support always, especially over the last four years.

Contents

1 | Introduction

Concurrent programs are infamously difficult to write. Locking has long been the default synchronisation technique within concurrent programming, but despite its simplicity, it comes with a myriad of disadvantages, arguably the most restrictive of which is its ability to cause deadlocks. As an alternative to lock-based techniques, *software transactional memory* (STM) has been proposed as a language-level abstraction designed to simplify the task of writing concurrent programs that are both efficient and correct. It utilises *transactions*, a synchronisation mechanism adapted from the databases world, to provide the "illusion of blocks of code"[1] that execute atomically and in isolation from all other concurrent blocks. Conceptually, STM allows programmers to focus on the algorithmic elements of their code by shielding them from low-level synchronisation details, and STM implementations for a number of widely-used programming languages[2, 3, 4] have emerged in recent years.

While various models have been introduced in the literature to formalise the behaviours of STM at different 'levels' of synchronisation (known as *transactional consistency models*), these have almost exclusively been in the context of volatile memory (e.g. RAM), which is fast and byte-addressable but loses its contents in the event of a crash. Meanwhile, recent advances in *non-volatile memory* (NVM) technologies have led to an emerging class of memory that exhibits the speed and byte-addressability of RAM while also ensuring the *durability* of data in memory, meaning it can be recovered after a crash. As a result, NVM is expected to eventually supplant its volatile counterpart[5]. In order to be able to make use of existing memory models (including those for STM) in the context of these novel memory technologies, their guarantees must be able to account for the set of permissible persistency behaviours as well, via formal *persistency models*[6].

Ensuring the correctness of persistent programs is, however, far from easy. The main difficulty lies in the relationship between what are known as the *consistency order* and the *persistency order*, with the former describing the order in which a single thread's memory instructions (e.g. writes) are made *visible* to other threads, and the latter describing the order in which writes are *propagated* to persistent memory. Under certain models, these orders may not necessarily coincide in the interest of enhancing performance, but this comes at the cost of increased unpredictability, making reasoning about such models significantly more difficult.

Even sequential (single-threaded) programs may not always behave as expected in the context of NVM. This type of behaviour would be most apparent in the event of a crash, where the out-of-order propagation of writes to memory might mean an unexpected subset of the program was able to persist to memory before the crash. For example, in the simple sequential program $x := 1; y := 1$, if a crash occurs before the program is allowed to finish executing, we may find that y has the value 1 in memory but x still contains its initial value of 0, due to these writes persisting to memory in a different order to that of the program execution. These problems are compounded when dealing with multithreaded programs, indicating that there is a strong need for models that can formally define the permissible behaviours of concurrent programs in the persistent setting.

Existing work in the development of transactional consistency models for non-volatile memory is scarce. The only known model in the literature that accounts for both the consistency and persistency guarantees of STM, known as PSER[7, 5], enforces a total order on transactions, meaning that they must all appear to have executed one after each other. This model is known as *serialisability*, and is generally considered to be too expensive due to the strength of the synchronisation it imposes on transactions. An alternative transactional model, known as *snapshot isolation* (SI), weakens these guarantees by allowing transactions to execute concurrently as long as they do not try to write to the same location in memory. SI therefore provides much better performance than serialisability, but has not been investigated in the context of

non-volatile memory to date.

This forms the premise of this project, where our main objective has been to develop the *first* formal STM model for SI that also accounts for persistency guarantees. We have named this the **durable snapshot isolation (DSI)** model, and an outline of our work is given below.

1.1 Contributions

The main technical contributions of this project are as follows:

- We develop the **declarative (axiomatic) DSI model** (Chapter 4), which is, to our knowledge, the first transactional model to provide the consistency guarantees of snapshot isolation in the context of non-volatile memory. Our model is an instantiation of a general declarative framework for persistency models[7] (Chapter 3) which is commonly used within the literature[7, 5].
- We provide an **implementation and recovery mechanism of DSI in Px86 ('persistent Intel-x86')** (Chapter 5). This demonstrates that DSI can be practically implemented using a mainstream architecture that provides support for non-volatile memory. We make use of existing fine-grained Intel-x86 persistency primitives and an efficient lock library as mechanisms for persistency and concurrency control respectively.
- We **prove soundness for our DSI implementation** (Chapter 6). We achieve this by formulating the execution trace of our implementation as a sequence of events and using this trace to construct implementation- and abstract-level execution graphs which follow the general framework of Chapter 3. We then prove a number of lemmas to show that our implementation provides both the consistency and persistency guarantees of DSI, and make use of the axioms provided by existing declarative models for Px86[5, 8] in these proofs. This soundness proof demonstrates *correct* Px86-to-DSI implementation; thus, we have provided a basis for our implementation being used as a *high-level persistent transactional library* that ensures the consistency guarantees of snapshot isolation, which is, to the best of our knowledge, the first of its kind.
- We provide a **more general implementation of DSI in Px86** (Chapter 7), which is more practical than the DSI implementation of Chapter 5 as it makes fewer assumptions on the transactional information known in advance of the implementation being executed. The soundness proof for this more general implementation is omitted as it is analogous to that of the previous DSI implementation from Chapter 5.

2 | Background

This chapter introduces a number of background concepts which are important to understand in preparation for following this project. It starts below with a high-level introduction to non-volatile memory (NVM), and then continues with an explanation of formal memory consistency and persistency models and how they relate to NVM. Finally, we discuss transactional memory and transactional consistency models.

2.1 Non-Volatile Memory

Historically, memory has been divided into two main categories: *volatile* and *non-volatile* memory. Volatile memory, such as random-access memory (RAM), is typically fast and byte-addressable, but loses its contents in the event of a crash. Meanwhile, non-volatile memory (NVM), such as hard disk drives, is *persistent*, meaning its contents are retained after device power is lost. NVM has traditionally been much slower than volatile memory and has only provided block-addressability (as opposed to the byte-addressability of RAM). Therefore, there has traditionally been a tradeoff between the two types of memory, with each category offering a different set of advantages.

However, NVM technologies have emerged in recent years that may "render this dichotomy obsolete"[9] by providing the performance and byte-addressability of RAM as well as the durability of hard disks. As a result, there has been a surge in NVM research over the past decade[10, 11, 5, 7], with NVM expected to eventually supplant traditional volatile memory. Concrete examples of these newer NVM technologies include Intel Optane[12], phase change memory[13], spin-transfer torque RAM[14] and memristors[15].

2.2 Memory Consistency

Memory consistency models describe the permitted behaviours of programs by constraining the visible order of memory accesses (i.e. reads and writes) to different memory locations[7]. Informally, a memory consistency model can be viewed as a "contract" between the programmer, the compiler and the hardware of a given machine[16]: the model specifies to the programmer which behaviours are possible while also enforcing restrictions on the compiler and hardware-level optimisations that can be implemented[17]. Many such consistency models exist in the literature, at both the **hardware** (architecture) level[18, 19] and the **software** (programming language) level[20, 21]. Hardware-level consistency models are concerned with how multiple *processors* behave with regards to some shared memory, whereas at the software level, consistency models define the permissible behaviours between software *threads*.

Memory consistency models can also vary in strength. Stronger (*stricter*) consistency models enforce more constraints on the visible shared memory between threads, whereas weaker (more *relaxed*) consistency models allow more unexpected behaviours and subtly different views of shared memory between threads. While relaxed memory models are more difficult to reason about for programmers, they also provide a more realistic description of the permissible behaviours on most modern software and hardware platforms. These behaviours, although often counterintuitive, are observable in most multiprocessors and languages due to the improved performance they help to facilitate.

In general, mathematically precise memory consistency models that describe the allowed behaviours of multithreaded programs and multiprocessors provide a strong foundation for ensuring the correctness and reliability of major programming languages and architectures. In the remainder of this section, we discuss some notable memory consistency models and consider the strengths and weaknesses of each model.

2.2.1 Sequential Consistency

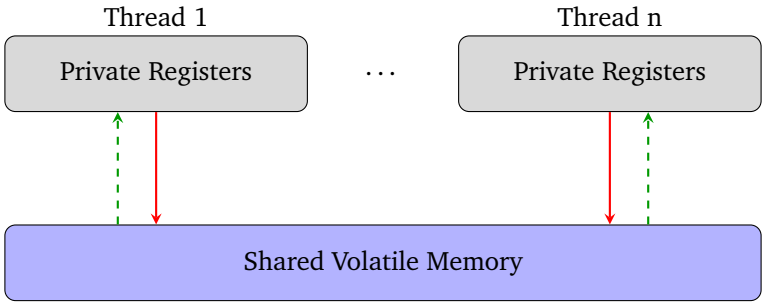

Figure 2.1: The storage subsystem of the sequential consistency memory model

The model that has underpinned much of the literature for memory consistency models is that of *sequential consistency* (SC), also known as interleaving semantics. In the context of concurrent programs, sequential consistency restricts what is known as the *volatile memory order* (i.e. the order in which memory instructions such as writes are made visible to other threads) to agree with the program execution order. Intuitively, this means that the instructions of each constituent thread of a concurrent program are executed in order within the thread, whereas the instructions of different threads can be interleaved arbitrarily[22]. The storage subsystem of SC is given in Fig. 2.1, where reads (dashed green arrows) and writes (solid red arrows) occur directly between the registers that are private to each thread and the volatile memory that is shared by all threads.

Consider the simple two-threaded program in Fig. 2.2 (where the threads are separated using ||)[22]. We assume from now on that x and y are shared memory locations that are visible to all threads and initialised to 0, and that a and b are local registers that are only accessible by the thread they are used in. This means that an instruction of the form $x := 1$ is a *memory write* to x while an instruction of the form $a := x$ is a *memory read* from x, and this can be generalised to all memory locations and local registers. Under sequential consistency, the *program order* of each thread must be respected, which means that $x := 1$ must take place before $a := y$ and $y := 1$ must take place before $b := x$. Under this restriction, a range of possible interleavings between the two threads are allowed to take place, which in turn can lead to different end-values for the local registers a and b once the program has finished executing (we call these values v_a and v_b respectively). For example, if all of the first (left) thread executes before all of the second (right) thread, then the program will execute in the following order: (1) the value 1 will be written to x due to the write $x := 1$; (2) the value 0 will be read from y into a due to the read $a := y$ and the fact that y has not been written to since it was initialised; (3) (1) the value 1 will be written to y due to the write $y := 1$; and (4) the value 1 will be read from x into b due to the read $b := x$. Therefore, at the end of program execution, we have $v_a = 0$ and $v_b = 1$.

A different possible interleaving would occur if the writes $x := 1$ and $y := 1$ both took place before the reads into a and b (i.e. if the program was executed as $x := 1$; $y := 1$; $a := y$; $b := x$). By reasoning about the program execution for this interleaving in a similar way to the previous example, we have $v_a = 1$ and $v_b = 1$ at the end of program execution. Both of these interleavings are allowed under sequential consistency.

We note that due to the restrictions imposed by SC, it is not possible to observe any interleavings that do not respect the program order of the individual threads. For example, we cannot have $v_a = 0$ and $v_b = 0$, since this would mean the reads $a := y$ and $b := x$ both need to have occurred before the writes $x := 1$ and $y := 1$. Herein lies the issue with the SC model: although it is conceptually simple and intuitive to understand, it is too strong of a model for real-world platforms. In particular, it does not allow for the *out-of-order* behaviours that occur on most platforms as a result of compiler and hardware-level optimisations. Therefore, it would be desirable to have a formal memory consistency model that is more *relaxed* than SC, and which permits the behaviours of real-world architectures and languages.

$$\begin{array}{l|\!|l} x := 1; & y := 1; \\ a := y; & b := x; \end{array}$$

Figure 2.2: An example program with two threads. Sequential consistency ensures that the program order of the individual threads is respected.

2.2.2 Total Store Ordering (TSO) on Intel-x86

While the sequential consistency model can serve as a high-level abstraction of a processor, it is too strong to account for the often unexpected behaviours of modern multiprocessors, which use sophisticated techniques to achieve higher performance (such as store buffers, hierarchies of local cache and speculative execution[18]). Many of these techniques alter the way in which shared memory appears, meaning that a weaker memory model must be used to capture such behaviours.

Perhaps the most widely-used multiprocessor that makes use of such optimisation techniques is the Intel-x86 processor, which is capable of observing weak behaviours that are not observable under SC. A relaxed memory model that formally describes the permitted consistency behaviours of x86 processors was first introduced as the *total store ordering (x86-TSO) model* by Sewell et al. [18] in 2010. In short, the x86-TSO model is a slightly weakened form of SC in that it allows *write-read reordering* on different locations, with no other reorderings allowed. This means that for some locations x and y, a later read on y is allowed to be reordered before some earlier write on x when x and y are distinct locations[23]. Looking back at our example program from Fig. 2.2, we can see that under the x86-TSO model, it is possible for the reads $a := y$ and $b := x$ to be executed *before* the writes $x := 1$ and $y := 1$ are executed. Therefore, it is possible to have $v_a = v_b = 0$ at the end of the program execution, which was not possible under sequential consistency.

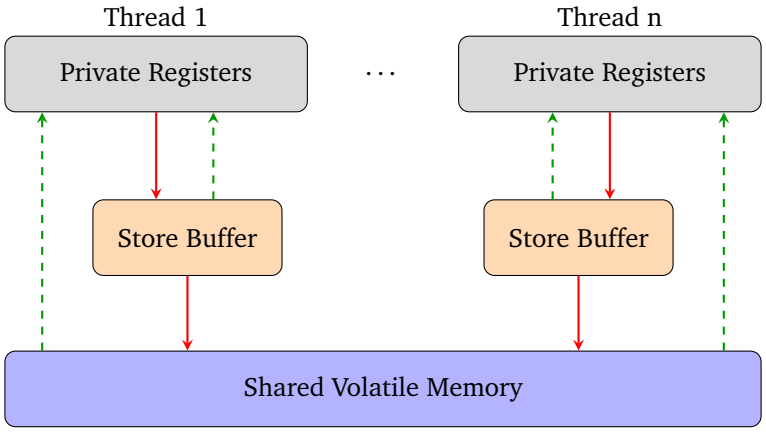

Figure 2.3: The storage subsystem of the x86-TSO memory model

The storage subsystem of the x86-TSO model is given in Fig. 2.3. In this model, each thread is connected to the main (volatile) memory via a first-in-first-out (FIFO) *store buffer*. In contrast with the storage subsystem for sequential consistency (Fig. 2.1), the execution of writes under x86-TSO is no longer immediate, since each write issued by a thread on a given location is first recorded in the thread's store buffer. The pending writes in a given thread's store buffer are propagated in FIFO order to the main memory at non-deterministic points in time. Both of these sets of events are signified by the solid red arrows in Fig. 2.3.

Meanwhile, reads get executed in real time. A thread issuing a read on a location x will first consult its own store buffer. If the buffer contains delayed writes on x that have not yet been propagated to main memory, the thread will read the value of the last buffered write to x, and will read from memory directly otherwise. These sets of events are signified by the dashed green arrows in Fig. 2.3. It is clear from this diagram that the write-read reordering of x86-TSO can be modelled for some distinct locations x, y by delaying the propagation of a write on x from the store buffer to main memory until a later read y has executed in real time[5].

The introduction of the TSO consistency model for the Intel-x86 architecture was a milestone in the semantics and verification literature. However, one limiting assumption made by the x86-TSO model is that the memory being shared between threads is *volatile*. With the emergence of high-performance non-volatile memory in recent years (see Section 2.1), it would be desirable to have formal models that also describe the *persistency* semantics of major architectures like Intel-x86, while maintaining the consistency guarantees that have been discussed in this section; we discuss this in detail in the next section.

2.3 Memory Persistency

With the rise of NVM and persistent programming, analogous concepts and definitions to those discussed in Section 2.2 have been established within the world of persistency. Memory persistency models, first introduced by Pelley et al. [6], formally define the persistency semantics of programs (i.e. what behaviours are permitted following recovery after a crash), by prescribing the order in which the effects of memory instructions are committed to memory (known as the *persistent memory order*). We distinguish between the volatile memory order of consistency models and the persistent memory order of persistency models by differentiating memory *stores*, which denote the process of making an instruction visible to other threads (consistency), from memory *persists*, which denote the process of committing the effects of an instruction durably to persistent memory (persistency). We note that persistency models are an *extension* of consistency models which also account for persistency guarantees[6].

Persistency models are typically categorised along two axes: (1) *strict* versus *relaxed*, and (2) *unbuffered* versus *buffered*.

Strict vs Relaxed Persistency. Strict persistency models are those in which the volatile and persistent memory orders are the same. In other words, the order in which memory instructions are made visible to other threads is the *same* as the order in which the effects of those memory instructions persist to memory. Stricter persistency models can often incur a large slowdown in performance, since execution gets *stalled* by persists, meaning a program cannot execute its next instruction until the effects of the previous one have been persisted to memory. In order to mitigate for this, relaxed persistency models were introduced, which allow the volatile and persistent memory orders to disagree. Relaxed models are able to provide significant improvements in performance, but are also more difficult to reason about due to the unexpected memory states they can cause to be observed upon recovery from a crash.

Buffered vs Unbuffered Persistency. The second dichotomy for persistency models is concerned with when persists occur. Under unbuffered persistency, persists are required to occur *synchronously*, which means the effects of an instruction must be committed to memory as soon as the instruction is executed. Performance can be improved by using buffered persistency[24], under which persists are buffered (i.e. temporarily stored) in a queue, where they "wait" to be committed to memory at some later time. This type of execution is *asynchronous*. One advantage of using a buffered persistency model is that program execution is able to proceed ahead of persists, and so no longer gets stalled. This means that only a *prefix* of the persistent memory order may have persisted upon recovery from a crash, with the remainder having not yet been committed to persistent memory from the buffer at the time of the crash. For situations where buffered persists may need to be committed explicitly (e.g. before performing I/O operations), buffered models usually provide some explicit *persist* instructions that allow the buffer to be "drained" of pending persists, which are then committed to memory. These persist instructions may themselves be synchronous or asynchronous[5].

Various persistency models have been proposed in recent years, many of which have the goal of ensuring the correctness and reliability of architectures which have recently begun to provide support for up-and-coming NVM technologies. These include the Px86[5] and PARMv8[7] models, which provide both consistency *and* persistency guarantees for the mainstream Intel-x86 and ARMv8 architectures, respectively. We spend the remainder of this section discussing the Px86 model in order to get an intuitive understanding of real-world persistency and how it ties in with what we have learnt about consistency thus far.

2.3.1 The Px86 Model

The Px86 ("persistent x86") persistency model was first introduced in January 2020 by Raad et al. [5] as an extension of the x86-TSO consistency model (discussed in Section 2.2.2). In addition to the consistency guarantees provided by the x86-TSO model, Px86 also ensures correct recovery after a crash on x86 machines via the inclusion of persistency guarantees. A diagram of its storage subsystem is given in Fig. 2.4.

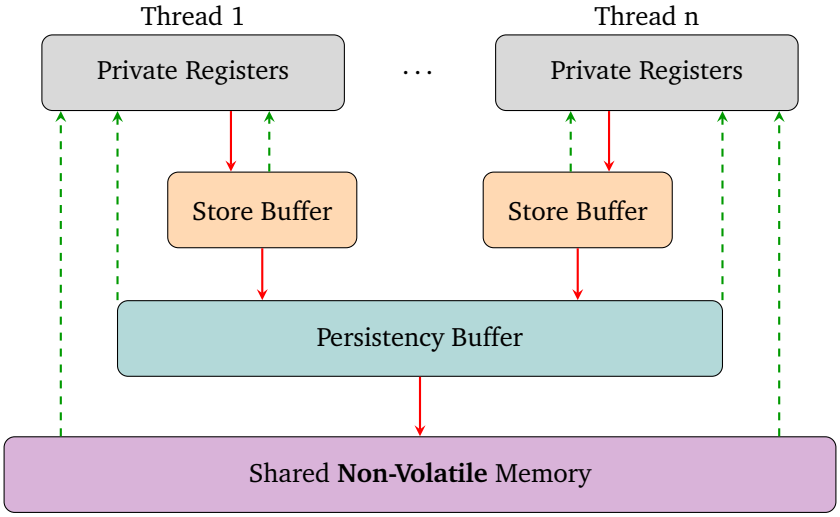

Figure 2.4: The storage subsystem of the Px86 memory model

There are a few key differences between the storage systems of Px86 (Fig. 2.4) and x86-TSO (Fig. 2.3). Firstly, the Px86 storage system has an additional layer, consisting of a FIFO *persistency buffer*, which contains the writes that are "waiting" or pending to be persisted to memory. We note that this persistency buffer is volatile, and so its contents are lost upon a crash; it is also a shared buffer between all threads. Furthermore, the bottom layer of memory is *non-volatile* in the case of Px86, as expected in the persistent setting. As in x86-TSO, each write issued by a thread on a given location is still recorded in the thread's store buffer first, but under Px86 it must get propagated to the persistency buffer before it can then be committed to the shared non-volatile memory. Debuffering writes from a given thread's store buffer to the shared persistency buffer denotes the *store* that is associated with the write, representing that the write is now visible to all threads. Once writes have reached the persistency buffer, they are again debuffered and then propagated to memory at non-deterministic points in time. This second stage of debuffering denotes the *persist* that is associated with the write, meaning the write has been committed durably to memory. This process of debuffering in turn until a given write reaches non-volatile memory is represented by the solid red arrows in Fig. 2.4. Reads also adhere to this three-tier hierarchy, as shown by the dashed green arrows in Fig. 2.4. Threads that wish to read from some location x first consult their individual store buffers for the last buffered write to x if it exists; otherwise the persistency buffer for the last buffered store to x; and otherwise the shared non-volatile memory.

$x := 1;$ $y := 1;$
rec: $x, y \in \{0, 1\}$

Figure 2.5: An example Px86 program and the possible values of x, y upon recovery (given in blue).

Intel-x86 follows a relaxed, buffered persistency model, with the latter property already clear from the existence of the persistency buffer in its storage system. We now consider an example program that demonstrate the additional complexities that can arise in the persistent setting. For example, if a crash occurs during the execution of the program in Fig. 2.5, we have $x, y \in \{0, 1\}$ upon recovery. In particular, the case of $x = y = 0$ could occur even if the program had executed to completion before crashing, and this is due to the *buffered* nature of the Px86 model; the writes to x and y may not have reached the shared non-volatile memory before the crash took place.

Furthermore, due to the *relaxed* nature of the Px86 model, certain behaviours are observable in the event of a crash that are not observable in normal, non-crashing executions. For example, during the normal execution of the program, it is not possible to observe $x = 0, y = 1$ at any point; this is because writes are not allowed to be reordered under Intel-x86 (as we know from the consistency guarantees of the x86-TSO model). Thus, for this program, we have $y = 1 \implies x = 1$ in the case of a non-crashing execution. However, if a crash takes place before all of the program writes have been able to persist to memory, it *is* actually possible to observe $x = 0, y = 1$; this is due to the relaxed nature of the persistency model, which allows the store order and persist orders to be different. Concretely, this could occur if the write to y was propagated from the persistency buffer to memory before this was done for the write to x.

The discussion above has shown how simple sequential programs can display quite nuanced behaviours in the context of non-volatile memory. We note that the difficulty of reasoning about such behaviours is further compounded when dealing with longer, multithreaded programs. Formal persistency models play a crucial part in mitigating for these complexities in a mathematically rigorous way; they ensure the correctness of mainstream architectures like Intel-x86 by defining which behaviours are permitted on these platforms, and therefore help to provide explanations for the counterintuitive behaviours that may be observed.

2.4 Transactions

We now move the focus of our discussion away from memory models and delve into some database theory. In particular, we consider transactions, which are the go-to synchronisation mechanism in the databases world. Due to their strong guarantees and intuitive usability, they have inspired *transactional memory*, an abstraction that simplifies writing concurrent programs. In this section, we provide a brief overview of what transactions are and how they are used within databases, and then explore their adapted shared-memory counterpart, transactional memory. Finally, we discuss some common transactional consistency models, which we define in due course.

2.4.1 Database Transactions

Historically, one of the most crucial components used in the concurrency control and recovery of database systems has been the *transaction*. A transaction can informally be described as an execution of a program that accesses a shared database, and each transaction is expected to satisfy a number of properties in order to ensure that shared data can be accessed and modified safely. In particular, transactions are expected to execute atomically, which means that: (1) each transaction is able to access shared data without interfering with any other transactions, and (2) a transaction's effects are only made permanent if it terminates normally. These guarantees are especially important in a concurrent context, where multiple transactions may be allowed to access the same shared data in parallel.

In order to understand this in more detail, we first note that a database system (DBS) is a collection of modules at the hardware and software level that support commands to access a given database. These commands are known as *database operations*, the most important of which are reads from and writes to the database. Therefore, in the context of a DBS, a transaction can be more formally defined as the execution of one or more programs that include database and transaction operations, with the latter representing a set of commands that control the execution of the transaction itself.

Transaction operations include the *start* operation (which indicates that a program is about to begin executing a new transaction), as well as the *commit* and *abort* operations, which roughly correspond to "success" and "failure" cases for the termination of the transaction respectively. In other words, in the case of a commit, the program in question is relaying to the DBS that the given transaction has terminated normally and that all of its effects should be made permanent as a result. On the other hand, the abort command signifies that the given transaction has not terminated normally, and that none of its effects should be made permanent. In terms of these operations, a transaction can therefore be expressed as a start command, followed by an execution of database operations that either end in a commit or an abort[25].

A transaction is considered to be correct in terms of the validity of data in the given database if it satisfies the following properties, commonly abbreviated as **ACID**[26]:

1. **A**tomicity: Either every operation in the transaction succeeds, or the transaction is aborted and *rolled back*, and so none of the operations persist. Informally, this ensures that every transaction is "all-or-nothing".
2. **C**onsistency: Assuming the data in question was consistent *before* the transaction, it must also be consistent *after* the transaction. Here, consistency means the constraints that are enforced upon the database.
3. **I**solation[1]: The effects of a transaction that is in progress are not visible to any other transaction. In other words, a transaction executes as though no other transactions are executing at the same time, and the effects of a transaction are only made permanent (and, therefore, visible) to future transactions once the original transaction has committed successfully.
4. **D**urability: The results of a transaction persist in the database once it has successfully committed, and these results are guaranteed to survive a crash.

These guarantees help transactions to maintain data integrity in a given database.

We note that although transactions have so far only been introduced in the context of databases, their strong guarantees and usability have made them a useful concurrency abstraction in the context of shared memory. This is known as transactional memory, and we discuss this in more detail in the next section.

2.4.2 Transactional Memory

Transactional memory borrows the concept of transactions from databases and applies it to the shared-memory setting, for use as a programming language abstraction that can simplify the task of writing concurrent programs. It provides the "illusion" of blocks of code that can execute *atomically* and *in isolation* from any other such concurrent blocks[1]. Transactional memory has been gaining widespread adoption within the setting of shared memory due to the strong guarantees that transactions are able to provide, as well as the improvement in both performance and scalability offered by transactions when compared with lock-based synchronisation[27].

Transactional memory can be considered both at the low level by the hardware of a machine (known as hardware transactional memory, or HTM) as well as at the high level within software (known as software transactional memory, or STM). We focus on the latter in this project. STM specifications are often required to go a step further than database transactions by additionally accounting for the interactions between what are known as *mixed-mode* accesses to the same locations[1], which are accesses made by some combination of transactions and non-transactional code.

We note that the ACID guarantees outlined in Section 2.4.1 can also be extended to transactions in the shared-memory setting, where they now ensure the integrity of data in memory rather than in a database.

2.4.3 Transactional Consistency Models

Transactional consistency models formally describe the permitted behaviours of transactions in terms of their consistency guarantees. These models are analogous to memory consistency models (Section 2.2) in that they serve as a "contract" or set of rules to describe which behaviours are permitted for a set of transactions under the given model. The set of allowable behaviours defined by a transactional consistency model usually encompasses both the *internal* behaviours of transactions (i.e. which behaviours are permitted within each transaction) and the *external* behaviours of transactions (i.e. how distinct transactions interact with each other).

While the correctness of transactions and how they interact with data is dependent on whether the ACID properties from Section 2.4.1 hold, it is common for consistency guarantees to be relaxed in consistency models for real-world transactional systems in order to enhance performance. For example,

[1] In database systems, 'isolation level' is the term used to describe consistency. ACID's 'Consistency' property means something different from this!

models like *serialisability* provide stronger consistency guarantees but also incur a performance penalty. Meanwhile, models like *snapshot isolation* have more relaxed consistency guarantees, which results in higher performance but can also cause data inconsistencies. We now discuss these two models, which are well-known in the literature for transactional consistency models, and explore how each model accounts for the tradeoff between consistency and performance.

$$\textbf{Tx1:} \begin{bmatrix} x := 1; \\ \mathtt{a} := y; \end{bmatrix} \quad \Vert \quad \textbf{Tx2:} \begin{bmatrix} y := 1; \\ \mathtt{b} := x; \end{bmatrix}$$

Figure 2.6: An example transactional program with two concurrently executing transactions t76(inspired by [5])

Serialisability. Serialisability is one of the strongest transactional consistency models. It enforces a *total order* for all transactions; this means that all transactions must seem to have executed atomically one after the other in some order[1]. For example, if we consider the transactional program given in Fig. 2.6 and assume x and y are initialised to 0, serialisability would enforce that either **Tx1** executes entirely before **Tx2**, resulting in $\mathtt{a} = 0, \mathtt{b} = 1$, or **Tx2** executes entirely before **Tx1**, resulting in $\mathtt{a} = 1, \mathtt{b} = 0$.

The strong guarantees of serialisability, along with its simple and intuitive semantics, make it the "gold standard" of transactional models. However, these strong guarantees incur a slowdown in performance due to the fact that conflicting transactions cannot execute and commit in parallel. In other words, under serialisability, if more than one transaction tries to read from or write to a given location in memory at the same time, these transactions are aborted and must restart execution. As a result of these properties, serialisability is generally regarded to be an impractical choice of consistency model for transactions.

Snapshot Isolation (SI). One weaker transactional consistency model that attempts to remedy the problems caused by strong models like serialisability is snapshot isolation (SI)[28]. SI allows conflicting transactions to execute concurrently and successfully commit if the transactions do *not* have a write-write conflict. This means that transactions with other types of conflicts (e.g. read-write conflicts) are allowed to execute concurrently without having to abort. This leads to improved performance, since concurrent transactions only need to abort and restart if they are both trying to write to the same location; in the case of any other type of conflict, they are allowed to execute to completion and commit.

Snapshot isolation can be informally expressed in the context of shared memory by using the following *multi-version concurrency control* (MVCC) algorithm[29]:

Algorithm 1 An MVCC Algorithm for Snapshot Isolation

1: Start with a transaction T.
2: Takes a *snapshot* S of shared memory.
3: Execute **T** locally: read operations read directly from S and write operations update S.
4: Check whether any other concurrently executing transaction has written to a location that T has written to. Store this in `T_write_conflicted`.
5: `if (!T_write_conflicted):`
6: Commit changes to T and succeed.
7: `else:`
8: Abort T and restart.

We note that this algorithm is expressed in terms of shared objects, which are commonly used in the context of databases, but the algorithm can easily be extended to the transactional memory setting by considering shared memory locations instead.

One disadvantage of SI is that due to its relaxed consistency guarantees, it can cause *anomalies*; these are inconsistencies caused by multiple transactions accessing the same shared data concurrently[30]. Consider

Figure 2.7: A litmus test illustrating the write-skew anomaly (inspired by [1]).
We assume x, y are initialised to 0

the litmus test shown in Fig. 2.7 which contains two concurrently executing transactions. Firstly, we know that the *snapshots* of shared memory taken by **Tx1** and **Tx2** must contain a value of 0 for both x and y, since these are the values they have in memory before the transactions start their execution. After the writes to x and y during the local execution of **Tx1** and **Tx2** respectively, we know that the following must hold: (1) `a` must read a value of 0 from y, due to the value for y in the snapshot of **Tx1** being 0; and similarly (2) `b` must read a value of 0 from x, due to the value for x in the snapshot of **Tx2** being 0. Since there is no write-write conflict between the transactions, they are both able to commit successfully, and so x and y have both been set to 1 but neither transaction was able to observe the effect of the other's write. This is an example of the *write-skew anomaly*, and is permitted under the SI model[31]. We note that this is one of the few anomalies permitted by SI, and so arguably this is a small price to pay for the improved performance that the model provides. Indeed, SI has been adopted as the default consistency model for most major databases[29], thus demonstrating that despite its permittance of anomalous behaviour in certain situations, it is still a highly useful and practical model for real-world settings.

2.5 Related Work

Now that we have discussed the background required to understand the premise of this work, we can provide an overview of the related work for this project. Specifically, we discuss the **p**ersistent **s**erialisability (PSER) model, introduced by Raad et al. [7] in 2019 as the first model to formalise transactional semantics in the context of non-volatile memory.

2.5.1 Persistent Serialisability (PSER)

The persistent serialisability (PSER) model[7] was the first formal transactional persistency model to be developed. It enforces the transactional consistency model of serialisability (Section 2.4.3) in the context of non-volatile memory, and therefore provides both consistency and persistency guarantees, which are formulated using a declarative (axiomatic) model. PSER was designed with the aim of making persistent programming accessible to a wide range of programmers by abstracting away low-level details, while still maintaining a level of correctness by accounting for the crashing behaviour of programs formally. Specifically, PSER can be seen as a high-level persistent transactional library; it allows users to interact with simple transactions which follow the serialisability model, while the low-level implementation details (e.g. architecture-specific persistency primitives) are hidden from the user. Implementations of PSER exist for both PARMv8[7] and Px86[5], and these implementations have been proven to be *sound*, guaranteeing correct PARMv8-to-PSER and Px86-to-PSER compilation.

The PSER model and its corresponding implementations were milestones in the literature of formal transactional persistency models for a number of reasons. Not only were they the first of their kind, but also existing language-level persistency models[32, 33] that had been proposed in the literature up to this point had been inaccessible to all except the most experienced persistent programmers[5]. However, one considerable disadvantage of PSER is the assumption of serialisability, which causes performance slowdown due to its strong guarantees (see Section 2.4.3). This motivates the need for a formal transactional persistency model with weaker consistency guarantees to be developed, which is the premise of this project.

3 | Preliminaries

In order to define a formal declarative model for durable snapshot isolation, it would be useful to establish a general way of representing the persistency semantics of concurrent programs that are executed in the context of non-volatile memory. For this purpose, we start by presenting a **general declarative framework** that describes the persistency semantics of such programs. This framework was first presented by Raadet al. in conjunction with the PARMv8 and PSER models[7]; both of these models are instantiations of the framework. Later, in Chapter 4, we will introduce the DSI model as an instance of this general framework as well. In this chapter, we begin by describing the programming language and semantics used for these programs in Section 3.1, and go on to present the components that are required for representing the persistency guarantees of such programs in Section 3.2.

3.1 Programming Language

The general form of the simple concurrent programming language used throughout this thesis is given in Fig. 3.1. We assume a finite set REG of of registers (local variables); a finite set VAL of values; a finite set TID $\subseteq \mathbb{N}$ of thread identifiers; and a standard interpreted language for expressions, EXP, containing at least registers and values. We use v as a metavariable for values, τ for thread identifiers and e for expressions. The sequential fragment of the language is given by the COM grammar and includes *primitive commands* (c) which are model-specific and determined by the underlying memory model, as well as the standard constructs of expressions, local variable assignment, conditionals and loops. We model a multi-threaded program P as a function that maps each thread to its corresponding sequential program. We write $\mathsf{P} = \mathsf{C}_1 || \ldots || \mathsf{C}_n$ when $dom(\mathsf{P}) = \{\tau_1 \ldots \tau_n\}$ and $\mathsf{P}(\tau_i) = \mathsf{C}_i$. For better readability, we shorten the way we write commands from this language (e.g. $\mathsf{a} := \mathsf{C}$ for **let** $\mathsf{a}{:=}\mathsf{C}$ **in** a, and $\mathsf{C}_1; \mathsf{C}_2$ for **let** $\mathsf{a}{:=}\mathsf{C}_1$ **in** C_2, where a is a fresh local variable).

Basic domains	**Expressions and sequential commands**
$\mathsf{a} \in$ REG Registers	EXP $\ni \mathsf{e} ::= v \mid \mathsf{a} \mid \mathsf{e} + \mathsf{e} \mid \ldots$
$v \in$ VAL Values	PCOM $\ni \mathsf{c} ::= \ldots$
$\tau \in$ TID Thread IDs	COM $\ni \mathsf{C} ::= \mathsf{e} \mid \mathsf{c} \mid$ **let** $\mathsf{a}{:=}\mathsf{C}$ **in** C
Programs	$\mid$ **if** (e) **then** C **else** $\mathsf{C} \mid$ **while** (e) **do** C
$\mathsf{P} \in$ PROG $\triangleq$ TID $\xrightarrow{\text{fin}}$ COM	

Figure 3.1: A simple concurrent programming language

3.2 Persistency Semantics

3.2.1 Locations & Cache Lines

In order to capture the possibility of hybrid memory hierarchies (where both volatile and non-volatile RAMs are combined together), we assume a set of *memory locations*, LOC. We typically use $x, y, \ldots$ as meta-variables for locations. We define the set of *cache lines* as CL $\triangleq \mathcal{P}$(LOC), and use $X, Y, \ldots$ as meta-variables for cache lines.

3.2.2 Labels and Events

In the literature for declarative models, the traces of shared memory accesses that are generated by a program are often represented as a set of execution graphs. Here, the graph nodes denote execution *events* and the graph edges capture the relations on events. Each event corresponds to the execution of a primitive command ($\mathtt{c} \in$ PCOM in Fig. 3.1) and is a tuple of the form $\mathtt{e} = \langle n, \tau, l \rangle$, where $n \in \mathbb{N}$ is the *event identifier* uniquely identifying $\mathtt{e}$; $\tau \in$ TID is the thread identifier of the executing thread; and $l \in$ LAB is the event *label*, described below.

We assume a set of labels, LAB, that is associated with primitive commands that are speciifc to the underlying memory model. This general set of labels can in turn be divided into different *types* of label. We assume a set of *read labels*, RLAB, and a set of *write labels*, WLAB, such that RLAB $\cup$ WLAB $\subseteq$ LAB. These read and write labels are associated with (primitive) read and write commands, respectively. The functions $\mathtt{loc}$, $\mathtt{val_r}$ and $\mathtt{val_w}$ respectively project the location, the read value and the written value of a label, where applicable. Additionally, we assume a set of *durable labels*, DLAB $\subseteq$ LAB, associated with durable commands, which can intuitively be described as those whose effects may be observed when recovering from a crash. For example, the label of $x := v$ is durable because the effects of this instruction *could* be observed upon recovery if the write of the value v has persisted on the location x prior to the crash. We note that durability does not reflect whether the effects of the associated command *do persist*, but rather whether they *could persist*. For example, a read instruction $a := x$ has no durable effects and so its label is not durable. Since write instructions are durable, we require that the labels for these instructions be included in DLAB : WLAB $\subseteq$ DLAB.

Parameter 1. (Labels). Assume a set of *labels* LAB, a set of *read labels* RLAB $\subseteq$ LAB, and a set of *write labels* WLAB $\subseteq$ LAB. Assume functions $\mathtt{loc} :$ LAB $\rightharpoonup$ LOC, $\mathtt{val_r} :$ RLAB $\to$ VAL, and $\mathtt{val_w} :$ WLAB $\to$ VAL. Assume a set of *durable labels*, DLAB $\subseteq$ LAB, such that WLAB $\subseteq$ DLAB.

Definition 1. (Events). An *event* is a tuple $\langle n, \tau, l \rangle$ where $n \in \mathbb{N}$ is an event identifier, $\tau \in$ TID is a thread identifier, and $l \in$ LAB is an event label.

We usually use a, b and e to range over events. The functions $\mathtt{tid}$ and $\mathtt{lab}$ respectively project the thread identifier and the label of an event. We lift the label functions $\mathtt{loc}$, $\mathtt{val_r}$ and $\mathtt{val_w}$ to events, and for a given event e, we write e.g. $\mathtt{loc}(e)$ for $\mathtt{loc}(\mathtt{lab}(e))$.

3.2.3 Notation

Given a relation r on a set A, we write $\mathsf{r}^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closures of r, respectively. We write r^{-1} for the inverse of r; $\mathsf{r}|_A$ for $\mathsf{r} \cap (A \times A)$; $[A]$ for the identity relation on A (i.e. $\{(a,a) \mid a \in A\}$); $\mathsf{irreflexive}(r)$ for $\nexists a.(a,a) \in r$; and $\mathsf{acyclic}(r)$ for $\mathsf{irreflexive}(r^+)$. We write $\mathsf{r}_1; \mathsf{r}_2$ for the relational composition of r_1 and r_2 (i.e. $\{(a,b) \mid \exists c.(a,c) \in \mathsf{r}_1 \wedge (c,b) \in \mathsf{r}_2\}$). When A is a set of events, we write A_x for $\{a \in A \mid \mathtt{loc}(a) = x\}$, and write A_X for $\{a \in A \mid \mathtt{loc}(a) \in X\}$. Similarly, we write r_x for $\mathsf{r} \cap (A_x \times A_x)$, and write r_X for $\mathsf{r} \cap (A_X \times A_X)$.

3.2.4 Executions

As explained in Section 3.2.2, it is common practice to represent the traces of shared memory accesses generated by a program a set of *execution graphs*. Each execution G is a graph comprising: (i) a set of events denoting the graph nodes, and (ii) a number of relations on these events denoting the graph edges.

Definition 2. (Executions). An *execution*, $G \in$ EXEC, is a tuple $(E, I, P, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{nvo})$, where:

- E denotes a set of *events*. The set of *read* events in E is: $R \triangleq \{e \in E \mid \exists x, v.\ \mathtt{lab}(e) \in$ RLAB$\}$; the set of *write* events, W, and *durable* events, D, are defined analogously.
- I is a set of *initialisation* events, comprising a single write event $w_x \in W$ for each location x.
- P is a set of *persisted events* such that $I \subseteq P \subseteq D$.
- $\mathsf{po} \subseteq E \times E$ denotes the *'program-order' relation*, defined as a disjoint union of strict total orders, each ordering the events of one thread, with $I \times (E \setminus I) \subseteq \mathsf{po}$.

- $\mathsf{rf} \subseteq W \times R$ denotes the *'reads-from' relation* between write and read events of the same location with matching values; i.e. $(a,b) \in \mathsf{rf} \implies \mathtt{loc}(a) = \mathtt{loc}(b) \wedge \mathtt{val_w}(a) = \mathtt{val_r}(b)$. Moreover, rf is total and functional on its range; i.e. every read is related to exactly one write.
- $\mathsf{mo} \subseteq E \times E$ is the *'modification-order' relation*, denoting a strict partial order defined as the disjoint union of relations $\{\mathsf{mo}_x\}_{x \in \text{LOC}}$, such that each mo_x is a strict total order on W_x and $I_x \times (W_x \setminus I_x) \subseteq \mathsf{mo}_x$.
- $\mathsf{nvo} \subseteq D \times D$ is the *'non-volatile order' relation*, defined as a strict total order on D, such that $I \times (D \setminus I) \subseteq \mathsf{nvo}$ and $dom(\mathsf{nvo}; [P]) \subseteq P$.

In the context of an execution graph G, where the "$G.$" prefix is often used to make this context explicit, the persisted events P include the durable events whose effects have reached persistent memory (and from this, we have $P \subseteq D$). Hence, persisted events include initialisation writes on persistent locations, so $I \subseteq P$. The 'modification-order' relation mo constrains the visible order of stores to memory between threads. Similarly, the 'non-volatile-order' relation nvo constrains the visible order in which stores are *committed* to persistent memory. Therefore, we require that the persisted events in P are *downward-closed* with respect to nvo. This can be expressed as $dom(\mathsf{nvo}; [P]) \subseteq P$. In other words, if we let $e_1 \dots e_n$ denote the enumeration of D in the order prescribed by nvo (and we know this to be a complete enumeration since nvo is total on D), then P being downward-closed with respect to nvo means that there exists some i satisfying $0 \leq i \leq n$ such that $e_1, \dots, e_i \in P$ and $e_{i+1}, \dots, e_n \in D \setminus P$. Lastly, we define the *'reads-before'* relation as: $\mathsf{rb} \triangleq (\mathsf{rf}^{-1}; \mathsf{mo}) \setminus [E]$, relating a read r to all writes w that are mo-after the write that r reads from. This is also known as the anti-dependency relation in the transactional literature[34].

We note that in this initial stage, executions are unrestricted in that there are not many constraints on rf, mo and nvo. These restrictions are determined by the set of memory-model-specific *consistent* executions. In the upcoming sections, we define execution consistency for the Px86 model and the novel DSI model.

3.2.5 Chains

Considering only *complete* executions (i.e. those that do not crash) is too simplistic to capture the potential crashing behaviour of executions in the presence of persistent memory. Therefore, we model an *execution chain* $\mathcal{C}$ as a sequence $G_1, \dots, G_n$, where (1) G_1 describes the intiial era between the start of execution up to the first crash; (2) for all $i \in \{2, \dots, n-1\}$, G_i denotes the i^{th} execution era, recovering from the $(i-1)^{\text{th}}$ crash; and (3) G_n desribes the final execution era terminating successfully.

Definition 3. (Chains.) A *chain* $\mathcal{C}$ is a sequence $G_1, \dots, G_n$ of executions such that for $1 \leq i < n$ and $G_i = (E_i, I_i, P_i, \mathsf{po}_i, \mathsf{rf}_i, \mathsf{mo}_i, \mathsf{nvo}_i)$:

- $\forall x \in \text{LOC}. \exists w. w \in I_1 \wedge \mathtt{loc}(w) = x \wedge \mathtt{val_w}(w) = 0$;
- $\forall x \in \text{LOC}. \exists w, e. w \in I_{i+1} \wedge \mathtt{loc}(w) = x \wedge e = \max\left(\mathsf{nvo}_i|_{P_i \cap (W_x \cup U_x)}\right) \wedge \mathtt{val_w}(w) = \mathtt{val_w}(e)$
- $P_n = D_n$

Given a memory model $\mathcal{M}$, a chain $\mathcal{C} = G_1, \dots, G_n$ is *$\mathcal{M}$-valid* if each G_i is *$\mathcal{M}$-consistent*.

The first axiom ensures that all locations are initialised to 0 in the first execution era. The second axiom ensures that in each successive ($(i+1)^{\text{st}}$) era, every location gets initialised with a value that has been *persisted* by a write or update in the previous (i^{th}) era to the same location. This write/update is *maximal* with respect to the non-volatile order for that era, nvo_i, which intuitively means that the final value that was able to persist for a given location x in the previous (i^{th}) era is the value that x should be initialised with in the following ($(i+1)^{\text{st}}$) era.

3.2.6 Persistent Programs & Recovery Mechanisms

In the persistent setting, each program P is associated with a *recovery mechanism* that describes the code to be executed upon recovery from a crash. Therefore, we model a persistent program, $\mathbb{P} \in \text{PPROG}$, as a pair $\langle \mathsf{P}, \mathtt{rec} \rangle$, where $\mathsf{P} \in \text{PROG}$ denotes the original program and $\mathtt{rec}$ denotes its recovery mechanism. A naive recovery mechanism for P may choose to always *restart* the execution of P from the start upon the occurrence of a crash, essentially erasing any progress made up to that point. Conversely, a more sophisticated recovery mechanism may *resume* the execution of P upon recovery based on the progress made before the crash occurred. We therefore model a recovery mechanism as a function $\mathtt{rec} : \text{PROG} \times \text{EXEC} \to \text{PROG}$, where $\mathtt{rec}(\mathsf{P}, G)$ describes the recovery mechanism associated with the

program P when the state of the memory upon recovery is that which has been obtained after some execution G. On an intuitive level, G corresponds to the previous execution era, and so it is possible to establish what the state of memory upon such a recovery should be by inspecting G.

3.2.7 Semantics of Persistent Programs

Typically, the *semantics* of a program P is defined as a set of consistent executions associated with P and is defined using induction on the structure of P. Analogously, in the persistent setting, the semantics of a *persistent* program $\mathbb{P}$ is defined as a set of valid chains associated with $\mathbb{P}$. Specifically, a persistent program $\mathbb{P}$ is associated with a valid chain $\mathcal{C} = G_1, \ldots, G_n$ if the following hold: (1) G_1 is a partial execution of P; (2) G_i is a partial execution of $\texttt{rec}(\mathsf{P}, G_{i-1})$ for all i such that $2 \leq i \leq n-1$; and (3) G_n is an execution of $\texttt{rec}(\mathsf{P}, G_{n-1})$. We note that the executions of all eras except the last (G_n) are *partial* since they have failed to run to completion due to a crash occurring.

4 | The Declarative DSI Model

Having described the general framework of Raad et al. [7] for declarative persistency models in Chapter 3, it is now possible to introduce a high-level instantiation of this framework which allows transactions to be used as a concurrency control mechanism in the context of non-volatile memory. We call this the *declarative DSI model*, and in particular, this model consists of axioms which ensure that the *snapshot isolation* transactional consistency model is upheld in the *persistent* setting (i.e. **d**urable **s**napshot **i**solation is maintained). We begin this chapter by instantiating the general framework of Chapter 3 with model-specific details for DSI, and then present the axioms with an intuitive explanation of how they ensure the consistency and persistency guarantees of durable snapshot isolation.

4.1 DSI Programming Language

The DSI programming language is an extension of the general concurrent programming language provided in Fig. 3.1. The *DSI primitive commands* comprise load (read) and store (write) operations, and are given by the following grammar:

$$\text{PCOM}_{\text{DSI}} \ni \mathtt{c} ::= \textbf{load}(x) \mid \textbf{store}(x, \mathtt{e})$$

We assume for the sake of simplicity that the sequential programs in each thread are composed of a sequence of DSI *transactions*. The set of *DSI programs*, $\text{PROG}_{\text{DSI}} \subseteq \text{PROG}$, are defined as follows, where C denotes a sequential program as defined in Fig. 3.1, instantiated with the DSI primitive commands introduced above:

$$\text{PROG}_{\text{DSI}} \ni \mathsf{P} ::= \text{TID} \xrightarrow{\text{fin}} \text{COM}_{\text{DSI}} \qquad \text{COM}_{\text{DSI}} \ni \mathsf{C}_{\text{DSI}} ::= [\mathsf{C}] \mid \mathsf{C}_{\text{DSI}}; \mathsf{C}_{\text{DSI}}$$

4.2 DSI Labels and Events

In order to distinguish between events belonging to different transactions, we assume a finite set of transaction identifiers, TXID, ranged over by ξ. A DSI label is then either: (1) a *read* label $\mathtt{R}(x, v, \xi)$, for reading value v from location x in transaction ξ; or (2) a *write* label $\mathtt{W}(x, v, \xi)$ for writing v to x in ξ; or (3) a *begin* label $\mathtt{B}(\xi)$ that marks the beginning of transaction ξ; or (4) an *end* label $\mathtt{E}(\xi)$ that marks the end of ξ. A *DSI event* is an event (Definition 1) with a DSI label. *DSI read and write events* are defined as events with read and write labels, respectively. *DSI durable events* coincide with DSI write events. The $\mathtt{tx}$ function returns the transaction identifier of a DSI label. We lift the $\mathtt{tx}$ function to events and write $\mathtt{tx}(e)$ for $\mathtt{tx}(\mathtt{lab}(e))$.

Given an execution G, the *'same-transaction' relation*, $\mathsf{st} \in G.E \times G.E$, is the equivalence relation given by $\mathsf{st} \triangleq \{(a, b) \in G.E \times G.E | \mathtt{tx}(a) = \mathtt{tx}(b)\}$. Given a relation r on $G.E$, r_{T} signifies *lifting* r to equivalence classes: in other words, $r_{\mathsf{T}} \triangleq \mathsf{st}; (r \setminus \mathsf{st}); \mathsf{st}$, and $[a]_{\mathsf{st}}$ denotes the st class that contains a. In other words, $[a]_{\mathsf{st}} \triangleq \{e \in G.E | (a, e) \in \mathsf{st}\}$. We note that a class that does not contain an end event denotes a transaction whose execution was rendered *incomplete* due to the occurrence of a crash. The events of *complete transactions* in G are denoted by $G.T$; i.e. those events whose associated end events are in G. We define $G.T \triangleq \{a \in G.E | \exists e \in [a]_{\mathsf{st}}.\mathtt{lab}(e) = \mathtt{E}(-)\}$.

4.3 DSI Executions

An execution G is a *DSI execution* if the following hold: (1) $G.E$ are DSI events; (2) each transaction class contains *exactly one* begin event; (3) each transaction class contains *at most one* end event; (4) each begin event is the first event in po within its transaction; (5) each end event is the last event in po within its transaction; and (6) only the final (i.e. maximal in terms of po) transaction in each thread may be incomplete (due to a crash).

4.4 DSI-Consistency

We now formalise what it means for an execution to be DSI-consistent.

Definition 4. (DSI-consistency). A DSI execution $(E, I, P, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{nvo})$ is *DSI-consistent* iff:

- $(\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb}) \cap G.\mathsf{st} \subseteq \mathsf{po}$ ($\textsc{si}_1$)
- $\mathsf{hb}_{\mathsf{SI}} \triangleq (\mathsf{po}_{\mathsf{T}} \cup \mathsf{rf}_{\mathsf{T}} \cup \mathsf{mo}_{\mathsf{T}} \cup \mathsf{rb}_{\mathsf{SI}})^+$ is irreflexive, where: ($\textsc{si}_2$)
  - $\mathsf{rb}_{\mathsf{SI}} \triangleq [R_{\mathsf{E}}]; \mathsf{rb}_{\mathsf{T}}; [W]$, and
  - $R_{\mathsf{E}} \triangleq \{r \mid \exists w.(w, r) \in \mathsf{rf}_{\mathsf{E}}\}$
- $\mathsf{hb}_{\mathsf{SI}}|_D \subseteq \mathsf{nvo}$ (DSI-NVO)
- $dom([D]; G.\mathsf{st}; [P]) \subseteq P \subseteq G.T$ (DSI-ATOMIC)

Consistency axioms. The ($\textsc{si}_1$) and ($\textsc{si}_2$) axioms are those of the snapshot isolation consistency model for transactions[29], adapted to the declarative framework we are using here. Intuitively, ($\textsc{si}_1$) ensures the consistency of transactions internally, while ($\textsc{si}_2$) provides synchronisation guarantees between distinct transactions.

More concretely, ($\textsc{si}_1$) expresses that events within the same transaction which are related by rf, mo or rb must respect the program order, po. This makes sense intuitively as well: if a write event with label $\mathtt{W}(x, v, \xi)$ and a read event with label $\mathtt{R}(x, v, \xi)$ are related by rf within the same transaction ξ, then the read should only be able to access the value v at location x if an earlier write has written v to x. Similarly in the second case, mo should follow the program order for writes in the same transaction since it represents the order in which memory stores to the same location take place. For rb, we note that if a read event r with label $\mathtt{R}(x, v, \xi)$ is rb-related to some write event w with label $\mathtt{W}(x, v', \xi)$ for some x, v, v' then ($\textsc{si}_1$) enforces that the write *must* be po-after the read if they are in the same transaction. Intuitively, this means that r reads v from some separate write that is mo-before (and, as a result of this axiom, po-before) w, and so r "reads before" w as required.

Next, the ($\textsc{si}_2$) axiom expresses that events in different transactions follow some order, which is defined using the arbitrary extension of a partial "happens-before" relation $\mathsf{hb}_{\mathsf{SI}}$. This captures synchronisation that is either due to the program order (po_{T}), or that is enforced by the implementation ($\mathsf{rf}_{\mathsf{T}} \cup \mathsf{mo}_{\mathsf{T}} \cup \mathsf{rb}_{\mathsf{SI}}$). We note that events *within* the same transaction are unordered, since the implementation can execute them in a different order. For example, the mechanism of taking a snapshot before locally executing a transaction means that a transaction's external reads get executed before its writes.

We examine the synchronisation that is enforced by the implementation by taking each component of ($\mathsf{rf}_{\mathsf{T}} \cup \mathsf{mo}_{\mathsf{T}} \cup \mathsf{rb}_{\mathsf{SI}}$) in turn. Firstly, rf_{T} corresponds to transactional synchronisation that occurs as a result of *causality*. In other words, it corresponds to synchronisation that occurs in cases where a transaction ξ_2 observes an effect of some earlier transaction ξ_1. Including rf_{T} in this union makes sure that ξ_2 can only read from ξ_1 if it has observed its *entire* effect; this means transactions exhibit "all-or-nothing" behaviour with respect to reads, so they cannot mix-and-match reads from different transactions. For example, consider three transactions ξ_0, ξ_1 and ξ_2 that take place in $\mathsf{hb}_{\mathsf{SI}}$ order. If ξ_1 writes to locations x and y, then ξ_2 is not allowed to read the value of x from ξ_1 but read the value of y from ξ_0, as this would mean the effect of the write to x from ξ_1 had been observed but the write to y from ξ_1 had not been. In other words, since ξ_1 is the latest transaction (in $\mathsf{hb}_{\mathsf{SI}}$ order) to write to x and y, its effects cannot be *partially* observed by some $\mathsf{hb}_{\mathsf{SI}}$-later transaction (such as ξ_2).

Next, mo_T corresponds to transactional synchronisation that takes place due to *write-write conflicts*, which we recall is the single type of conflict under snapshot isolation that can prevent concurrently executing transactions from committing successfully. The inclusion of mo_T in this union enforces the write-write-conflict-freedom of transactions under snapshot isolation, since if some transactions ξ_1 and ξ_2 both write to some location x via some write events w_1, w_2 such that $(w_1, w_2) \in \mathsf{mo}$, then ξ_1 is earlier in hb_SI-order than ξ_2, and so ξ_1 must commit before ξ_2 in order to make its *entire* effect visible to ξ_2. Intuitively, there is no way for a write-write conflict to take place if, for any two transactions that both write to some location x, one of them is required to commit successfully and make its entire effect visible before the other.

Finally, we consider rb_SI. We first note that R_E denotes the *external* transactional reads, which corresponds to the set of reads whose values were written in a different transaction. This also means that R_E represents the set of reads in a transaction that get their values from when the snapshot for that transaction was taken. In contrast, internal reads (which are those that read values that have been written by the *same* transaction) can only happen after the snapshot has been taken. This is due to the fact that writes within the same transaction update the array that contains values from when the snapshot was taken, and so since these array updates clearly take place after the snapshot is taken, the internal reads from these very writes must also take place after the snapshot is taken. Now, assume there is an rb_T edge between two transactions ξ_1 and ξ_2. This means that there must be at least one rb edge between some read r in ξ_1 and some write w in ξ_2. More formally, we express this as $(r, w) \in \mathsf{rb}$, and so from the definition of rb this means that there exists some write w' such that $(w', r) \in \mathsf{rf}$ and $(w', w) \in \mathsf{mo}$. If r reads internally, this means that since r is reading from w', then w' is in ξ_1. Therefore, ξ_1 and ξ_2 are write-conflicting transactions (i.e. there is an mo_T edge between them), and so the mo_T case from above applies and hb_SI-orders ξ_1 before ξ_2. Meanwhile, if r reads externally (i.e. $r \in R_\mathrm{E}$), then w' is not in ξ_1. From our high-level understanding of the SI implementation, we know that the snapshot that is taken before the local execution of ξ_1 must take place before all of the writes in ξ_2 are committed. This is because otherwise, r would read the value written by w since they have the same location and because w is mo-after w' on the same location. In other words, we know that all of the reads in the snapshot of ξ_1 (i.e. all external reads in ξ_1) must happen before all writes in ξ_2. This corresponds directly to the definition of rb_SI, and so ξ_1 is hb_SI-before ξ_2.

Persistency axioms. The (DSI-NVO) and (DSI-ATOMIC) axioms describe the persistency semantics of DSI, and are adapted from the persistency axioms of the PSER model[7]. The (DSI-NVO) axiom ensures that transactional writes (which coincide with durable events for DSI) persist to memory in hb_SI order. This allows inter-transactional synchronisation orderings to be preserved across crashes. For example, if there is an rf_T edge between some transactions ξ_1 and ξ_2, then there exists some read event in ξ_2 which reads from a write event in ξ_1. Under (DSI-NVO), ξ_1 must persist before ξ_2, and so in the event of a crash and the subsequent recovery that takes place, we note that the scenario where ξ_2 has persisted but ξ_1 (the transaction it has read from) has not persisted can never occur.

Meanwhile, (DSI-ATOMIC) ensures that transactions persist in an *atomic* way. From the axiom, it is clear that the following hold: only *complete* transactions are allowed to persist (due to $P \subseteq G.T$), and either all of a transaction's durable events persist or none of them persist (due to $dom([D]; G.\mathsf{st}; [P]) \subseteq P$). This notion of *persist atomicity* is inspired by the write atomicity of database transactions, which requires that either all or none of the writes in a transaction must commit.

5 | Implementing DSI in Px86

Now that we have formalised the semantics of DSI declaratively in Chapter 4, we show that our model is *feasible* by developing a reference implementation and recovery mechanism for durable snapshot isolation written over **Px86** (the persistency model for Intel-x86, first described in Section 2.3.1). Given a high-level DSI transaction, the reference implementation locally executes the transaction under the snapshot isolation model while accounting for the required persistency guarantees as well. In particular, the implementation is accompanied by a *recovery mechanism*, which gets called every time a program recovers from a crash.

Together, the reference implementation and recovery mechanism serve as the underlying implementation for a high-level persistent transactional library. The aim of this library is to facilitate persistent programming for programmers of all abilities, including those who may not be familiar with the low-level hardware details of mainstream NVM hardware architectures. Concretely, a user only needs to interact with simple, high-level transactions consisting of reads and writes, and the underlying Px86 implementation takes care of the consistency and persistency guarantees of durable snapshot isolation. In addition, this implementation provides a more intuitive explanation of how durable snapshot isolation works in comparison with the axiomatic model of the previous chapter, as the shorthand notation we use makes the implementation readable to anyone familiar with a high-level programming language and a few Px86-specific persistency primitives and commands (which we outline in due course).

In this chapter, we start by providing a brief overview of the persistency primitives and commands available for Intel-x86. We then go on to introduce the reference DSI implementation and recovery mechanism in Px86, with intuitive explanations of how these ensure the consistency and persistency guarantees of durable snapshot isolation included alongside them.

5.1 Persistency Primitives for Intel-x86

As first mentioned in Section 2.3.1, with the ascent of NVM technologies in recent years, mainstream architectures have begun to provide support for persistent programming at the hardware level. The ubiquitous Intel-x86 architecture is no exception; indeed, Intel has been one of the forerunners in developing NVM capabilities within its own processors. In particular, Intel-x86 processors provide a number of *persistency primitives* to afford users more control over when pending writes in their programs are persisted. In this section, we provide an overview of the most important Px86 persistence instructions and informally describe their behaviours (as given in [5, 35]), in preparation for the use of these primitives in the reference DSI implementation.

5.1.1 Flushes

Two of the explicit persist instructions available as part of Intel-x86 are known as **flush** and $\textbf{flush}_{\text{opt}}$. Given a location x, **flush** x and $\textbf{flush}_{\text{opt}}$ x both *asynchronously* persist all pending writes on all locations that are in the cache line of x. In other words, when x is in cache line X ($x \in X$), executing one of these instructions on x results in the pending writes on all locations $x' \in X$ being persisted to memory. These instructions vary in strength, which in this context means they vary with regards to the number of constraints they enforce upon instruction reordering. This indicates that the weaker a persist instruction is (i.e. the fewer constraints it imposes upon the reordering of itself and other instructions), the better its performance, at the expense of inducing more complicated program behaviours. In terms of the program order, both **flush** x and $\textbf{flush}_{\text{opt}}$ x are ordered with respect to *earlier* writes that are on the same

$x := 1;$ **flush**$_{\text{opt}}$ $x';$ **FAA**$(y, 1);$ (a)	$x := 1;$ **flush**$_{\text{opt}}$ $x';$ $y := 1;$ (b)	$x := 1;$ **flush** $x';$ $y := 1;$ (c)	$x := 1;$ **flush**$_{\text{opt}}$ $x';$ **sfence**; $y := 1;$ (d)
rec: $y{=}1 \Rightarrow x{=}1$	rec: $x, y \in \{0, 1\}$	rec: $y{=}1 \Rightarrow x{=}1$	rec: $y{=}1 \Rightarrow x{=}1$

Figure 5.1: Examples of Px86 programs that include persistency primitives and the possible values of x, y upon recovery. In all examples, x, y are locations in (persistent) memory, $x, x' \in X$ (x and x' are in cache line X), $y \notin X$ and initially $x{=}y{=}0$.

cache line as x, as well as all earlier *and* later atomic read-modify-write (RMW) instructions. However, **flush** instructions are *stronger* than **flush**$_{\text{opt}}$ instructions, in that **flush** x is not just ordered with respect to earlier writes on the cache line of x, but is actually ordered with respect to *all* writes, earlier and later in program order, *regardless* of the cache line.

To gain an intuition for the difference between the **flush** and **flush**$_{\text{opt}}$ instructions, consider the example programs given in Fig. 5.1. In Fig. 5.1a, due to the ordering constraints we have defined for **flush**$_{\text{opt}}$ above, the **flush**$_{\text{opt}}$ x' instruction cannot be reordered with respect to either the $x := 1$ write or the **FAA**$(y, 1)$ instruction (representing an atomic "fetch-and-add"). This is because the former instruction is an earlier write on x, which we assume to be on the same cache line as x', while the latter instruction is an RMW instruction. Therefore, the writes for this example program are required to persist in the program order, so if y is 1 upon recovery after a crash, the write on x must have already persisted. We observe different behaviour in Fig. 5.1b, however, where the **FAA** instruction has been replaced with a write on y. Since y is not on the same cache line as x', the **flush**$_{\text{opt}}$ instruction may be reordered with respect to the write to y, and in particular, there is no guarantee that $x := 1$ persists before $y := 1$, despite the **flush**$_{\text{opt}}$ instruction that separates the two. The reordering constraints on this example program are strengthened in Fig. 5.1c, where the **flush**$_{\text{opt}}$ x' instruction has been replaced with a **flush** x'. Due to the stronger nature of **flush** , the writes on x and y are required to persist in the program order, since **flush** is ordered with respect to all writes in a program, regardless of their associated cache line or where they are in the program.

5.1.2 Fences

Intel-x86 also provides *fence* instructions for giving programmers more control over the order in which writes on *different* locations are persisted to memory. These include memory fences, written as **mfence**, and store fences, written as **sfence**. These instructions are standalone in that they take no memory locations as parameters; instead, they enforce ordering constraints on memory operations on *all* locations. These fences can also be characterised in terms of the strength of the reordering constraints they impose upon the program in question, as done above for flushes. These ordering constraints for **mfence** and **sfence** are as follows: memory fences cannot be reordered with respect to *any* memory instruction, whereas store fences are allowed to be reordered with respect to only reads. We note that this means **flush** and **flush**$_{\text{opt}}$ are not allowed to be reordered with respect to either type of fence. In our reference implementation, we use **sfence** instructions throughout but no **mfence** instructions, and so we focus on the former from this point onwards.

We again look to a concrete example in order to understand the effect of including **sfence** instructions in Px86 programs. Fig. 5.1d shows a modified version of the program given in Fig. 5.1b, simply augmented with an **sfence** between the **flush**$_{\text{opt}}$ instruction and the write on y. The introduction of this **sfence** ensures that none of the instructions can be reordered, since they are all memory instructions (a write, a flush, a fence and a write respectively) and none of them are reads. Therefore, the behaviour upon recovery from a crash changes, with y guaranteed to persist after x.

5.2 A Reference DSI Implementation in Px86

We now introduce an implementation for DSI in Px86, along with a corresponding recovery mechanism for restarting execution in the event of a crash. These are shown in Fig. 5.2. As writes often need to be

```
0.  [T]_{DSI→Px86} ≜
1.    τ := getTID(); ξ := getTxID();
2.    log[τ] :=_fo ξ; sfence;
3.    LS := ∅;
4.    for(x ∈ RS ∪ WS) {
5.      r-lock(x);
6.      l[x] :=_fo ξ;
7.    }
8.    sfence;
9.    snapshot(RS);
10.   for(x ∈ RS \ WS) r-unlock(x);
11.   for(x ∈ WS) {
12.     if (promote(x)) LS.add(x);
13.     else {
14.       for(x ∈ LS) w-unlock(x);
15.       for(x ∈ WS \ LS) r-unlock(x);
16.       goto line 3; } }
17.   w :=_fo new-array();
18.   (|T|); sfence;
19.   ws[ξ] :=_fo w;
20.   for(x ∈ WS) {
21.     a := w[x];
22.     x :=_fo a;
23.   } sfence;
24.   for(x ∈ WS) w-unlock(x);
```

```
snapshot(RS) ≜ for(x ∈ RS) s[x] := x
(|a := x|) ≜ a := s[x]
(|x := a|) ≜ s[x] := a; w[x] :=_fo a;
(|T1; T2|) ≜ (|T1|); (|T2|)
...and so on...
```

```
25. recover(P) ≜
26.   for(x ∈ dom(l))
27.     w-unlock(x);
28.   for(τ ∈ dom(P)) {
29.     ξ := log[τ];
30.     w := ws[ξ];
31.     if (w = ⊥)
32.       P'[τ] := sub(P[τ], ξ);
33.     else
34.       P'[τ] := sub(P[τ], ξ+1);
35.       if (!committed(w, ξ)) {
36.         for(x ∈ dom(w))
37.           x :=_fo w[x];
38.       }
39.     }
40.   }
41.   sfence;
42.   run(P');
```

where committed(w, ξ) $\overset{\text{def}}{\iff} dom(w) = \emptyset \vee \exists x, \xi'. x \in dom(w) \wedge \xi' \neq \xi \wedge l[x] = \xi'$

Figure 5.2: DSI implementation of transaction [C] in Px86 (left and top right), where grey code ensures deadlock avoidance and highlighted code ensures persistency; DSI recovery implementation (bottom right).

persisted explicitly, $x :=_{\texttt{fo}} \texttt{e}$ is used as a shorthand for $x := \texttt{e}$; $\textbf{flush}_{\text{opt}}\ x$. We later show in Chapter 6 that this implementation for DSI is *sound*, demonstrating correct Px86-to-DSI implementation.

5.2.1 MRSW Locks

The DSI implementation given in Fig. 5.2 uses locks as a synchronisation mechanism for managing concurrent accesses to shared data. In this case, the snapshot isolation model permits concurrent transactions to read from the same memory location *simultaneously*, and so we assume the existence of a **multiple-readers-single-writer (MRSW) lock library** in order to manage these concurrent accesses. Intuitively, these locks are acquired in reader mode so that a snapshot can be taken of the memory locations that have been accessed by a given transaction. The locks associated with locations that are written to during this transaction then get promoted to writer mode; this is done in order to enforce an ordering on transactions with *write-write conflicts*, where a transaction T is said to have such a conflict if another *committed* transaction T′ has written to a location also written to by T, since T was able to record its snapshot.

In terms of the implementation of this lock library, we assume that each location x is associated with an MRSW lock that can be acquired by either (i) multiple threads reading from x simultaneously; or (ii) one single thread writing to x. A reader lock on x is acquired by calling r-lock(x) and released by calling r-unlock(x). Similarly, a write lock on x is acquired by calling w-lock(x) and released by calling w-unlock(x). In addition, a reader lock on x can be *promoted* to a write lock by calling promote(x). A call to promote(x) returns a boolean that represents either (i) successful promotion (true); or (ii) failed promotion in the case of another reader lock on x being promoted simultaneously. This is done since two distinct reader locks on x may attempt to promote their locks at the same time as each other, and so promotion is done on a 'first-come-first-served' basis in order to mitigate for this. A call to promote(x)

returns true for the calling reader on x once all other readers have released their locks on x; this allows the calling reader to safely assume that it has exclusive ownership of this lock, now in write mode.

It is straightforward to implement this lock library in Px86; indeed, it has been implemented as part of the technical appendix for the paper in which Px86 was first introduced[5]. For the sake of brevity and simplicity, we do not provide the implementations of these lock library functions, and assume they behave the way described above. These are formalised later in Chapter 6.

5.2.2 Snapshot Isolation in DSI Implementation

Given a transaction $[\mathsf{T}]$, the DSI implementation of $[\mathsf{T}]$ in Px86, written as $[\mathsf{T}]_{\text{DSI}\to\text{Px86}}$, is given in Fig. 5.2. If we ignore the code in grey (lines 3, 12-16) and the highlighted code, $[\mathsf{T}]_{\text{DSI}\to\text{Px86}}$ describes an implementation of $[\mathsf{T}]$ that follows the SI model, using MRSW locks . We call the set of locations read from by T the *read set* of T, denoted as $\mathtt{RS}$. Similarly, we call the set of locations written to by T the *write set* of T, denoted as $\mathtt{WS}$. In theory, an SI implementation of transaction T would be expected to do the following: (i) obtain a *snapshot* of the locations read from by T; (ii) lock the locations written to by T; (iii) execute the transaction T *locally*; and (iv) unlock the locations that were written to by T once it has finished executing locally.

We note that our implementation is *prescient*, meaning it requires *a priori* knowledge about the locations read from and written to by the given transaction. Concretely, this means that $\mathtt{RS}$ and $\mathtt{WS}$ are assumed to be known in advance of the transaction being executed locally. Furthermore, in order to reduce *lock contention* (i.e. to reduce the chance of threads attempting to acquire locks that have already been acquired by some other thread), we acquire *all* necessary locks (i.e. those in $\mathtt{RS} \cup \mathtt{WS}$) in read mode, and promote the locks in $\mathtt{WS}$ just before our local execution of T.

Our implementation of durable snapshot isolation therefore proceeds as follows. We start by acquiring read locks on all locations accessed by T (lines 4-5). We then obtain a snapshot via a call to $\mathtt{snapshot(RS)}$ (line 9), where the values of the locations in $\mathtt{RS}$ are recorded in a *local*, volatile array s. Next, we release the reader locks on $\mathtt{RS}$ (line 10) before promoting the locks on $\mathtt{WS}$ (lines 11-12). Once lock promotion is complete, the write log w is initialised to an empty array (line 17) and T is executed locally (line 18). This local execution is denoted by $(|\mathsf{T}|)$ and is described in more detail shortly. After T has finished executing locally, the writes recorded in w are committed to memory (lines 20-22), and finally all remaining write locks get released (line 24).

The local execution $(|\mathsf{T}|)$ is given in the top right of Fig. 5.2, and is obtained from T as follows. For each read operation $\mathtt{a := } x$, instead of reading directly from a location x, we read locally from the snapshot s at location x to retrieve the value x had when the snapshot was taken (prior to the local execution of T). For each write operation $x \mathtt{ := a}$, we take an analogous approach to the one described above, in that we do not update the value of x directly and instead update the snapshot s with the new value for x. We also log this value in $w[x]$ at each write operation. The local execution of the remaining inductive cases (e.g. composition: $\mathsf{T}_1; \mathsf{T}_2$) is defined by simple induction on the structure of commands (e.g. $(|\mathsf{T}_1; \mathsf{T}_2|) \triangleq (|\mathsf{T}_1|); (|\mathsf{T}_2|)$).

5.2.3 Deadlock Avoidance

We recall that a call to $\mathtt{promote}(x)$ by a reader r returns false if another reader r' is going through the process of promoting a lock on x. Whenever this occurs, r is required to release its reader lock on x so that r' can successfully promote its lock on x, thus avoiding deadlocks. In order to enforce this behaviour, the provided implementation contains a deadlock avoidance mechanism (lines 12-16) which works as follows. We record a set $\mathtt{LS}$ (initialised to $\emptyset$ on line 3) of the locks on the write set $\mathtt{WS}$ which have already been promoted successfully at a given point during the execution of the implementation. When promoting a lock on some location x succeeds (line 12), $\mathtt{LS}$ gets extended with x. However, when x fails to be promoted, we recall from Section 5.2.1 that this means that another reader lock on x is in the process of being promoted. This behaviour is undesirable as no more than one thread should have ownership over the given lock in write mode. In order to prevent such scenarios, the failure of a lock being promoted results in all of the locks promoted thus far (i.e. in $\mathtt{LS}$) as well as the locks that were yet to be promoted (i.e. in $\mathtt{WS} \setminus \mathtt{LS}$) being released (lines 14 and 15), and the transaction gets restarted (line 16).

We note that it is sufficient for the implementation to be restarted from line 3 rather than line 1. This is because the commands on lines 1 and 2, which use the identifiers for the current thread (τ) and the current transaction (ξ) to update the log array (explained later in Section 5.2.4), only need to execute once at the beginning. In other words, even after the transaction has been restarted, these values are allowed to remain the same and are guaranteed to have persisted, and so the transaction is restarted from a later point, preventing the same code from executing (and in the case of the write on line 2, persisting) again unnecessarily.

5.2.4 Persistency of DSI Implementation

We recall that given a DSI program $\mathsf{P} \in \textsc{Prog}_{\text{DSI}}$, the sequential program in every thread $\tau_i \in dom(P)$ is made up of a sequence of transactions: $\mathsf{P}(\tau_i) = [\mathsf{T}_i^1]; \ldots; [\mathsf{T}_i^n]$. Therefore, we can represent $\mathsf{P}(\tau_i)$ as an array T_i such that $T_i[j] = [\mathsf{T}_i^j]$. In other words, the j^{th} element of the array of transactions for thread i is the j^{th} executed transaction, written $[\mathsf{T}_i^j]$. In addition, we assume that the context of each thread τ_i is such that a call to `getTID()` returns i and a call to `getTxID()` returns j when $[\mathsf{T}_i^j]$ is being executed. Both of these calls are on line 1 of our implementation. A program P is executed via a call to `run`(P).

In order to ensure *correct* recovery, this implementation needs to be able to account for the possibility of a crash at any point during the execution of the program. To achieve this, we record the *metadata* for tracking the progress of each thread during its execution, and store this information in the log, ws and l arrays as follows. For a given thread τ, $log[\tau]$ records the last executed transaction for that thread. In addition, for a given transaction ξ, $ws[\xi]$ records the *effect* of ξ; concretely, the effect of ξ is the information stored in the write log w once the transaction has finished locally executing. Furthermore, for each location x, $l[x]$ records the last transaction that acquired a lock on x. In the provided DSI implementation, the last executed transaction ξ for thread τ gets recorded in the log array (line 2); then, ξ gets recorded in $l[x]$ for all locations x that are accessed (i.e. read from or written to) by transaction T (line 6, inside for loop); and finally, the effect of the transaction is stored in $ws[\xi]$ by assigning w to it (line 19).

Ensuring that the transactional metadata is persisted *fully* (i.e. not partially) in the case of a crash is crucial for correct recovery. For this purpose, the **sfence** instruction is used throughout the implementation, which we recall from Section 5.1.2 can only be reordered with respect to reads. **sfence** instructions provide improved performance over the stronger **mfence** while also constraining the writes to log, ws and l to persist *before* the execution of any writes that may be after the given **sfence** instruction. One such use is on line 18, where the **sfence** ensures that all pending writes, including those of w which occurred during the local execution of T (line 18), are persisted before the l array is updated on line 19.

We note that **sfence** instructions can be expensive as they cause the program to stall until all earlier writes on *all locations* have persisted. Hence, we use the minimum possible number of sfence instructions needed to ensure correct recovery, which is *four* in the case of this implementation. The purpose of each of these **sfence** instructions is as follows. The first **sfence** (line 2) ensures that the storing of the last executed transaction's identifier ξ in $log[\tau]$ (also on line 2) persists. The second **sfence** (line 8) ensures that the storing of ξ as the last transaction to acquire a lock on all locations in the read set and write set of ξ (line 6) persists. The purpose of the third **sfence** has been outlined in the previous paragraph, and the fourth **sfence** makes sure the update of l (line 19) as well as all of the writes from w being assigned to their corresponding true locations in memory (lines 20-22) are persisted. Together, these ensure that in the case of a crash, the persisted progress of each thread τ could be at most one step behind the persisted metadata in log, ws and l.

5.2.5 DSI Recovery Implementation

After a crash has taken place, a program P is restored by calling the *recovery mechanism*, `recover`(P), which is given in the bottom right of Fig. 5.2. At a high level, it works as follows: (1) all locks are released in order to avoid deadlocks (lines 26-27) ; (2) the progress of the threads that were executing before the crash took place is restored via the generation of a new program P′ (lines 28-41); and (3) this new program P′ is eventually run (line 42).

We recall from Section 5.2.4 that due to the placement of **sfence** instructions throughout the implementation, the progress of each thread is *at most* one step behind its persisted metadata. Therefore, it is

sufficient to check whether the *last* recorded transaction for a given thread τ has persisted in order to calculate where to resume the program from. From the `recover` function for a given program P, let us take the last transaction executed by thread τ to be ξ (line 29), and take w to be a local variable which contains the effect of ξ. Then, one of two cases must hold, which are as follows. Either (i) the effect of ξ in the ws array has *not* persisted before the crash and so $\mathsf{w} = \bot$; or (ii) the effect of ξ in ws *has* persisted (meaning the crash took place after line 19), so $\mathsf{w} \neq \bot$ and the effect has been stored in w. In case (i), since the effect of the transaction in ws did not persist, there is no evidence in memory indicating whether the transaction was able to start executing or not, and so the program $\mathsf{P}[\tau]$ has to be resumed from the beginning of transaction ξ (line 32). Meanwhile, in case (ii), $\mathsf{P}[\tau]$ is advanced to $\xi + 1$ since the effect of the previous transaction in ws was able to persist before the crash occurred. Here, $\mathtt{sub}(\mathsf{P}[\tau], n)$ denotes the subarray of $\mathsf{P}[\tau]$ starting from n.

We note that in case (ii) (when the effect of ξ in ws was able to persist before the crash), the *full* effect of ξ may not have been able to persist to memory. This means that even if $ws[\xi]$ was able to persist, the writes of the values contained in $ws[\xi]$ may not have been able to persist to their true locations in the write set of ξ (i.e. lines 20-22 were not persisted). If the effect of ξ has only partially persisted, it needs to be committed "manually"; this is ascertained using the `committed` predicate, defined at the bottom of Fig. 5.2.

The `committed` predicate is intended to hold if the entire effect of ξ has persisted, which could be as a result of either of the two disjuncts of `committed` being true. On the one hand, $\mathtt{committed}(\mathsf{w}, \xi)$ could hold as a result of its first disjunct when $dom(\mathsf{w}) = \emptyset$, which means that $ws[\xi]$ was able to persist before the crash but it is empty; therefore, the transaction ξ must be *read-only*, since the w array in the implementation must have remained an empty array even after the transaction was locally executed on line 18. Alternatively, this predicate could hold as a result of its second disjunct being true: i.e. $dom(\mathsf{w}) \neq \emptyset$ and $x \in dom(\mathsf{w})$ for some location x. In this case, we can assume that the effect of ξ has fully persisted if some distinct transaction $\xi' \neq \xi$ is the *last* transaction to have acquired a lock on x (i.e. $l[x] = \xi'$). This can be explained in the following way: we know that the crash must have taken place after line 19 of the execution of the implementation for ξ, since the effect of ξ in ws has fully persisted. Therefore, lines 4-7 must have also been executed as they are earlier in program order and there are **sfence** instructions in between which enforce constraints on which instructions can be reordered. In particular, we know that the lock on x must have been acquired by ξ at some point during the execution of lines 4-7 due to this, as well as from the knowledge that $x \in dom(\mathsf{w})$. In addition, since ξ' is the last transaction to have acquired the x lock, we know that ξ must have released its lock on x at some point (i.e. line 24 must have been executed during the execution of the implementation for ξ). Therefore, ξ must have fully committed and persisted. Furthermore, the **sfence** instruction on line 41 makes sure that the committed writes (lines 36-37) have persisted before any writes in the restarted program P'.

6 | Soundness of Implementation

Next, we show that the DSI implementation given in Section 5.2 is *sound*. In other words, given a Px86-consistent execution graph G of the implementation, we show that it is possible to construct an *abstract* graph G' from it which satisfies the declarative DSI axioms given in Chapter 4. The purpose of this is to show that the reference implementation given in Chapter 5 provides the guarantees of the DSI declarative model from Chapter 4, thus demonstrating *correct* Px86-to-DSI implementation.

We begin this chapter by formalising the MRSW lock library that was introduced in Section 5.2.1. Then, we prove that the DSI reference implementation is sound. Note that we use two declarative models for Px86 from the literature[5, 8] extensively throughout the proofs in this chapter; the details of these models are provided in Appendix A.

6.1 MRSW Lock Library

We now begin to formalise the concepts discussed Section 5.2.1, where the MRSW lock library was first introduced as part of the reference implementation for DSI.

We assume that each call to the `r-lock` function on a given location x yields an event with the label $\texttt{RL}(x)$. We analogously define the labels $\texttt{RU}(x)$, $\texttt{WL}(x)$, $\texttt{WU}(x)$ and $\texttt{PL}(x)$ to correspond to the events yielded by the function calls `r-unlock(x)`, `w-lock(x)`, `w-unlock(x)` and `promote(x)` respectively. For a set of events E, we define the set of *read unlock* events in E as follows: $RU \triangleq \{e \in E \mid \exists x.\, \texttt{lab}(e) = \texttt{RU}(x)\}$. The sets of *read lock* (RL), *write unlock* (WU), *write lock* (WL) and *promote lock* (PL) events are defined analogously. We further assume that these MRSW locks ensure certain synchronisation guarantees, which we discuss later on in (WWSYNC) and (RWSYNC).

6.2 Execution Trace

Taking an arbitrary program P and a Px86-valid execution chain $\mathcal{C} = G_1; ...; G_n$ of P with $G_i = (E_i, I_i, P_i, \mathsf{po}_i, \mathsf{rf}_i, \mathsf{mo}_i, \mathsf{nvo}_i)$, we observe that when P comprises k threads, the trace of each execution era is formed of two stages:

i) The trace of the *initialisation* stage by the master thread τ_0, which performs either initialisation or recovery (via `start()` or `recover()`, respectively). This is what precedes the call to `run(P)`.

ii) The trace of each of the constituent program threads $\tau_1 ... \tau_k$. This stage follows the initialisation stage in po order, and is conditional upon no crashes occurring during the initialisation stage.

We note that since the execution is Px86-valid, due to the placement of **sfence** instructions throughout the implementation, we know that the set of persisted events for a given thread τ_j in execution era i (namely, P_i) contains an approximate *prefix* in po order of the trace of thread τ_j. This can be explained more concretely: for every constituent thread $\tau_j \in \tau_1, ..., \tau_k = dom(\mathsf{P})$, there exist $p_1^j \dots p_n^j, q_1^j \dots q_n^j, w_1^j \dots w_n^j$ such that:

(1) $\mathsf{P}[\tau_j] = \mathsf{T}_j^0; \dots; \mathsf{T}_j^{p_1^j}; \mathsf{T}_j^{p_1^j+1}; \dots; \mathsf{T}_j^{p_2^j}; \dots; \mathsf{T}_j^{p_{n-1}^j+1}; \dots; \mathsf{T}_j^{p_n^j}$, where:

- Each T_j^k denotes the kth transaction of thread τ_j

- $\mathsf{T}_j^{p_i^j}$ denotes the last transaction of thread τ_j that was logged in the ith era. In other words, the ith crash occurred when $log[\tau_j] = \xi_j^{p_i^j}$. Thus, p_i^j represents the *identifier* of this last transaction logged in the ith era for the thread τ_j

(2) At the *beginning* of each execution era $i \in \{1 \dots n\}$, for all j, the program executed by thread τ_j is that of $\texttt{sub}(\mathsf{P}[\tau_j], q_j^i)$, such that:

$$q_j^i = \begin{cases} p_j^{i-1} + 1, & \text{if } w_i^j \neq \bot \\ p_j^{i-1}, & \text{if } w_i^j = \bot \end{cases}$$

Intuitively, q_j^i represents the identifier of the transaction from which the program gets resumed during recovery, and w_j^i represents the write set of this transaction. This corresponds to lines 31 to 34 of the implementation's recovery mechanism, where the program to be executed ($\texttt{sub}(\mathsf{P}[\tau_j], q_j^i)$) is stored in P'. P' is later executed as part of the call to $\texttt{run}(\mathsf{P}')$ on line 42.

(3) In each execution era $i \in \{1 \dots n\}$, the **trace** of the program is of the form $\theta^p_{\texttt{init}(i)} \xrightarrow{\mathsf{po}} (\theta_{(i,1)} || \dots || \theta_{(i,k)})$. These components are defined as follows:

- $\theta^p_{\texttt{init}(i)}$ denotes a prefix of $\theta_{\texttt{init}(i)}$. This prefix may potentially be a full one (i.e. $\theta^p_{\texttt{init}(i)} = \theta_{\texttt{init}(i)}$)
- $\theta_{\texttt{init}(i)}$ denotes the execution of either the initialisation or recovery mechanism, as defined in detail shortly
- $\theta_{(i,j)}$ denotes the trace of the jth constituent thread $\tau_j \in dom(\mathsf{P})$ and is defined as:

$$\theta_{(i,j)} \triangleq \begin{cases} \theta_i(\xi_j^{q_j^i}) \xrightarrow{\mathsf{po}} \dots \xrightarrow{\mathsf{po}} \theta_i^p(\xi_j^{p_j^i}) & \text{if } \theta^p_{\texttt{init}(i)} = \theta_{\texttt{init}(i)} \\ \emptyset & \text{otherwise} \end{cases}$$

This means that whenever $\theta^p_{\texttt{init}(i)} = \theta_{\texttt{init}(i)}$, no crash occurred during the execution of $\theta^p_{\texttt{init}(i)}$, and so it is a full prefix of $\theta_{\texttt{init}(i)}$. In this case, $\theta_{(i,j)}$ denotes the (q_j^i)th to (p_j^i)th transactions of thread τ_j, with $\theta_i(\xi)$ defined shortly. On the other hand, if $\theta^p_{\texttt{init}(i)} \neq \theta_{\texttt{init}(i)}$, then there was a crash during the execution of $\theta^p_{\texttt{init}(i)}$ and so we proceed no further, setting $\theta_{(i,j)}$ to be empty. That is, the trace $\theta_{(i,j)}$ of the jth constituent thread τ_j is non-empty only when the initialisation/recovery mechanism has executed to completion.

In addition, due to the placement of the **sfence** instructions within the implementation, before crashing and proceeding to the next era, *all* durable events in $\theta_i(\xi_j^{q_j^i}) \xrightarrow{\mathsf{po}} \dots \xrightarrow{\mathsf{po}} \theta_i^p(\xi_j^{p_j^i - 1})$ are guaranteed to have persisted, and a *subset* S of the durable events in $\theta_i(\xi_j^{p_j^i})$ are guaranteed to have persisted. We note that it is possible to have $S = \theta_i(\xi_j^{p_j^i})$, in which case all of the durable events in $\theta_i(\xi_j^{p_j^i})$ have persisted. We write T^i for the set of all transactions *executed* in the ith era.

We now define the trace of the arbitrary program P introduced at the beginning of this section, in terms of the two aforementioned stages for the trace: $\theta_{\texttt{init}(i)}$, the trace of the initialisation stage by the master thread τ_0, and $\theta_{(i,j)}$, the trace of the jth constituent thread τ_j.

Trace of the initialisation/recovery. Recall that the initialisation stage is where the master thread τ_0 performs initialisation or recovery. In the very first era, when $i = 1$, we have $\theta_{\texttt{init}(1)} = \emptyset$, since there is no previous era to perform recovery on. When $i > 1$, $\theta_{\texttt{init}(i)}$ is of the following form:

$$Us \xrightarrow{\mathsf{po}} C(i,1) \xrightarrow{\mathsf{po}} W(i,1) \xrightarrow{\mathsf{po}} \dots \xrightarrow{\mathsf{po}} C(i,k) \xrightarrow{\mathsf{po}} W(i,k) \xrightarrow{\mathsf{po}} sf$$

This corresponds to the recovery mechanism $\texttt{recover}(\mathsf{P})$ defined on lines 25 to 42 of Fig. 5.2. This sequence is defined as follows, with the correspondence between these components and the lines of the reference implementation that they relate to provided *in brackets* after each component:

- Us denotes the sequence of events releasing all locks (Lines 26-27)

- For all $i \in \{1 \dots n\}$ and $j \in \{1 \dots k\}$:

$$C(i+1,j) \triangleq rlog_{(i+1,j)} \overset{\mathsf{po}}{\to} rwmap_{(i+1,j)} \overset{\mathsf{po}}{\to} wp'_{(i+1,j)}$$

  where $\mathtt{lab}(rlog_{(i+1,j)}) = \mathtt{R}(log[\tau_j], \xi_j^{p_j^i})$, $\mathtt{lab}(rwmap_{(i+1,j)}) = \mathtt{R}(ws[\xi_j^{p_j^i}], w_j^{i+1})$ and $\mathtt{lab}(wp'_{(i+1,j)}) = \mathtt{W}(\mathsf{P}'[\tau_j], q_j^{i+1})$ (Lines 29-34. The explicit inclusion of $\{C(i,1), \dots C(i,k)\}$ in the sequence means all threads $\tau_j \in \mathsf{P}$ are covered, and so line 28 is also included)

- When $dom(w_j^{i+1}) = x_1 \dots x_m$:

$$W(i+1,j) \triangleq W_1^{(i+1,j)} \overset{\mathsf{po}}{\to} \dots \overset{\mathsf{po}}{\to} W_m^{(i+1,j)}$$

  and for all $t \in \{1 \dots m\}$:

$$W_t^{(i+1,j)} \triangleq \begin{cases} wx_t^{(i+1,j)} \overset{\mathsf{po}}{\to} fox_t^{(i+1,j)} & \text{if } q_j^{i+1} = p_j^i + 1 \text{ and } \neg\mathtt{committed}(w_j^{i+1}, \xi_j^{p_j^i}) \\ \emptyset & \text{otherwise} \end{cases}$$

  such that $\mathtt{lab}(wx_t^{(i+1,j)}) = \mathtt{W}(x_t, w_j^{i+1}[x_t])$ and $\mathtt{lab}(fox_t^{(i+1,j)}) = \mathtt{FO}(x_t)$. (Lines 35-38. The explicit inclusion of $\{W(i,1), \dots W(i,k)\}$ in the sequence means all threads $\tau_j \in \mathsf{P}$ are covered, and so line 28 is also included)

- sf denotes the **sfence** instruction, such that $\mathtt{lab}(sf) = \mathtt{SF}$ (Line 41)

Additionally, we write T_{rec}^i for the set of all transactions *recovered* in the ith era:

$$T_{rec}^i \triangleq \{\xi \mid \exists j.\ \mathtt{lab}(rlog_{(i,j)}) = \mathtt{R}(log[\tau_j], \xi) \wedge W(i,j) \neq \emptyset\}$$

Trace of the constituent threads. We now move on to the second stage of the trace for a given execution era: the trace of a constituent thread τ_j. When ξ is a transaction of thread τ with body T, then the trace $\theta_i(\xi)$ is of the following form:

$$log_1 \overset{\mathsf{po}}{\to} logfo_1 \overset{\mathsf{po}}{\to} sf_1 \overset{\mathsf{po}}{\to} Fs \overset{\mathsf{po}}{\to} Rs \overset{\mathsf{po}}{\to} sf_2 \overset{\mathsf{po}}{\to} Ss \overset{\mathsf{po}}{\to} RUs \overset{\mathsf{po}}{\to} PLs \overset{\mathsf{po}}{\to} we \overset{\mathsf{po}}{\to} wefo \overset{\mathsf{po}}{\to} Ts$$
$$\overset{\mathsf{po}}{\to} sf_3 \overset{\mathsf{po}}{\to} log_2 \overset{\mathsf{po}}{\to} logfo_2 \overset{\mathsf{po}}{\to} Ws \overset{\mathsf{po}}{\to} sf_4 \overset{\mathsf{po}}{\to} WUs$$

This corresponds to the implementation for $[\mathsf{T}]_{\mathsf{DSI}\to\mathsf{Px86}}$ provided on lines 1 to 25 of Fig. 5.2. First we note that sf_1, sf_2, sf_3 and sf_4 represent the **sfence** instructions on lines 2, 8, 18 and 23 of the implementation respectively (i.e. $\mathtt{lab}(sf_i) = \mathtt{SF}$ for $i \in [1,4]$). Next, we define the remaining components of this sequence below, with the correspondence between components and their relevant lines in the reference implementation again provided in brackets after each definition:

- $\mathtt{lab}(log_1) = \mathtt{W}(log[\tau], \xi)$ and $\mathtt{lab}(logfo_1) = \mathtt{FO}(log[\tau])$ (Line 2)

- Fs denotes the sequence of events failing to obtain the required locks for the transaction to be executed. In other words, it represents those iterations that do not succeed in promoting the writer locks. Anything considered after this point in the trace corresponds to a successful iteration. (Lines 3-16, with special emphasis on Line 16 since this is where iterations that have failed to promote locks return to Line 3 and try again)

- Rs denotes the sequence of events acquiring the reader locks on all locations accessed, and updating the l array accordingly. Rs is of the following form:

$$rl_{x_1} \overset{\mathsf{po}}{\to} wlog_{x_1} \overset{\mathsf{po}}{\to} \dots \overset{\mathsf{po}}{\to} rl_{x_i} \overset{\mathsf{po}}{\to} wlog_{x_i}$$

  where for all $n \in \{1 \dots i\}$, $wlog_{x_n} \triangleq \mathtt{W}(l[x_n], \xi) \overset{\mathsf{po}}{\to} \mathtt{FO}(l[x_n])$. (Lines 4-7)

- Ss denotes the sequence of events capturing a snapshot of the read set, and is of the form $S_{x_1} \overset{\text{po}}{\to} \ldots \overset{\text{po}}{\to} S_{x_i}$ where for all $n \in \{1 \ldots i\}$:

$$S_{x_n} \triangleq \begin{cases} r_{x_n} \overset{\text{po}}{\to} \widehat{ws}_{x_n} & \text{if } x_n \in \mathtt{RS}_\xi \\ \emptyset & \text{otherwise} \end{cases}$$

  with $r_{x_n} \triangleq \mathtt{R}(x_n, v_n)$ and $\widehat{ws}_{x_n} \triangleq \mathtt{W}(s[x_n], v_n)$ (Line 9 and `snapshot` definition in top right)

- RUs denotes the sequence of events releasing the reader locks (when the given location is only in the read set) and is of the form $RU_{x_1} \overset{\text{po}}{\to} \ldots \overset{\text{po}}{\to} RU_{x_i}$, where for all $n \in \{1 \ldots i\}$:

$$RU_{x_n} \triangleq \begin{cases} ru_{x_n} & \text{if } x_n \notin \mathtt{WS}_\xi \\ \emptyset & \text{otherwise} \end{cases}$$

  where $\mathtt{lab}(ru_{x_n}) = \mathtt{RU}(x_n)$ (Line 10).

- PLs denotes the sequence of events promoting the reader locks to writer locks (when the given location is in the write set) and is of the form $PL_{x_1} \overset{\text{po}}{\to} \ldots \overset{\text{po}}{\to} PL_{x_n}$, where for all $n \in \{1 \ldots i\}$:

$$PL_{x_n} \triangleq \begin{cases} pl_{x_n} & \text{if } x_n \in \mathtt{WS}_\xi \\ \emptyset & \text{otherwise} \end{cases}$$

  where $\mathtt{lab}(pl_{x_n}) = \mathtt{PL}(x_n)$
  (Lines 11-12)

- $\mathtt{lab}(we) = \mathtt{W}(w, [])$ and $\mathtt{lab}(wefo) = \mathtt{FO}(w)$ (Line 17)

- Ts denotes the sequence of events corresponding to the execution of $(|\mathsf{T}|)$ and is of the form $t_1 \overset{\text{po}}{\to} \ldots \overset{\text{po}}{\to} t_k$, where for $m \in \{1 \ldots k\}$, each t_m is either of the form $rd(x_m, v_m)$ or $wr(x_m, v_m)$, with:

$$rd(x_m, v_m) \triangleq rs_{x_m}$$
$$wr(x_m, v_m) \triangleq ws_{x_m} \overset{\text{po}}{\to} lw_{x_m} \overset{\text{po}}{\to} lfo_{x_m}$$

  where $rs_{x_n} \triangleq \mathtt{R}(s[x_n], v_n)$, $ws_{x_n} \triangleq \mathtt{W}(s[x_n], v_n)$, $lw_{x_n} \triangleq \mathtt{W}(w[x_n], v_n)$ and $lfo_{x_n} \triangleq \mathtt{FO}(w[x_n])$ (Line 18 and transaction-related definitions in top right).

- $\mathtt{lab}(log_2) = \mathtt{W}(ws[\xi], w)$ and $\mathtt{lab}(logfo_2) = \mathtt{FO}(ws[\xi])$ (Line 19)

- Ws denotes the sequence of events committing the writes of $(|\mathsf{T}|)$ and is of the form $c_{x_1} \overset{\text{po}}{\to} \ldots \overset{\text{po}}{\to} c_{x_i}$, where for all $n \in \{1 \ldots i\}$:

$$c_{x_n} \triangleq \begin{cases} lr_{x_n} \overset{\text{po}}{\to} w_{x_n} \overset{\text{po}}{\to} fo_{x_n} & \text{if } x_n \in \mathtt{WS}_\xi \\ \emptyset & \text{otherwise} \end{cases}$$

  and $\mathtt{lab}(lr_{x_n}) = \mathtt{R}(w[x_n], v_n)$, $\mathtt{lab}(w_{x_n}) = \mathtt{W}(x_n, v_n)$ and $\mathtt{lab}(fo_{x_n}) = \mathtt{FO}(x_n)$ (Lines 20-23).

- WUs denotes the sequence of events releasing the writer locks and is of the form $WU_{x_1} \overset{\text{po}}{\to} \ldots \overset{\text{po}}{\to} WU_{x_i}$, where for all $n \in \{1 \ldots i\}$:

$$WU_{x_n} \triangleq \begin{cases} wu_{x_n} & \text{if } x_n \in \mathtt{WS}_\xi \\ \emptyset & \text{otherwise} \end{cases}$$

  where $\mathtt{lab}(wu_{x_n}) = \mathtt{WU}(x_n)$ (Line 24).

We note that for all $\xi_1, \xi_2 \in T^i_{rec}$, if $\xi_1 \neq \xi_2$, then $\mathtt{WS}_{\xi_1} \cap \mathtt{WS}_{\xi_2} = \emptyset$. As such, for every location x, there is at most one write to x during the execution of the recovery $\theta_{\mathtt{init}(i)}$. We denote this write by rec_x.

For each location $x \in \mathtt{WS}_\xi$, let fw_x denote the maximal write (in po order) logging a write for x in $w[x]$. In other words, when $Ts = t_1 \overset{\text{po}}{\to} \ldots \overset{\text{po}}{\to} t_m$, let $fw_x = \mathtt{wmax}(x, [t_1 \ldots t_m])$, where:

$$\texttt{wmax}(x, [\,]) \quad \text{undefined}$$
$$\texttt{wmax}(x, L.[t]) \triangleq \begin{cases} t.lw_x & \text{if } t = wr(x, -) \\ \texttt{wmax}(x, L) & \text{otherwise} \end{cases}$$

Note that if an execution is Px86-consistent, then $(fw_{x_n}, lr_{x_n}) \in \mathsf{rf}$, for all $x_n \in \texttt{WS}_\xi$.

6.3 Implementation Soundness

In order to establish that the DSI implementation is sound, it suffices to show that given a Px86-consistent execution graph G of the implementation, it is possible to construct a corresponding DSI-consistent execution graph G' with the same outcome. In era i, given a transaction ξ of thread τ_j with code T, $\texttt{RS}_\xi \cup \texttt{WS}_\xi = \{x_1 \dots x_i\}$ and trace $\theta_i(\xi)$ as given above, with $\theta_i(\xi).Ts = t_1 \overset{\mathsf{po}}{\to} \dots \overset{\mathsf{po}}{\to} t_k$, we construct the corresponding DSI execution trace $\theta'_i(\xi)$ as follows:

$$\theta'_i(\xi) \triangleq t'_1 \overset{\mathsf{po}}{\to} \dots \overset{\mathsf{po}}{\to} t'_k$$

where for all $m \in \{1 \dots k\}$:

$$\texttt{lab}(t'_m) = \texttt{R}(x_m, v_m, \xi) \quad \text{when} \quad t_m = rd(x_m, v_m) \tag{6.1}$$
$$\texttt{lab}(t'_m) = \texttt{W}(x_m, v_m, \xi) \quad \text{when} \quad t_m = wr(x_m, v_m) \tag{6.2}$$

In the first case, the identifier of t'_m is that of $\theta_i(\xi).rs_{x_m}$; and in the second case, the identifier of t'_m is that of $\theta_i(\xi).lw_{x_m}$. We are therefore able to define a function, $\texttt{imp(.)}$, mapping each DSI event t'_m to its corresponding Px86 event: $\theta_i(\xi).rs_{x_m}$ when $\texttt{lab}(t'_m) = \texttt{R}(x_m, v_m, \xi)$, or $\theta_i(\xi).lw_{x_m}$ when $\texttt{lab}(t'_m) = \texttt{W}(x_m, v_m, \xi)$.

It is now possible to demonstrate the soundness of the DSI implementation. Given a Px86-consistent execution graph G_i of the implementation in the i^{th} era, we construct a DSI execution graph G'_i as follows and demonstrate that it is DSI-consistent:

- $G'_i.E = G'_i.I \cup Rec \cup Run$, with $Rec \triangleq \bigcup_{\xi \in T^i_{rec}} \theta'_{i-1}(\xi).E$, $\theta'_0(-) = \emptyset$ and $Run \triangleq \bigcup_{\xi \in T^i} \theta'_i(\xi).E$
- $G'_i.I = \left\{ \texttt{W}(x, v, 0) \;\middle|\; \begin{array}{l} x \in \textsc{Loc} \wedge (i = 0 \implies v = 0) \wedge \\ (i > 0 \implies \exists e \in \max(\mathsf{nvo}_i|_{G'_{i-1}.P \cap W_x}).\, \texttt{val}_\texttt{w}(e) = v \end{array} \right\}$
- $G'_i.P = G'_i.I \cup PRec \cup \bigcup_{\xi \in T^i} \mathrm{p}(\xi)$, where:

$$PRec \triangleq \begin{cases} Rec & \text{if } \theta^p_{\texttt{init}(i)} = \theta_{\texttt{init}(i)} \wedge \theta_{\texttt{init}(i)}.E \cap D \subseteq G_i.P \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathrm{p}(\xi) \triangleq \begin{cases} \theta'_i(\xi).E & \text{if } \theta_i(\xi).E \cap D \subseteq G_i.P \\ \emptyset & \text{otherwise} \end{cases}$$

- $G'_i.\mathsf{po} = G'_i.I \times (G'_i.E \setminus G'_i.I) \cup (Rec \times Run)_i \cup G.\mathsf{po}|_{G'.E}$
- $G'_i.\mathsf{rf} = \bigcup_{\xi \in T^i} \mathrm{RF}_\xi \cup \bigcup_{\xi \in T^i_{rec}} \mathrm{RF}'_\xi$

- $G'_i.\mathsf{mo} = \Big((G'_i.I \times ((G'_i.E \setminus G'_i.I) \cap W))\Big)_{loc}$
  $\cup\, ((Rec \cap W) \times (Run \cap W))_{loc}$
  $\cup\, \{(e,e') | \exists x.\, e,e' \in (W_x \cap Rec) \wedge \mathtt{tx}(e) = \mathtt{tx}(e') \wedge (e,e') \in G'_i.\mathsf{po}\}$
  $\cup\, \mathsf{MO}$
- $G'_i.\mathsf{nvo} = G'_i.I \times ((G'_i.E \setminus G'_i.I) \cap D)$
  $\cup\, \{(e,e') | e,e' \in (G'_i \cap D) \wedge \mathtt{id}(e) < \mathtt{id}(e')\}$
  $\cup\, ((Rec \cap D) \times (Run \cap D))$
  $\cup\, \{(e,e') | e,e' \in (G'_i.D \cap Rec) \wedge (e,e') \in (G'_i.\mathsf{st} \cap \mathsf{po})\}$
  $\cup\, \{(e,e') | e,e' \in (G'_i.Rec \cap D) \wedge (e,e') \notin G'_i.\mathsf{st} \wedge (e,e') \in G'_i.\mathsf{hb}_{\mathsf{SI}}\}$
  $\cup\, \{(e,e') | e,e' \in (G'_i.Rec \cap D) \wedge (e,e') \notin (G'_i.\mathsf{st} \cup \mathsf{hb}_{\mathsf{SI}}) \wedge \mathtt{tx}(e) \prec \mathtt{tx}(e')\}$
  $\cup\, \mathsf{NVO}$

where $\prec$ denotes a strict total order on transaction identifiers (e.g. natural number ordering), and:

$$\mathrm{RF}_\xi \triangleq \left\{ (t_k{}', t_j{}') \;\middle|\; \begin{array}{l} \exists x, v.\ \mathtt{lab}(t_j{}') = \mathtt{R}(x,v) \wedge \mathtt{lab}(t_k{}') = \mathtt{W}(x,v) \\ \wedge (\theta_i(\xi).t_k.ws_x, \theta_i(\xi).t_j.rs_x) \in G.\mathsf{rf} \end{array} \right\}$$

$$\cup \left\{ (t_k{}', t_j{}') \;\middle|\; \begin{array}{l} \exists x, v.\ \mathtt{lab}(t_j{}') = \mathtt{R}(x,v) \wedge \mathtt{lab}(t_k{}') = \mathtt{W}(x,v) \wedge \xi \neq \xi' \wedge t_k = \theta_i(\xi').fw_x \\ \wedge (\theta_i(\xi).\widehat{ws}_x, \theta_i(\xi).t_j.rs_x) \in G.\mathsf{rf} \wedge (\theta_i(\xi').w_x, \theta_i(\xi).r_x) \in G.\mathsf{rf} \end{array} \right\}$$

$$\mathrm{RF}'_\xi \triangleq \{(w,r) | \mathtt{tx}(r) = \xi \wedge (w,r) \in G'_{i-1}.\mathsf{rf} \wedge \mathtt{tx}(w) = \mathtt{tx}(r)\}$$

$$\cup \left\{ (w_0, r) \;\middle|\; \begin{array}{l} \mathtt{tx}(r) = \xi \wedge \mathtt{loc}(r) = \mathtt{loc}(w_0) \wedge w_0 \in G'_i.I \\ \wedge \exists w.\, (w,r) \in G'_{i-1}.\mathsf{rf} \wedge \mathtt{tx}(w) \neq \mathtt{tx}(r) \end{array} \right\}$$

$$\mathrm{MO} \triangleq \{(t'_k, t'_j) | \mathtt{tx}(t'_k) = \mathtt{tx}(t'_j) \wedge \mathtt{loc}(t'_k) = \mathtt{loc}(t'_j) \wedge t'_k, t'_j \in W \wedge (t_k, t_j) \in G.\mathsf{po}\}$$

$$\cup \left\{ (t_k{}', t_j{}') \;\middle|\; \begin{array}{l} t'_k, t'_j \in W \wedge \exists x, \xi_j, \xi_k.\, \mathtt{loc}(t'_k) = \mathtt{loc}(t'_j) = x \\ \wedge t_k \in \theta_i(\xi_k) \wedge t_j \in \theta_i(\xi_j) \wedge (\theta_i(\xi_k).w_x, \theta_i(\xi_j).w_x) \in G.\mathsf{mo} \end{array} \right\}$$

$$\mathrm{NVO} \triangleq \{(t'_k, t'_j) | \mathtt{tx}(t'_k) = \mathtt{tx}(t'_j) \wedge t'_k, t'_j \in D \wedge (t_k, t_j) \in G.\mathsf{po}\}$$

$$\cup \left\{ (t_k{}', t_j{}') \;\middle|\; \begin{array}{l} t'_k, t'_j \in W \wedge \exists x, y, \xi_j, \xi_k.\, \mathtt{loc}(t'_k) = x \wedge \mathtt{loc}(t'_j) = y \\ \wedge t_k \in \theta_i(\xi_k) \wedge t_j \in \theta_i(\xi_j) \wedge (\theta_i(\xi_k).w_x, \theta_i(\xi_j).w_y) \in G.\mathsf{nvo} \end{array} \right\}$$

We now proceed with the soundness proof. We start by proving some auxiliary lemmas that will be useful later on in the proofs. We then show that for any Px86-consistent execution graph G of the implementation, its corresponding DSI execution graph G' satisfies the DSI consistency axioms and persistency axioms in turn. As a result of this, we are able to show soundness.

6.3.1 Auxiliary Lemmas

We start by assuming that the following lemma holds for any consistent execution graph with calls to the MRSW lock library defined in Section 6.1:

Proposition 1. *Given a Px86-consistent execution, for all* $pl, pl' \in \mathcal{PL}_x$, $wu, wu' \in \mathcal{WU}_x$, $rl, rl' \in \mathcal{RL}_x$ *and* $ru, ru' \in \mathcal{RU}_x$*:*

$$(rl, pl, wu), (rl', pl', wu') \in \mathsf{po}_x|_{imm} \implies (wu, rl') \in \mathsf{ob}_x \vee (wu', rl) \in \mathsf{ob}_x \qquad \text{(WWSync)}$$

$$(rl, pl, wu), (rl', ru) \in \mathsf{po}_x|_{imm} \implies (wu, rl') \in \mathsf{ob}_x \vee (ru, pl) \in \mathsf{ob}_x \qquad \text{(RWSync)}$$

where given a relation r we write $(a, b, c) \in r$ as shorthand for $(a, b), (b, c) \in r$.

The proof of a similar lemma is given in [29], where instead of the ob relation, a "lock-order" relation is used. The proof is similar and omitted here.

Intuitively, (WWSYNC) states that for some location x, if two threads are each able to acquire a reader lock on x, promote it to a writer lock and then release their writer lock on x, then one of them must release their writer lock on x ob-before the other can acquire their reader lock on x. Meanwhile, (RWSYNC) states that for some location x, if one thread τ_1 is able to acquire a reader lock on x, promote it to a writer lock and then release their writer lock on x, and another thread τ_2 is able to acquire and then release a reader lock on x, then either τ_1 has to release its writer lock on x ob-before τ_2 can acquire its reader lock on x, *or* τ_2 must release its reader lock on x ob-before τ_1 can *promote* its reader lock to a writer lock.

Next, we prove that for any distinct transactions ξ_a and ξ_b that are in the same thread such that ξ_a is not an initialisation or recovery transaction, and for some location x, the following two statements must hold: (1) if ξ_a releases a writer lock on x ob-before ξ_b acquires a reader lock on x, then these events are also po-related; and (2) if ξ_a releases a reader lock on x ob-before ξ_b promotes a reader lock on x, then these events are also po-related.

Lemma 1. *Given a Px86-consistent execution graph G of the implementation, for all a, b, ξ_a, ξ_b, x:*

$$\xi_a \neq \xi_b \wedge \xi_a \neq 0 \wedge \xi_a \notin T_{rec} \wedge \texttt{thrd}(\xi_a) = \texttt{thrd}(\xi_b) \wedge a \in \theta'(\xi_a) \wedge b \in \theta'(\xi_b) \wedge \texttt{loc}(a) = \texttt{loc}(b) = x \implies$$

$$(\theta(\xi_a).wu_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_x \implies \theta(\xi_a).wu_x \overset{G.\mathsf{po}}{\to} \theta(\xi_b).rl_x) \tag{6.3}$$

$$\wedge\ (\theta(\xi_a).ru_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).pl_x \implies \theta(\xi_a).ru_x \overset{G.\mathsf{po}}{\to} \theta(\xi_b).pl_x) \tag{6.4}$$

PROOF. Pick an arbitrary Px86-consistent execution graph G of the implementation. Pick an arbitrary a, b, ξ_a, ξ_b, x such that $\xi_a \neq \xi_b, \xi_a \neq 0, \xi_a \notin T_{rec}, \texttt{thrd}(\xi_a) = \texttt{thrd}(\xi_b)$ and $\texttt{loc}(a) = \texttt{loc}(b) = x$.

RTS. (6.3)
Assume $\theta(\xi_a).wu_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_x$. Since $\texttt{thrd}(\xi_a) = \texttt{thrd}(\xi_b)$, either (a) $\theta(\xi_a).wu_x \overset{G.\mathsf{po}}{\to} \theta(\xi_b).rl_x$ or (b) $\theta(\xi_b).rl_x \overset{G.\mathsf{po}}{\to} \theta(\xi_a).wu_x$. Assume (b) holds. Then, using REDEFINED-OB from the extended Px86$_{\text{axiom}}$ model (Fig. A.4), since $\theta(\xi_b).rl_x, \theta(\xi_a).wu_x \in L$ (where L represents the set of lock events as previously defined), $(\theta(\xi_b).rl_x, \theta(\xi_a).wu_x) \in \mathsf{dob}$. By definition of ob, this means that $\theta(\xi_b).rl_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_a).wu_x$, and so we have $\theta(\xi_a).wu_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_a).wu_x$. In other words, we have $\theta(\xi_a).wu_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_a).wu_x$, which is a cycle, but we know from EXTERNAL that ob is acyclic, so we have reached a contradiction and so (b) cannot hold. Therefore, $\theta(\xi_a).wu_x \overset{G.\mathsf{po}}{\to} \theta(\xi_b).rl_x$ must hold.

RTS. (6.4)
We proceed in a similar way to the proof done above for 6.3. Assume $\theta(\xi_a).ru_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).pl_x$. Then, either (a) $\theta(\xi_a).ru_x \overset{G.\mathsf{po}}{\to} \theta(\xi_b).pl_x$ or (b) $\theta(\xi_b).pl_x \overset{G.\mathsf{po}}{\to} \theta(\xi_a).ru_x$ since $G'.\mathsf{po}$ is a strict total order. Assume (b) holds. Then, using REDEFINED-OB from the extended Px86$_{\text{axiom}}$ model, since $\theta(\xi_b).pl_x, \theta(\xi_a).ru_x \in L$, we have that $(\theta(\xi_b).pl_x, \theta(\xi_a).ru_x) \in \mathsf{dob}$. By definition of ob, this means that $\theta(\xi_b).pl_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_a).ru_x$, and so we have $\theta(\xi_a).ru_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).pl_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_a).ru_x$. In other words, we have $\theta(\xi_a).ru_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_a).ru_x$, which is a cycle, but we know from EXTERNAL that ob is acyclic, so we have reached a contradiction and so (b) cannot hold. Therefore, $\theta(\xi_a).ru_x \overset{G.\mathsf{po}}{\to} \theta(\xi_b).pl_x$ must hold. □

Now we can prove another result relating to events in distinct transactions. At a high level, this lemma states that for any distinct transactions ξ_a and ξ_b such that ξ_a is not an initialisation or recovery transaction, and for some location x, then if there are any rf, mo or rb edges between the transactions at the abstract level, then there is some unlock event in ξ_a that is ob-before some lock or promotion event in ξ_b. This is expressed more formally below, and we prove this using our graph construction.

Lemma 2. *Given a Px86-consistent execution graph G of the implementation and its corresponding DSI execution graph G' constructed as above, for all a, b, ξ_a, ξ_b, x:*

$$\xi_a \neq \xi_b \wedge \xi_a \neq 0 \wedge \xi_a \notin T_{rec} \wedge a \in \theta'(\xi_a) \wedge b \in \theta'(\xi_b) \wedge \texttt{loc}(a) = \texttt{loc}(b) = x \implies$$

$$((a,b) \in G'.\mathsf{rf} \implies \theta(\xi_a).wu_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_x) \tag{6.5}$$

$$\wedge \quad ((a,b) \in G'.\mathsf{mo} \implies \theta(\xi_a).wu_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_x) \tag{6.6}$$

$$\wedge \quad ((a,b) \in G'.\mathsf{rb} \implies (x \in \texttt{WS}_{\xi_a} \wedge \theta(\xi_a).wu_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_x) \tag{6.7}$$

$$\vee\, (x \notin \texttt{WS}_{\xi_a} \wedge \theta(\xi_a).ru_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).pl_x))$$

PROOF. Pick an arbitrary Px86-consistent execution graph G of the implementation and its corresponding DSI execution graph G' constructed as above. Pick an arbitrary a, b, ξ_a, ξ_b, x such that $\xi_a \neq \xi_b, \xi_a \neq 0, \xi_a \notin T_{rec}, a \in \theta'(\xi_a) \wedge b \in \theta'(\xi_b)$ and $\texttt{loc}(a) = \texttt{loc}(b) = x$.

RTS. (6.5)
Assume $(a,b) \in G'.\mathsf{rf}$. Since $\xi_a \neq 0$, we know that $\xi_b \notin T_{rec}$. As such, from the definition of G'.rf given above, we know that $(\theta(\xi_a).w_x, \theta(\xi_b).r_x) \in G.\mathsf{rf}$. Clearly we have that $x \in \texttt{WS}_{\xi_a}$. Therefore, we proceed by case analysis on whether (1) $x \in \texttt{WS}_{\xi_b}$ or (2) $x \notin \texttt{WS}_{\xi_b}$.

In case (1), we have $(\xi_a.rl_x, \xi_a.pl_x, \xi_a.wu_x), (\xi_b.rl_x, \xi_b.pl_x, \xi_b.wu_x) \in \mathsf{po}_x|_{imm}$. By WWSYNC, either $\xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_a.rl_x$ or $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x$. On the other hand, in case (2), we have $(\xi_a.rl_x, \xi_a.pl_x, \xi_a.wu_x)$, $(\xi_b.rl_x, \xi_b.ru_x) \in \mathsf{po}_x|_{imm}$. By RWSYNC, either $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x$ or $\xi_b.ru_x \overset{G.\mathsf{ob}}{\to} \xi_a.pl_x$. We notice that the second case of (1) and the first case of (2) are the same, and so we combine this information together in order to deduce that one of the following three cases must hold: either (i) $x \in \texttt{WS}_{\xi_b}$ and $\xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_a.rl_x$; or (ii) $x \notin \texttt{WS}_{\xi_b}$ and $\xi_b.ru_x \overset{G.\mathsf{ob}}{\to} \xi_a.pl_x$; or (iii) $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x$.

In case (i), we have $\xi_a.w_x \overset{G.\mathsf{rf}}{\to} \xi_b.r_x \overset{G.\mathsf{po}}{\to} \xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_a.rl_x \overset{G.\mathsf{po}}{\to} \xi_a.w_x$. We know that $G.\mathsf{rf} = G.\mathsf{rf}_i \cup G.\mathsf{rf}_e$, and from the Px86-consistency of the execution we have that $G.\mathsf{rf}_i \subseteq G.\mathsf{po}$ (by CO-RW) and $G.\mathsf{rf}_e \subseteq G.\mathsf{obs} \subseteq G.\mathsf{ob}$ (by definition of $G.\mathsf{obs}$ and $G.\mathsf{ob}$ in Fig. A.3). We now consider the following two cases: a) ξ_a and ξ_b are in the same thread; or b) ξ_a and ξ_b are in different threads. In case (i.a), from the fact that $G.\mathsf{rf}_i \subseteq G.\mathsf{po}$ and Lemma 1, we have $\xi_a.w_x \overset{G.\mathsf{po}}{\to} \xi_b.r_x \overset{G.\mathsf{po}}{\to} \xi_b.wu_x \overset{G.\mathsf{po}}{\to} \xi_a.rl_x \overset{G.\mathsf{po}}{\to} \xi_a.w_x$. In other words, we have $\xi_a.w_x \overset{G.\mathsf{po}}{\to} \xi_a.w_x$, which contradicts the assumption that G is Px86-consistent. Therefore, case (i.a) cannot hold. In case (i.b), from the fact that $G.\mathsf{rf}_e \subseteq G.\mathsf{ob}$ and using REDEFINED-OB from the extended $\text{Px86}_{\text{axiom}}$ model, we have $\xi_a.w_x \overset{G.\mathsf{ob}}{\to} \xi_b.r_x \overset{G.\mathsf{ob}}{\to} \xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_a.rl_x \overset{G.\mathsf{ob}}{\to} \xi_a.w_x$. That is, we have $\xi_a.w_x \overset{G.\mathsf{ob}}{\to} \xi_a.w_x$, which contradicts the assumption that G is Px86-consistent by the acylicity of ob (EXTERNAL). Therefore, neither (i.a) not (i.b) can hold, and so case (i) cannot hold.

For case (ii), we can use a similar argument. Here we have $\xi_a.w_x \overset{G.\mathsf{rf}}{\to} \xi_b.r_x \overset{G.\mathsf{po}}{\to} \xi_b.ru_x \overset{G.\mathsf{ob}}{\to} \xi_a.pl_x \overset{G.\mathsf{po}}{\to} \xi_a.w_x$. Again we consider the two possible cases of a) ξ_a and ξ_b being in the same thread; and b) ξ_a and ξ_b being in different threads. In (ii.a) we use the fact that $G.\mathsf{rf}_i \subseteq G.\mathsf{po}$ and Lemma 1 to show that $\xi_a.w_x \overset{G.\mathsf{po}}{\to} \xi_b.r_x \overset{G.\mathsf{po}}{\to} \xi_b.ru_x \overset{G.\mathsf{po}}{\to} \xi_a.pl_x \overset{G.\mathsf{po}}{\to} \xi_a.w_x$, and so having $\xi_a.w_x \overset{G.\mathsf{po}}{\to} \xi_a.w_x$ contradicts the assumption that G is Px86-consistent. In (ii.b), we use the fact that $G.\mathsf{rf}_e \subseteq G.\mathsf{ob}$ and REDEFINED-OB to show that $\xi_a.w_x \overset{G.\mathsf{ob}}{\to} \xi_b.r_x \overset{G.\mathsf{ob}}{\to} \xi_b.ru_x \overset{G.\mathsf{ob}}{\to} \xi_a.pl_x \overset{G.\mathsf{ob}}{\to} \xi_a.w_x$, from which we get that $\xi_a.w_x \overset{G.\mathsf{ob}}{\to} \xi_a.w_x$. This contradicts the assumption that G is Px86-consistent by the acylicity of ob (EXTERNAL). Therefore, neither (ii.a) nor (ii.b) can hold, and so case (ii) cannot hold.

This leaves case (iii), from which the desired result immediately holds.

RTS. (6.6)
Assume $(a,b) \in G'.\mathsf{mo}$. As such, from the definition of $G'.\mathsf{mo}$, we know that $(\theta(\xi_a).w_x, \theta(\xi_b).w_x) \in G.\mathsf{mo}$. Clearly we have $x \in \texttt{WS}_{\xi_a}$ and $x \in \texttt{WS}_{\xi_b}$, so $(\xi_a.rl_x, \xi_a.pl_x, \xi_a.wu_x), (\xi_b.rl_x, \xi_b.pl_x, \xi_b.wu_x) \in \mathsf{po}_x|_{imm}$. Therefore, by WWSYNC, either (i) $\xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_a.rl_x$ or (ii) $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x$.

In case (i), we have $\xi_a.w_x \overset{G.\mathsf{mo}}{\to} \xi_b.w_x \overset{G.\mathsf{po}}{\to} \xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_a.rl_x \overset{G.\mathsf{po}}{\to} \xi_a.w_x$. We know that $G.\mathsf{mo} \subseteq G.\mathsf{obs} \subseteq G.\mathsf{ob}$ from the Px86-consistency of the execution; using this and REDEFINED-OB, we have that $\xi_a.w_x \overset{G.\mathsf{ob}}{\to} \xi_b.w_x \overset{G.\mathsf{ob}}{\to} \xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_a.rl_x \overset{G.\mathsf{ob}}{\to} \xi_a.w_x$, and so we also have $\xi_a.w_x \overset{G.\mathsf{ob}}{\to} \xi_a.w_x$, which contradicts the assumption that G is Px86-consistent by the acylicity of ob (EXTERNAL). Therefore, case (i) cannot hold.

This leaves case (ii), from which the desired result holds immediately.

RTS. (6.7)
Assume $(a, b) \in G'.\mathsf{rb}$. From the definition of rb, this means that $(a, b) \in (G'.\mathsf{rf}^{-1}; G'.\mathsf{mo})$, and so by unpacking this definition we find that there exists some $w \in W$ such that $(w, a) \in G'.\mathsf{rf}$ and $(w, b) \in G'.\mathsf{mo}$. We know that there exists some $x \in \mathrm{LOC}$ such that $\mathtt{loc}(a) = \mathtt{loc}(b) = \mathtt{loc}(w) = x$. Clearly we have $x \in \mathtt{WS}_{\xi_b}$ by the definition of $G'.\mathsf{mo}$. Therefore, we proceed by case analysis on whether (1) $x \in \mathtt{WS}_{\xi_a}$ or (2) $x \notin \mathtt{WS}_{\xi_a}$.

In case (1), we have $(\xi_a.rl_x, \xi_a.pl_x, \xi_a.wu_x), (\xi_b.rl_x, \xi_b.pl_x, \xi_b.wu_x) \in \mathsf{po}_x|_{imm}$. By WWSYNC, either $\xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_a.rl_x$ or $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x$. On the other hand, in case (2), we have $(\xi_b.rl_x, \xi_b.pl_x, \xi_b.wu_x)$, $(\xi_a.rl_x, \xi_a.ru_x) \in \mathsf{po}_x|_{imm}$. By RWSYNC, either $\xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_a.rl_x$ or $\xi_a.ru_x \overset{G.\mathsf{ob}}{\to} \xi_b.pl_x$. We notice that the first case of (1) and the first case of (2) are the same, and so we combine this information together in order to deduce that one of the following three cases must hold: either (i) $\xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_a.rl_x$; or (ii) $x \in \mathtt{WS}_{\xi_a}$ and $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x$; or (iii) $x \notin \mathtt{WS}_{\xi_a}$ and $\xi_a.ru_x \overset{G.\mathsf{ob}}{\to} \xi_b.pl_x$.

In case (i), we must consider the following three sub-cases relating to the transaction that w belongs to, which we call ξ_w: either (a) $\xi_w = \xi_a$; or (b) $\xi_w = \xi_b$; or (c) $\xi_w \neq \xi_a$ and $\xi_w \neq \xi_b$. In case (i.a), we have that $\xi_w \neq \xi_b$, $\xi_w \neq 0$ and $\xi_w \notin T_{rec}$ directly from the fact that $\xi_w = \xi_a$, for which those conditions already hold by assumption. Hence, using this and the fact that $(w, b) \in G'.\mathsf{mo}$, we have that $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x$ from 6.6. Therefore, we have $\xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_a.rl_x \overset{G.\mathsf{po}}{\to} \xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x \overset{G.\mathsf{po}}{\to} \xi_b.wu_x$. We can then use REDEFINED-OB to show that $\xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_a.rl_x \overset{G.\mathsf{ob}}{\to} \xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x \overset{G.\mathsf{ob}}{\to} \xi_b.wu_x$, and so we also have $\xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.wu_x$, which contradicts the assumption that G is Px86-consistent by the acylicity of ob (EXTERNAL). Therefore, (i.a) cannot hold.

In case (i.b), we have $\xi_w = \xi_b$, and so from the definition of $G'.\mathsf{mo}$ we have that $(\xi_b.t_w.lw_x, \xi_b.t_b.lw_x) \in G.\mathsf{po}$, where $t_w, t_b \in \theta(\xi_b).Ts$ are the implementation-level transaction event identifiers associated with w and b respectively. On the other hand, from the definition of $G'.\mathsf{rf}$, since $\xi_w \neq \xi_a$ we have $t_w = \xi_b.fw_x$. By definition, $\xi_b.fw_x$ is the maximal transactional write in po order on x, and so specifically we have $(\xi_b.t_b.lw_x, \xi_b.t_w.lw_x) \in G.\mathsf{po}$. This gives $\xi_b.t_w.lw_x \overset{G.\mathsf{po}}{\to} \xi_b.t_b.lw_x \overset{G.\mathsf{po}}{\to} \xi_b.t_w.lw_x$, and so $\xi_b.t_w.lw_x \overset{G.\mathsf{po}}{\to} \xi_b.t_w.lw_x$, which contradicts the Px86-consistency of G since $G.\mathsf{po}$ is acyclic. Therefore, (i.b) cannot hold.

In case (i.c), we know that all three transactions ξ_a, ξ_b and ξ_w are distinct, and so using the definitions of $G'.\mathsf{rf}$ and $G'.\mathsf{mo}$ for events in separate transactions, we have that $(\xi_w.w_x, \xi_a.r_x) \in G.\mathsf{rf}$ and $(\xi_w.w_x, \xi_b.w_x) \in G.\mathsf{mo}$. Therefore, by the definition of rb, we have $(\xi_a.r_x, \xi_b.w_x) \in G.\mathsf{rb}$. Using this, we have $\xi_a.r_x \overset{G.\mathsf{rb}}{\to} \xi_b.w_x \overset{G.\mathsf{po}}{\to} \xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x \overset{G.\mathsf{po}}{\to} \xi_a.r_x$. We now consider the following two cases: 1) ξ_a and ξ_b are in the same thread; or 2) ξ_a and ξ_b are in different threads. In case (i.c.1), we use CO-WR to show that $\xi_a.r_x \overset{G.\mathsf{rb}}{\to} \xi_b.w_x$ implies $\xi_a.r_x \overset{G.\mathsf{po}}{\to} \xi_b.w_x$. Using this, as well as Lemma 1 on the ob-related events, we find that $\xi_a.r_x \overset{G.\mathsf{po}}{\to} \xi_b.w_x \overset{G.\mathsf{po}}{\to} \xi_b.wu_x \overset{G.\mathsf{po}}{\to} \xi_b.rl_x \overset{G.\mathsf{po}}{\to} \xi_a.r_x$, and so we have $\xi_a.r_x \overset{G.\mathsf{po}}{\to} \xi_a.r_x$, which contradicts the assumption that G is Px86-consistent. Therefore, case (i.c.1) cannot hold. In case (i.c.2), we can use the fact that $\mathsf{rb}_e \subseteq \mathsf{obs} \subseteq \mathsf{ob}$ to show that $\xi_a.r_x \overset{G.\mathsf{rb}}{\to} \xi_b.w_x$ implies $\xi_a.r_x \overset{G.\mathsf{ob}}{\to} \xi_b.w_x$. Using this, as well as REDEFINED-OB on the po-related events, we find that $\xi_a.r_x \overset{G.\mathsf{ob}}{\to} \xi_b.w_x \overset{G.\mathsf{ob}}{\to} \xi_b.wu_x \overset{G.\mathsf{ob}}{\to} \xi_a.rl_x \overset{G.\mathsf{ob}}{\to} \xi_a.r_x$, and so we have $\xi_a.r_x \overset{G.\mathsf{ob}}{\to} \xi_a.r_x$, which contradicts the assumption that G is Px86-consistent by the acyclicity of ob (EXTERNAL). Therefore, case (i.c.2) cannot hold either, and so (i.c) cannot hold overall.

This leaves cases (ii) and (iii), from which the desired result holds immediately.

□

6.3.2 Consistency Axioms

We now move beyond our auxiliary lemmas to prove that this implementation over Px86 is correct with respect to the DSI *consistency* guarantees. An outline of this section is as follows:

- We first prove Lemma 3, which states that for any abstract-level events a and b which are part of distinct (non-initialisation and non-recovery) transactions and are $\widetilde{\mathsf{hb}_{\mathsf{SI}}}$-related, their corresponding implementation-level events are ob-related.
- This can then be used in conjunction with Proposition 2, a result from [29] which can be interpreted as stating that showing the irreflexivity of $\widetilde{\mathsf{hb}_{\mathsf{SI}}}$ is equivalent to showing the irreflexivity of $\mathsf{hb}_{\mathsf{SI}}$.
- Next, we recall that the second DSI consistency axiom (SI_2) states that $G'.\mathsf{hb}_{\mathsf{SI}}$ is irreflexive and so we can prove this result by showing that $G'.\widetilde{\mathsf{hb}_{\mathsf{SI}}}$ is irreflexive, using the result of Proposition 2 and by applying Lemma 3. We use a proof by contradiction to show that this result holds.
- Finally, we prove the first DSI consistency axiom (SI_1) as well, thus showing that G' ensures the consistency guarantees of DSI.

We follow this outline to show that for any Px86-consistent execution graph G of the implementation, its corresponding DSI execution graph G' satisfies the DSI consistency axioms.

Lemma 3. *Given a Px86-consistent execution graph G of the implementation and its corresponding DSI execution graph G' constructed as shown above, for all a, b:*

$$(a, b) \in G'.\widetilde{\mathsf{hb}_{\mathsf{SI}}} \wedge a \notin G'.(I \cup Rec) \implies (\mathtt{imp}(a), \mathtt{imp}(b)) \in G.\mathsf{ob}$$

where $G'.\widetilde{\mathsf{hb}_{\mathsf{SI}}} \triangleq (G'.((\mathsf{po}_{\mathsf{T}} \cup \mathsf{rf}_{\mathsf{T}} \cup \mathsf{mo}_{\mathsf{T}}); \mathsf{rb}_{\mathsf{T}}^{?}))^{+}$.

PROOF. Let $G'.\widetilde{\mathsf{hb}_{\mathsf{SI}}}^{1} \triangleq G'.((\mathsf{po}_{\mathsf{T}} \cup \mathsf{rf}_{\mathsf{T}} \cup \mathsf{mo}_{\mathsf{T}}); \mathsf{rb}_{\mathsf{T}}^{?})$, and $G'.\widetilde{\mathsf{hb}_{\mathsf{SI}}}^{n+1} \triangleq G'.\widetilde{\mathsf{hb}_{\mathsf{SI}}}^{1}; G'.\widetilde{\mathsf{hb}_{\mathsf{SI}}}^{n}$, for all $n > 1$. We can then show the following equivalent result:

$$\forall n \in \mathbb{N}^{+}. (a, b) \in G'.\widetilde{\mathsf{hb}_{\mathsf{SI}}}^{n} \wedge a \notin G'.(I \cup Rec) \implies (\mathtt{imp}(a), \mathtt{imp}(b)) \in G.\mathsf{ob}$$

We proceed using a proof by induction on n.

Base Case: $n = 1$
Pick arbitrary a, b such that $(a, b) \in G'.\widetilde{\mathsf{hb}_{\mathsf{SI}}}^{1}$ and $a \notin G'.(I \cup Rec)$. Given the definition of $\widetilde{\mathsf{hb}_{\mathsf{SI}}}^{1}$, we therefore know that one of the following six cases must hold: either (i) $(a, b) \in G'.\mathsf{po}_{\mathsf{T}}$; or (ii) $(a, b) \in G'.(\mathsf{po}_{\mathsf{T}}; \mathsf{rb}_{\mathsf{T}})$; or (iii) $(a, b) \in G'.\mathsf{rf}_{\mathsf{T}}$; or (iv) $(a, b) \in G'.(\mathsf{rf}_{\mathsf{T}}; \mathsf{rb}_{\mathsf{T}})$; or (v) $(a, b) \in G'.\mathsf{mo}_{\mathsf{T}}$; or (vi) $(a, b) \in G'.(\mathsf{mo}_{\mathsf{T}}; \mathsf{rb}_{\mathsf{T}})$.

In case (i), we know that $a, b \in W \cup R$ due to the construction of G'; by this construction, $G'.E$ can only comprise read and write events, as shown in 6.1 and 6.2. Therefore, we must have $\mathtt{imp}(a), \mathtt{imp}(b) \in W \cup R$ as well, by definition of the $\mathtt{imp}$ function, which maps abstract read (resp. write) events to implementation-level read (write) events. We also have $(a, b) \in G'.\mathsf{po}$ directly from the assumption made in this case about $(a, b) \in G'.\mathsf{po}_{\mathsf{T}}$, since the distinct transactions ξ_a and ξ_b associated with a and b respectively are in the same thread, and so all events in ξ_a are po-ordered before all events in ξ_b. Using the fact that $a \notin G'.(I \cup Rec)$ as well as the definition of $G'.\mathsf{po}$, we see that $(\mathtt{imp}(a), \mathtt{imp}(b)) \in G.\mathsf{po}$. There are now two cases to consider: a) $(a, b) \notin W \times R$; or b) $(a, b) \in W \times R$.

In case (i.a), using the definition of $\mathtt{imp}$ we have $(\mathtt{imp}(a), \mathtt{imp}(b)) \notin W \times R$ as well. Therefore, since $(\mathtt{imp}(a), \mathtt{imp}(b)) \in G.\mathsf{po}$, and from the definition REDEFINED-OB, we have that $(\mathtt{imp}(a), \mathtt{imp}(b)) \in G.\mathsf{ob}$.

In case (i.b), we have that $(a, b) \in W \times R$. Taking $\mathtt{loc}(a) = x$ and $\mathtt{loc}(b) = y$ for some $x, y \in \text{LOC}$, we then know that $(\mathtt{imp}(a), \mathtt{imp}(b)) \in W \times R$, $\mathtt{loc}(\mathtt{imp}(a)) = x$ and $\mathtt{loc}(\mathtt{imp}(b)) = y$ (all by definition of the $\mathtt{imp}$ function). Since $\mathtt{imp}(a) \in W$ and $\mathtt{imp}(b) \in R$, we know from the definition of $\mathtt{imp}$ that $\mathtt{imp}(a) = \theta(\xi_a).lw_x$ and $\mathtt{imp}(b) = \theta(\xi).rs_y$. From this, the structure of G and the knowledge that $(\mathtt{imp}(a), \mathtt{imp}(b)) \in G.\mathsf{po}$, we then know that there exist ξ_a, ξ_b such that $\mathtt{imp}(a) \overset{G.\mathsf{po}}{\to} \theta(\xi_a).wu_x \overset{G.\mathsf{po}}{\to} \theta(\xi_b).rl_y \overset{G.\mathsf{po}}{\to} \mathtt{imp}(b)$. Considering each event individually, we can see that $\mathtt{imp}(a) \in W$, $\theta(\xi_a).wu_x \in L$, $\theta(\xi_b).rl_y \in L$ and $\mathtt{imp}(b) \in R$. Using

this information and the definition REDEFINED-OB, we can obtain $\texttt{imp}(a) \overset{G.\mathsf{ob}}{\to} \theta(\xi_a).wu_x \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_y \overset{G.\mathsf{ob}}{\to} \texttt{imp}(b)$, and so $(\texttt{imp}(a), \texttt{imp}(b)) \in G.\mathsf{ob}$, as required.

In case (ii), we know that there exists a c such that $(a, c) \in G'.\mathsf{po}_\mathsf{T}$ and $(c, b) \in G'.\mathsf{rb}_\mathsf{T}$. Let us take ξ_a, ξ_b, ξ_c to be the identifiers of the distinct transactions corresponding to the abstract events a, b, c respectively. From the fact that $(c, b) \in G'.\mathsf{rb}_\mathsf{T}$, we know that there exist some d, e such that $\xi_c \neq \xi_b, (d, e) \in G'.\mathsf{rb}, c, d \in \xi_c$ and $b, e \in \xi_b$. Due to the construction of G' and since $\xi_a \neq 0, \xi_a \notin T_{rec}$, we know that $\xi_c \neq 0, \xi_c \notin T_{rec}$ as well. Therefore, from Lemma 2 and taking $\texttt{loc}(d) = \texttt{loc}(e) = x$ for some $x \in \textsc{Loc}$, we can have two possible cases: either (a) $x \in \texttt{WS}_{\xi_c}$ and $\xi_c.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x$; or (b) $x \notin \texttt{WS}_{\xi_c}$ and $\xi_c.ru_x \overset{G.\mathsf{ob}}{\to} \xi_b.pl_x$.

In case (ii.a), we already know that $(\texttt{imp}(a), \texttt{imp}(c)) \in G.\mathsf{ob}$ by case (i) of this lemma. Combining this with the information we already have, we get $\texttt{imp}(a) \overset{G.\mathsf{ob}}{\to} \texttt{imp}(c) \overset{G.\mathsf{po}}{\to} \xi_c.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x \overset{G.\mathsf{po}}{\to} \texttt{imp}(b)$. Using the facts that $\texttt{imp}(b), \texttt{imp}(c) \in W \cup R$ and $\xi_c.wu_x, \xi_b.rl_x \in L$ in conjunction with REDEFINED-OB, we get $\texttt{imp}(a) \overset{G.\mathsf{ob}}{\to} \texttt{imp}(c) \overset{G.\mathsf{ob}}{\to} \xi_c.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x \overset{G.\mathsf{ob}}{\to} \texttt{imp}(b)$, and so $(\texttt{imp}(a), \texttt{imp}(b)) \in G.\mathsf{ob}$, as required. In case (ii.b), we deduce from $(a, c) \in G'.\mathsf{po}_\mathsf{T}$ that ξ_a and ξ_c are in the same thread, and additionally that all events in ξ_a are po-ordered before all events in ξ_c (both at the abstract and implementation levels). From this, we get $\texttt{imp}(a) \overset{G.\mathsf{po}}{\to} \xi_c.ru_x \overset{G.\mathsf{ob}}{\to} \xi_b.pl_x \overset{G.\mathsf{po}}{\to} \texttt{imp}(b)$. Using the facts that $\texttt{imp}(a), \texttt{imp}(b) \in W \cup R$ and $\xi_c.ru_x, \xi_b.pl_x \in L$ in conjunction with REDEFINED-OB, we get $\texttt{imp}(a) \overset{G.\mathsf{ob}}{\to} \xi_c.ru_x \overset{G.\mathsf{ob}}{\to} \xi_b.pl_x \overset{G.\mathsf{ob}}{\to} \texttt{imp}(b)$, and $(\texttt{imp}(a), \texttt{imp}(b)) \in G.\mathsf{ob}$, as required.

In case (iii), we know that there exist ξ_a, ξ_b, d, e such that $\xi_a \neq \xi_b, (d, e) \in G'.\mathsf{rf}, \xi_a \neq 0, \xi_a \notin T_{rec}, a, d \in \theta'(\xi_a)$ and $b, e \in \theta'(\xi_b)$. If we let $\texttt{loc}(d) = \texttt{loc}(e) = y$ (where $y \in \textsc{Loc}$), then we know that $\texttt{imp}(a) \overset{G.\mathsf{po}}{\to} \theta(\xi_a).wu_y \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_y \overset{G.\mathsf{po}}{\to} \texttt{imp}(b)$. Here, the first and final pairs of events are $G.\mathsf{po}$-related due to the construction of G and the subset of events that can be returned by $\texttt{imp}$ (i.e. $\theta(\xi).rs_x$ and $\theta(\xi).lw_x$ for some location x and a given transaction ξ). Meanwhile, the middle pair is $G.\mathsf{ob}$-related by Lemma 2. Considering each event individually, we can see that $\texttt{imp}(a) \in W \cup R$, $\theta(\xi_a).wu_y \in L$, $\theta(\xi_b).rl_y \in L$ and $\texttt{imp}(b) \in W \cup R$. Using this information and the definition REDEFINED-OB, we can obtain $\texttt{imp}(a) \overset{G.\mathsf{ob}}{\to} \theta(\xi_a).wu_y \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_y \overset{G.\mathsf{ob}}{\to} \texttt{imp}(b)$, and so $(\texttt{imp}(a), \texttt{imp}(b)) \in G.\mathsf{ob}$, as required.

In case (iv), we know that there exists a c such that $(a, c) \in G'.\mathsf{rf}_\mathsf{T}$ and $(c, b) \in G'.\mathsf{rb}_\mathsf{T}$. Let us take ξ_a, ξ_b, ξ_c to be the identifiers of the distinct transactions corresponding to the abstract events a, b, c respectively. From the fact that $(c, b) \in G'.\mathsf{rb}_\mathsf{T}$, we know that there exist some d, e such that $\xi_c \neq \xi_b, (d, e) \in G'.\mathsf{rb}, c, d \in \xi_c$ and $b, e \in \xi_b$. Due to the construction of G' and since $\xi_a \neq 0, \xi_a \notin T_{rec}$, we know that $\xi_c \neq 0, \xi_c \notin T_{rec}$ as well. Therefore, from Lemma 2 and taking $\texttt{loc}(d) = \texttt{loc}(e) = x$ for some $x \in \textsc{Loc}$, we can have two possible cases: either (a) $x \in \texttt{WS}_{\xi_c}$ and $\xi_c.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x$; or (b) $x \notin \texttt{WS}_{\xi_c}$ and $\xi_c.ru_x \overset{G.\mathsf{ob}}{\to} \xi_b.pl_x$.

In case (iv.a), we already know that $(\texttt{imp}(a), \texttt{imp}(c)) \in G.\mathsf{ob}$ by case (iii) of this lemma. Combining this with the information we already have, we get $\texttt{imp}(a) \overset{G.\mathsf{ob}}{\to} \texttt{imp}(c) \overset{G.\mathsf{po}}{\to} \xi_c.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x \overset{G.\mathsf{po}}{\to} \texttt{imp}(b)$. Using the facts that $\texttt{imp}(b), \texttt{imp}(c) \in W \cup R$ and $\xi_c.wu_x, \xi_b.rl_x \in L$ in conjunction with REDEFINED-OB, we get $\texttt{imp}(a) \overset{G.\mathsf{ob}}{\to} \texttt{imp}(c) \overset{G.\mathsf{ob}}{\to} \xi_c.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x \overset{G.\mathsf{ob}}{\to} \texttt{imp}(b)$, and so $(\texttt{imp}(a), \texttt{imp}(b)) \in G.\mathsf{ob}$, as required. In case (iv.b), we use the fact that $(a, c) \in G'.\mathsf{rf}_\mathsf{T}$ and the argument from case (iii) of the proof of this lemma to get $\texttt{imp}(a) \overset{G.\mathsf{po}}{\to} \theta(\xi_a).wu_y \overset{G.\mathsf{ob}}{\to} \theta(\xi_c).rl_y$ for some $y \in \textsc{Loc}$. From this, we get $\texttt{imp}(a) \overset{G.\mathsf{po}}{\to} \xi_a.wu_y \overset{G.\mathsf{ob}}{\to} \xi_c.rl_y \overset{G.\mathsf{po}}{\to} \xi_c.ru_x \overset{G.\mathsf{ob}}{\to} \xi_b.pl_x \overset{G.\mathsf{po}}{\to} \texttt{imp}(b)$. Using the facts that $\texttt{imp}(a), \texttt{imp}(b) \in W \cup R$ and all of the remaining events are in L, as well as REDEFINED-OB, we get $\texttt{imp}(a) \overset{G.\mathsf{ob}}{\to} \xi_a.wu_y \overset{G.\mathsf{ob}}{\to} \xi_c.rl_y \overset{G.\mathsf{ob}}{\to} \xi_c.ru_x \overset{G.\mathsf{ob}}{\to} \xi_b.pl_x \overset{G.\mathsf{ob}}{\to} \texttt{imp}(b)$, and $(\texttt{imp}(a), \texttt{imp}(b)) \in G.\mathsf{ob}$, as required.

In case (v), we use an analogous strategy to the proof of case (iii) above. We know that there exist ξ_a, ξ_b, d, e such that $\xi_a \neq \xi_b, (d, e) \in G'.\mathsf{mo}, \xi_a \neq 0, \xi_a \notin T_{rec}, a, d \in \theta'(\xi_a)$ and $b, e \in \theta'(\xi_b)$. If we let $\texttt{loc}(d) = \texttt{loc}(e) = y$ (where $y \in \textsc{Loc}$), then we know that $\texttt{imp}(a) \overset{G.\mathsf{po}}{\to} \theta(\xi_a).wu_y \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_y \overset{G.\mathsf{po}}{\to} \texttt{imp}(b)$, where the first and final pairs of events are $G.\mathsf{po}$-related due to the construction of G and the subset of events that can be returned by $\texttt{imp}$ (i.e. $\theta(\xi).rs_x$ and $\theta(\xi).lw_x$ for some location x and a given transaction ξ). Meanwhile, the middle pair is $G.\mathsf{ob}$-related by Lemma 2. Considering each event individually, we can see that $\texttt{imp}(a) \in W \cup R$, $\theta(\xi_a).wu_y \in L$, $\theta(\xi_b).rl_y \in L$ and $\texttt{imp}(b) \in W \cup R$. Using this information and the definition REDEFINED-OB, we can obtain $\texttt{imp}(a) \overset{G.\mathsf{ob}}{\to} \theta(\xi_a).wu_y \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_y \overset{G.\mathsf{ob}}{\to} \texttt{imp}(b)$, and so

$(\texttt{imp}(a), \texttt{imp}(b)) \in G.\mathsf{ob}$, as required.

In case (vi), we know that there exists a c such that $(a, c) \in G'.\mathsf{mo}_\mathsf{T}$ and $(c, b) \in G'.\mathsf{rb}_\mathsf{T}$. Let us take ξ_a, ξ_b, ξ_c to be the identifiers of the distinct transactions corresponding to the abstract events a, b, c respectively. From the fact that $(c, b) \in G'.\mathsf{rb}_\mathsf{T}$, we know that there exist some d, e such that $\xi_c \neq \xi_b, (d, e) \in G'.\mathsf{rb}, c, d \in \xi_c$ and $b, e \in \xi_b$. Due to the construction of G' and since $\xi_a \neq 0, \xi_a \notin T_{rec}$, we know that $\xi_c \neq 0, \xi_c \notin T_{rec}$ as well. Therefore, from Lemma 2 and taking $\texttt{loc}(d) = \texttt{loc}(e) = x$ for some $x \in \textsc{Loc}$, we can have two possible cases: either (a) $x \in \texttt{WS}_{\xi_c}$ and $\xi_c.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x$; or (b) $x \notin \texttt{WS}_{\xi_c}$ and $\xi_c.ru_x \overset{G.\mathsf{ob}}{\to} \xi_b.pl_x$.

In case (vi.a), we already know that $(\texttt{imp}(a), \texttt{imp}(c)) \in G.\mathsf{ob}$ by case (iii) of this lemma. Combining this with the information we already have, we get $\texttt{imp}(a) \overset{G.\mathsf{ob}}{\to} \texttt{imp}(c) \overset{G.\mathsf{po}}{\to} \xi_c.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x \overset{G.\mathsf{po}}{\to} \texttt{imp}(b)$. Using the facts that $\texttt{imp}(b), \texttt{imp}(c) \in W \cup R$ and $\xi_c.wu_x, \xi_b.rl_x \in L$ in conjunction with REDEFINED-OB, we get $\texttt{imp}(a) \overset{G.\mathsf{ob}}{\to} \texttt{imp}(c) \overset{G.\mathsf{ob}}{\to} \xi_c.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x \overset{G.\mathsf{ob}}{\to} \texttt{imp}(b)$, and so $(\texttt{imp}(a), \texttt{imp}(b)) \in G.\mathsf{ob}$, as required. In case (vi.b), we use the fact that $(a, c) \in G'.\mathsf{mo}_\mathsf{T}$ and the argument from case (v) of the proof of this lemma to get $\texttt{imp}(a) \overset{G.\mathsf{po}}{\to} \theta(\xi_a).wu_y \overset{G.\mathsf{ob}}{\to} \theta(\xi_c).rl_y$ for some $y \in \textsc{Loc}$. From this, we get $\texttt{imp}(a) \overset{G.\mathsf{po}}{\to} \xi_a.wu_y \overset{G.\mathsf{ob}}{\to} \xi_c.rl_y \overset{G.\mathsf{po}}{\to} \xi_c.ru_x \overset{G.\mathsf{ob}}{\to} \xi_b.pl_x \overset{G.\mathsf{po}}{\to} \texttt{imp}(b)$. Using the facts that $\texttt{imp}(a), \texttt{imp}(b) \in W \cup R$ and all of the remaining events are in L, as well as REDEFINED-OB, we get $\texttt{imp}(a) \overset{G.\mathsf{ob}}{\to} \xi_a.wu_y \overset{G.\mathsf{ob}}{\to} \xi_c.rl_y \overset{G.\mathsf{ob}}{\to} \xi_c.ru_x \overset{G.\mathsf{ob}}{\to} \xi_b.pl_x \overset{G.\mathsf{ob}}{\to} \texttt{imp}(b)$, and $(\texttt{imp}(a), \texttt{imp}(b)) \in G.\mathsf{ob}$, as required.

Inductive Case: $n = m + 1$ **for** $m > 0$
We pick arbitrary a, b such that $(a, b) \in G'.\widetilde{\mathsf{hb}_\mathsf{SI}}^{n}$ and $a \notin G'.(I \cup Rec)$. This means that there exist c, ξ_c such that $(a, c) \in G'.\widetilde{\mathsf{hb}_\mathsf{SI}}^{1}, (c, b) \in G'.\widetilde{\mathsf{hb}_\mathsf{SI}}^{m}$ and $c \in \theta'(\xi_c)$. From the proof of the base case, we have $(\texttt{imp}(a), \texttt{imp}(c)) \in G.\mathsf{ob}$. In addition, due to the construction of G' and since $\xi_a \neq 0, \xi_a \notin T_{rec}$, we know that $\xi_c \neq 0, \xi_c \notin T_{rec}$ as well. Therefore, using the inductive hypothesis we have $(\texttt{imp}(c), \texttt{imp}(b)) \in G.\mathsf{ob}$. This implies that $\texttt{imp}(a) \overset{G.\mathsf{ob}}{\to} \texttt{imp}(c) \overset{G.\mathsf{ob}}{\to} \texttt{imp}(b)$, and so we have $(\texttt{imp}(a), \texttt{imp}(b)) \in G.\mathsf{ob}$, as required. □

Proposition 2. *Given an arbitrary abstract DSI execution graph G':*

$$\mathit{acyclic}(G'.((\mathsf{po}_\mathsf{T} \cup \mathsf{rf}_\mathsf{T} \cup \mathsf{mo}_\mathsf{T}); \mathsf{rb}_\mathsf{T}^?)) \iff \mathit{irreflexive}(\mathsf{hb}_\mathsf{SI})$$

The full proof of this is given in [29].

Lemma 4. *(Soundness – Consistency Axioms.) For all Px86-consistent execution graphs G of the implementation and their counterpart DSI execution graphs G' constructed as above:*

$$G'.((\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb}) \cap \mathsf{st}) \subseteq G'.\mathsf{po} \tag{6.8}$$
$$G'.\mathsf{hb}_\mathsf{SI} \textit{ is irreflexive} \tag{6.9}$$

PROOF. Pick an arbitrary Px86-consistent execution G of the implementation and its corresponding DSI execution graph G', constructed as given above.

RTS. (6.8)
This result follows directly from the construction of G' as follows. We wish to show that for some events a and b in the same transaction, if $(a, b) \in G'.(\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})$, then $(a, b) \in G'.\mathsf{po}$ as well. We consider the following three cases: either (i) $(a, b) \in G'.\mathsf{rf}$; or (ii) $(a, b) \in G'.\mathsf{mo}$; or (iii) $(a, b) \in G'.\mathsf{rb}$.

In case (i), we know we must have either $(a, b) \in G'.\mathsf{po}$ or $(b, a) \in G'.\mathsf{po}$ since po is a strict total order for events in the same thread. Assume for a contradiction that $(b, a) \in G'.\mathsf{po}$. Then, due to the construction of $G'.\mathsf{po}$, we have that all events in t_b are po-ordered before all of those in t_a, where t_a and t_b are the implementation-level events associated with a and b respectively. From this, we get $(t_b.rs_x, t_a.ws_x) \in G.\mathsf{po}$, since we know that $a \in W$ and $b \in R$. Additionally, from the construction of $G'.\mathsf{rf}$,, we have $(t_a.ws_x, t_b.rs_x) \in G.\mathsf{rf}$. Therefore, we have $t_a.ws_x \overset{G.\mathsf{rf}}{\to} t_b.rs_x \overset{G.\mathsf{po}}{\to} t_a.ws_x$. Since $\texttt{thrd}(a) = \texttt{thrd}(b)$ and $G.\mathsf{rf}_i \subseteq G.\mathsf{po}$ by CO-RW, we have $t_a.ws_x \overset{G.\mathsf{po}}{\to} t_b.rs_x \overset{G.\mathsf{po}}{\to} t_a.ws_x$, and so we have $t_a.ws_x \overset{G.\mathsf{po}}{\to} t_a.ws_x$,

which contradicts the Px86-consistency of G since $G.\mathsf{po}$ is acyclic as it is a strict total order. Therefore, our initial assumption was false, and so we must have $(a, b) \in G'.\mathsf{po}$.

In case (ii), the desired result holds immediately from the definition of $G'.\mathsf{mo}$.

In case (iii), if $(a, b) \in G'.\mathsf{rb}$, then there is some $c \in W$ such that $(c, a) \in G'.\mathsf{rf}$ and $(c, b) \in G'.\mathsf{mo}$. We call ξ_a and ξ_c the transaction identifiers associated with a and c respectively. There are now two cases to consider: either (a) $\xi_a = \xi_c$; or (b) $\xi_a \neq \xi_c$.

In case (iii.a), a, b and c are all in the same transaction. Therefore, from the definitions of $G'.\mathsf{rf}$ and $G'.\mathsf{mo}$ for abstract events in the same transaction, and for some location $x \in \mathrm{Loc}$ such that $\mathtt{loc}(a) = \mathtt{loc}(b) = \mathtt{loc}(c) = x$, we have $(t_c.ws_x, t_a.rs_x) \in G.\mathsf{rf}$ and $(t_c.ws_x, t_b.ws_x) \in G.\mathsf{po}$, where t_a, t_b and t_c are the implementation-level transactional events associated with a, b and c respectively. From the latter result, we also have $(t_c.ws_x, t_b.ws_x) \in G.\mathsf{mo}$ since we know that both events are writing to the same location x, as well as from the knowledge that mo is total on W_x and $G.\mathsf{mo}_i \subseteq G.\mathsf{po}$. From this, we have that $(t_a.rs_x, t_b.ws_x) \in G.\mathsf{rb}$. Now, we know that either $(a, b) \in G'.\mathsf{po}$ or $(b, a) \in G'.\mathsf{po}$ since po is total. Assume for a contradiction that $(b, a) \in G'.\mathsf{po}$. Then, since we know $a \in R$ and $b \in W$, we have $(t_b.ws_x, t_a.rs_x) \in G.\mathsf{po}$ from the definition of $G'.\mathsf{po}$. Therefore, we have $t_a.rs_x \overset{G.\mathsf{rb}}{\to} t_b.ws_x \overset{G.\mathsf{po}}{\to} t_a.rs_x$. Since $\mathtt{thrd}(a) = \mathtt{thrd}(b)$ and $G.\mathsf{rb}_i \subseteq G.\mathsf{po}$ by CO-WR, we have $t_a.rs_x \overset{G.\mathsf{po}}{\to} t_b.ws_x \overset{G.\mathsf{po}}{\to} t_a.rs_x$, and so $t_a.rs_x \overset{G.\mathsf{po}}{\to} t_a.rs_x$, which contradicts the Px86-consistency of G since $G.\mathsf{po}$ is acyclic. Therefore, our assumption that $(b, a) \in G'.\mathsf{po}$ was false, and so we have $(a, b) \in G'.\mathsf{po}$, as required.

In case (iii.b), we have a and b in the same transaction (which we call ξ_1), and c is in a separate transaction ξ_2. Therefore, from the definition of $G'.\mathsf{rf}$ for abstract events in different transactions, and for some location $x \in \mathrm{Loc}$ such that $\mathtt{loc}(a) = \mathtt{loc}(b) = \mathtt{loc}(c) = x$, we have $(\xi_1.\widehat{ws}_x, \xi_1.t_a.rs_x) \in G.\mathsf{rf}$. In addition, since $b \in W$, we have $(\xi_1.\widehat{ws}_x, \xi_1.t_b.ws_x) \in G.\mathsf{po}$, and from this we can deduce that $(\xi_1.\widehat{ws}_x, \xi_1.t_b.ws_x) \in G.\mathsf{mo}$ since both events are writing to the same location x and $G.\mathsf{mo}_i \subseteq G.\mathsf{po}$. Therefore, we have $(\xi_1.t_a.rs_x, \xi_1.t_b.ws_x) \in G.\mathsf{rb}$. Now, we know that either $(a, b) \in G'.\mathsf{po}$ or $(b, a) \in G'.\mathsf{po}$ since po is total. Assume for a contradiction that $(b, a) \in G'.\mathsf{po}$. Then, since we have $a \in R$ and $b \in W$, we also have $(\xi_1.t_b.ws_x, \xi_1.t_a.rs_x) \in G.\mathsf{po}$ from the definition of $G'.\mathsf{po}$. As a result, we have $\xi_1.t_a.rs_x \overset{G.\mathsf{rb}}{\to} \xi_1.t_b.ws_x \overset{G.\mathsf{po}}{\to} \xi_1.t_a.rs_x$. Since $\mathtt{thrd}(a) = \mathtt{thrd}(b)$ and $G.\mathsf{rb}_i \subseteq G.\mathsf{po}$ by CO-WR, we have $\xi_1.t_a.rs_x \overset{G.\mathsf{po}}{\to} \xi_1.t_b.ws_x \overset{G.\mathsf{po}}{\to} \xi_1.t_a.rs_x$, and so $\xi_1.t_a.rs_x \overset{G.\mathsf{po}}{\to} \xi_1.t_a.rs_x$, which contradicts the Px86-consistency of G since $G.\mathsf{po}$ is acyclic. Therefore, our assumption that $(b, a) \in G'.\mathsf{po}$ was false, and so we have $(a, b) \in G'.\mathsf{po}$, as required.

RTS. (6.9)
Here, we proceed by contradiction. We first notice that showing the irreflexivity of $\mathsf{hb}_{\mathsf{SI}}$ is equivalent to showing the irreflexivity of $\widetilde{\mathsf{hb}_{\mathsf{SI}}}$, using Proposition 2 and the definition of acyclity. Therefore, we assume there exists some a such that $(a, a) \in G'.\widetilde{\mathsf{hb}_{\mathsf{SI}}}$. We note that given the construction of G', the initialisation events in $G'.I$ have no incoming $G'.(\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})$ edges, and so this cycle does not contain any initialisation events in $G'.I$, and in particular, we have $a \notin G'.I$. We consider the incoming $G'.(\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})$ edges because by definition of $G'.\widetilde{\mathsf{hb}_{\mathsf{SI}}}$, we need to either consider the incoming $G'.(\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo})$ edges, or the incoming $G'.\mathsf{rb}$ edges, which can be combined to give the expression we use above. In addition, since the only incoming $G'.(\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})$ edges to the events in $G'.Rec$ are those from the initialisation events from $G'.I$, and since we already established that this cycle does not contain any initialisation events, we also have that this cycle contains no events from $G'.Rec$, and in particular, $a \notin G'.Rec$. As such, from Lemma 3, we have $(\mathtt{imp}(a), \mathtt{imp}(a)) \in G.\mathsf{ob}$, which contradicts the Px86-consistency of G since $G.\mathsf{ob}$ is acyclic by EXTERNAL. We have reached a contradiction, and so there is no a such that $(a, a) \in G'.\widetilde{\mathsf{hb}_{\mathsf{SI}}}$. Therefore, we must have that $G'.\widetilde{\mathsf{hb}_{\mathsf{SI}}}$ is irreflexive, and so $G'.\mathsf{hb}_{\mathsf{SI}}$ is irreflexive. □

6.3.3 Persistency Axioms

Now that we have established soundness with respect to the consistency guarantees of DSI, we now wish to show that this implementation is correct with respect to the DSI *persistency* guarantees. An outline of this section is as follows.

We start by defining the set of transactions that contain at least one write instruction, known as the set of writing instructions ($\mathcal{W}$), and the set of transactions that contain no write instructions, known as read-only transactions ($\mathcal{R}$). We then wish to show that the DSI axiom (DSI-NVO) holds for G'. For this part of the proof, we choose to use the Px86 model (tso) over the $\text{Px86}_{\text{axiom}}$ model (ob) since the former models nvo explicitly. We wish to show that any two abstract durable events that are related by $\mathsf{hb}_{\mathsf{SI}}$ are tso-related, with sufficient persistency primitives such as flushes and fences in between the two events, since this allows us to use the persistency axioms of the Px86 model to show that they are nvo-related as well. The difficulty of this proof is that while the endpoints of the $\mathsf{hb}_{\mathsf{SI}}$ edge between transactions are guaranteed to be durable events (i.e., in D), each $\mathsf{hb}_{\mathsf{SI}}$ edge is composed of (often multiple) edges belonging to po_{T}, rf_{T}, mo_{T} or $\mathsf{rb}_{\mathsf{SI}}$. These edges may start or end with read instructions, which are not durable events, and so this makes the proof of this axiom more difficult than usual since there may be non-durable nodes (instructions) on the path of an $\mathsf{hb}_{\mathsf{SI}}$ edge, meaning it is not always possible to use the inductive hypothesis when a read is present on such a path.

To remedy this, we prove a number of helper lemmas and then build up to the lemmas that involve proving both DSI persistency axioms. We start with the knowledge that $(a, b) \in G'.\mathsf{hb}_{\mathsf{SI}} \cap (D \times D)$ is the same as saying $(a, b) \in G'.\mathsf{hb}_{\mathsf{r}}; \mathsf{hb}_{\mathsf{SI}}{}^{n}$ (where $\mathsf{hb}_{\mathsf{r}} \triangleq G'.(\mathsf{po}_{\mathsf{T}} \cup \mathsf{rf}_{\mathsf{T}} \cup \mathsf{mo}_{\mathsf{T}})^{+}$). This is because due to having $G'.\mathsf{hb}_{\mathsf{SI}} \cap (D \times D)$, there exists no c such that $(a, c) \in G'.\mathsf{rb}_{\mathsf{SI}}$ and $(c, b) \in G'.\mathsf{hb}_{\mathsf{SI}}$ because $a \in R$ in this case. Therefore, to prove (DSI-NVO), it is sufficient to prove that $G.(\mathsf{hb}_{\mathsf{r}}; \mathsf{hb}_{\mathsf{SI}}{}^{n}) \cap (D \times D) \subseteq \mathsf{nvo}$ instead of the original format of the axiom presented in the model.

This can be explained in more detail, lemma by lemma.

- Lemma 5 states that if there is an hb_{r} edge between two distinct transactions ξ_a and ξ_b at the abstract level, then a lock is released on some location in ξ_a ob-before a reader lock is acquired on some (possibly distinct) location in ξ_b. We prove this by induction.
- Lemma 6 essentially states that for any two distinct transactions, if there is an $\mathsf{hb}_{\mathsf{SI}}$ edge between the two at the abstract level, then at the implementation level, there is some unlocking event in ξ_a which is ob-before some locking or promotion event in ξ_b. This is done by splitting $\mathsf{hb}_{\mathsf{SI}}$ into three sub-parts: $\mathsf{hb}_{\mathsf{r}}; \mathsf{hb}_{\mathsf{SI}}{}^{n}$ with $\xi_a \in \mathcal{W}$; $\mathsf{hb}_{\mathsf{r}}; \mathsf{hb}_{\mathsf{SI}}{}^{n}$ with $\xi_a \in \mathcal{R}$; and $\mathsf{rb}_{\mathsf{SI}}; \mathsf{hb}_{\mathsf{SI}}{}^{n}$.
- Lemma 7 uses the results from the rest of the section to show that the two DSI persistency axioms hold.

We follow this outline to show that for any Px86-consistent execution graph G of the implementation, its corresponding DSI execution graph G' satisfies the DSI persistency axioms.

Definition 5. The set of writing transactions, $\mathcal{W}$, and read-only transactions, $\mathcal{R}$, are defined as follows:

$$\mathcal{W} \triangleq \{\xi \mid \exists w \in W.\, \mathtt{tx}(w) = \xi\}$$
$$\mathcal{R} \triangleq \text{TXID} \setminus \mathcal{W}$$

Lemma 5. *Given a Px86-consistent execution graph G of the implementation and its corresponding DSI execution graph G' constructed as above, for all a, b, ξ_a, ξ_b:*

$$\begin{aligned}
&\xi_a \neq 0 \wedge \xi_a \notin T_{rec} \wedge a \in \theta'(\xi_a) \wedge b \in \theta'(\xi_b) \wedge (a, b) \in G'.\mathsf{hb}_{\mathsf{r}} \implies \\
&\exists c, d.\, [(\xi_a \in \mathcal{W} \wedge \theta(\xi_a).wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_{\mathtt{loc}(d)}) \\
&\quad \vee (\xi_a \in \mathcal{R} \wedge \theta(\xi_a).ru_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_{\mathtt{loc}(d)})
\end{aligned}$$

where $G'.\mathsf{hb}_{\mathsf{r}} \triangleq G'.(\mathsf{po}_{\mathsf{T}} \cup \mathsf{rf}_{\mathsf{T}} \cup \mathsf{mo}_{\mathsf{T}})^{+}$.

PROOF. Let $G'.\mathsf{hb}_{\mathsf{r}}{}^{1} \triangleq G'.(\mathsf{po}_{\mathsf{T}} \cup \mathsf{rf}_{\mathsf{T}} \cup \mathsf{mo}_{\mathsf{T}})$, and $G'.\mathsf{hb}_{\mathsf{r}}{}^{n+1} \triangleq G'.\mathsf{hb}_{\mathsf{r}}{}^{1}; G'.\mathsf{hb}_{\mathsf{r}}{}^{n}$, for all $n > 1$. We can then show the following equivalent result:

$$\begin{aligned}
&\forall n \in \mathbb{N}^{+}.\, \xi_a \neq 0 \wedge \xi_a \notin T_{rec} \wedge a \in \theta'(\xi_a) \wedge b \in \theta'(\xi_b) \wedge (a, b) \in G'.\mathsf{hb}_{\mathsf{r}}{}^{n} \implies \\
&\quad \exists c, d.\, [(\xi_a \in \mathcal{W} \wedge \theta(\xi_a).wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_{\mathtt{loc}(d)}) && (6.10) \\
&\quad\quad \vee (\xi_a \in \mathcal{R} \wedge \theta(\xi_a).ru_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_{\mathtt{loc}(d)}) && (6.11)
\end{aligned}$$

We proceed using a proof by induction on n.

Base Case: $n = 1$
Pick an arbitrary Px86-consistent execution graph G of the implementation. Pick an arbitrary a, b, ξ_a, ξ_b, x such that $\xi_a \neq \xi_b, \xi_a \neq 0, \xi_a \notin T_{rec}$. Given the definition of $\mathsf{hb_r}^1$, we know that one of the following three cases must hold: either (i) $(a,b) \in G'.\mathsf{po_T}$; or (ii) $(a,b) \in G'.\mathsf{rf_T}$; or (iii) $(a,b) \in G'.\mathsf{mo_T}$.

In case (i), since $(a,b) \in G'.\mathsf{po_T}$, we have $(a,b) \in G'.\mathsf{po}$ as well since our assumption implies that ξ_a and ξ_b are in the same thread, and so all events in ξ_a are po-before all events in ξ_b. We know that either (a) $\xi_a \in \mathcal{W}$ or (b) $\xi_b \in \mathcal{R}$ by the definitions of $\mathcal{W}$ and $\mathcal{R}$. In case (i.a), we know that there is some $c \in \xi_a$ such that $\mathtt{loc}(c) \in \mathtt{WS}_{\xi_a}$. From this and the fact that $b \in \xi_b$, we have $\xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{po}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(b)}$. Using REDEFINED-OB with this result, we get $\xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(b)}$, from which the first disjunct (6.10) holds. In case (i.b), since $\xi_a \in \mathcal{R}$, we know that $a \in R$ and $\mathtt{loc}(a) \notin \mathtt{WS}_{\xi_a}$ (since $\mathtt{WS}_{\xi_a} = \emptyset$ by definition of $\mathcal{R}$). From this and the fact that $b \in \xi_b$ we get $\xi_a.ru_{\mathtt{loc}(a)} \overset{G.\mathsf{po}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(b)}$. Using REDEFINED-OB with this result, we get $\xi_a.ru_{\mathtt{loc}(a)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(b)}$, from which the second disjunct (6.11) holds. Therefore, we have shown that case (i) holds overall.

In case (ii), since $(a,b) \in G'.\mathsf{rf_T}$, we know there exist some $(w,r) \in G'.\mathsf{rf}$ such that $w \in W, r \in R$ and $w \in \xi_a, r \in \xi_b$. Let us take $\mathtt{loc}(w) = \mathtt{loc}(r) = x$ for some $x \in \mathrm{LOC}$. We can see that ξ_a is clearly not a read-only transaction because it contains at least one write since $w \in \xi_a$. Therefore, we must have $\xi_a \in \mathcal{W}$, and so we only need to show that the first disjunct (6.10) holds. From Lemma 2, we have $\xi_a.wu_x \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_x$, from which the desired result holds. The second disjunct (6.11) holds vacuously, and so case (ii) holds overall.

In case (iii), since $(a,b) \in G'.\mathsf{mo_T}$, we know there exist some $(w_1, w_2) \in G'.\mathsf{mo}$ such that $w_1, w_2 \in W$ and $w_1 \in \xi_a, w_2 \in \xi_b$. Let us take $\mathtt{loc}(w_1) = \mathtt{loc}(w_2) = x$ for some $x \in \mathrm{LOC}$. We can see that ξ_a is clearly not a read-only transaction because it contains at least one write since $w_1 \in \xi_a$. Therefore, we must have $\xi_a \in \mathcal{W}$, and so we only need to show that the first disjunct (6.10) holds. From Lemma 2, we have $\xi_a.wu_x \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_x$, and the desired result holds from this. The second disjunct (6.11) holds vacuously, and so we have shown case (iii) holds overall.

Inductive Case: $n = m + 1$ **for** $m > 0$
We pick an arbitrary $(a,b) \in G'.\mathsf{hb_r}^n$. From the definition of $G'.\mathsf{hb_r}^n$, we have $(a,b) \in G'.(\mathsf{hb_r}^1; \mathsf{hb_r}^m)$. From the definition of composition of relations as well as the definition of $G'.\mathsf{hb_r}$ again, we know that there are some e, ξ_e such that $(a,e) \in G'.\mathsf{hb_r}^1$ and $(e,b) \in G'.\mathsf{hb_r}^m$, and also $e \in \theta'(\xi_e), \xi_e \neq \xi_a$ and $\xi_e \neq \xi_b$. From the proof of the base case, we know that there exist some c, d such that: either (i) $\xi_a \in \mathcal{W}$ and $\xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_e.rl_{\mathtt{loc}(d)}$; or (ii) $\xi_a \in \mathcal{R}$ and $\xi_a.ru_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_e.rl_{\mathtt{loc}(d)}$. We consider cases (i) and (ii) separately.

In case (i), we can use the inductive hypothesis on $(e,b) \in G'.\mathsf{hb_r}^m$ to show that there exist some c', d' such that either (a) $\xi_e \in \mathcal{W}$ and $\xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$; or (b) $\xi_e \in \mathcal{R}$ and $\xi_e.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$. In (i.a), we have $\xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_e.rl_{\mathtt{loc}(d)} \overset{G.\mathsf{po}}{\rightarrow} \xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$. Using REDEFINED-OB, we have $\xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_e.rl_{\mathtt{loc}(d)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$, and so we have $\xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$. In (i.b), we have $\xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_e.rl_{\mathtt{loc}(d)} \overset{G.\mathsf{po}}{\rightarrow} \xi_e.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$. Using REDEFINED-OB, we have $\xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_e.rl_{\mathtt{loc}(d)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_e.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$, and so $\xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$ holds. We have shown in both cases (i.a) and (i.b) that $\xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$ holds, and we already assumed $\xi_a \in \mathcal{W}$ as part of case (i), so we have proved the first disjunct (6.10).

In case (ii), we can again use the inductive hypothesis on $(e,b) \in G'.\mathsf{hb_r}^m$ to show that there exist some c', d' such that either (a) $\xi_e \in \mathcal{W}$ and $\xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$; or (b) $\xi_e \in \mathcal{R}$ and $\xi_e.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$. In (ii.a), we have $\xi_a.ru_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_e.rl_{\mathtt{loc}(d)} \overset{G.\mathsf{po}}{\rightarrow} \xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$. Using REDEFINED-OB, we have $\xi_a.ru_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_e.rl_{\mathtt{loc}(d)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$, and so we have $\xi_a.ru_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$. In (ii.b), we have $\xi_a.ru_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_e.rl_{\mathtt{loc}(d)} \overset{G.\mathsf{po}}{\rightarrow} \xi_e.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.rl_{\mathtt{loc}(d')}$. Using REDEFINED-OB, we have

$\xi_a.ru_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\texttt{loc}(d)} \overset{G.\mathsf{ob}}{\to} \xi_e.ru_{\texttt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\texttt{loc}(d')}$, and so $\xi_a.ru_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\texttt{loc}(d')}$ holds. We have shown in both cases (ii.a) and (ii.b) that $\xi_a.ru_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\texttt{loc}(d')}$ holds, and we already assumed $\xi_a \in \mathcal{R}$ as part of case (ii), so we have proved the second disjunct (6.11). Therefore, we have proved both disjuncts, so the lemma holds for all $n \in \mathbb{N}^+$.

□

Lemma 6. *Given a Px86-consistent execution graph G of the implementation and its corresponding DSI execution graph G' constructed as above, for all a, b, ξ_a, ξ_b:*

$$\begin{aligned}
&\forall n \in \mathbb{N}.\, \xi_a \neq 0 \wedge \xi_a \notin T_{rec} \wedge a \in \theta'(\xi_a) \wedge b \in \theta'(\xi_b) \implies \\
&\quad ((a,b) \in G'.(\mathsf{hb_r}; \mathsf{hb_{SI}}^n) \wedge \xi_a \in \mathcal{W} \implies \\
&\qquad \exists c, d.\, (\theta(\xi_a).wu_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_{\texttt{loc}(d)} \vee \theta(\xi_a).wu_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).pl_{\texttt{loc}(d)})) \qquad (6.12) \\
&\wedge\ ((a,b) \in G'.(\mathsf{hb_r}; \mathsf{hb_{SI}}^n) \wedge \xi_a \in \mathcal{R} \implies \\
&\qquad \exists c, d.\, (\theta(\xi_a).ru_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_{\texttt{loc}(d)} \vee \theta(\xi_a).ru_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).pl_{\texttt{loc}(d)})) \qquad (6.13) \\
&\wedge\ ((a,b) \in G'.(\mathsf{rb_{SI}}; \mathsf{hb_{SI}}^n) \implies \\
&\qquad \exists c, d.\, (\theta(\xi_a).wu_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_{\texttt{loc}(d)} \vee \theta(\xi_a).wu_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).pl_{\texttt{loc}(d)} \\
&\qquad \vee\ \theta(\xi_a).ru_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).rl_{\texttt{loc}(d)} \vee \theta(\xi_a).ru_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \theta(\xi_b).pl_{\texttt{loc}(d)})) \qquad (6.14)
\end{aligned}$$

where $G'.\mathsf{hb_{SI}}^0 \triangleq \mathtt{id}$, $G'.\mathsf{hb_{SI}}^1 \triangleq G'.(\mathsf{po_T} \cup \mathsf{rf_T} \cup \mathsf{mo_T} \cup \mathsf{rb_{SI}})$ and $G'.\mathsf{hb_{SI}}^{n+1} \triangleq G'.(\mathsf{hb_{SI}}^1; \mathsf{hb_{SI}}^n)$ for all $n > 1$.

PROOF. Pick an arbitrary Px86-consistent execution graph of the implementation and its corresponding DSI execution graph G' constructed as above. Pick an arbitrary a, b, ξ_a, ξ_b such that $\xi_a \neq 0, \xi_a \notin T_{rec}, a \in \theta'(\xi_a)$ and $b \in \theta'(\xi_b)$. We proceed using a proof by induction on n.

Base Case: $n = 0$
RTS. (6.12)
Assume $(a,b) \in G'.(\mathsf{hb_r}; \mathsf{hb_{SI}}^0)$ and $\xi_a \in \mathcal{W}$. Then we have $(a,b) \in G'.\mathsf{hb_r}$, and so using Lemma 5 we know that there exist c, d such that $\xi_a.wu_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\texttt{loc}(d)}$. This is one of the disjuncts of the expression we wish to show, and so by the semantics of disjunctions the result follows.

RTS. (6.13)
Assume $(a,b) \in G'.(\mathsf{hb_r}; \mathsf{hb_{SI}}^0)$ and $\xi_a \in \mathcal{R}$. Then we have $(a,b) \in G'.\mathsf{hb_r}$, and so by Lemma 5 there exist c, d such that $\xi_a.ru_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\texttt{loc}(d)}$. Since this is one of the disjuncts of the expression we wish to show, the result follows from this and using the semantics of disjunctions.

RTS. (6.14)
Assume $(a,b) \in G'.(\mathsf{rb_{SI}}; \mathsf{hb_{SI}}^0)$. Then we have $(a,b) \in G'.\mathsf{rb_{SI}}$, and so there exist some $r \in R, w \in W$ such that $(r,w) \in G'.\mathsf{rb}$, and also $r \in \xi_a$ and $w \in \xi_b$. Let us say $\texttt{loc}(r) = \texttt{loc}(w) = x$ for some $x \in \text{LOC}$. Then by Lemma 2 we know that $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_x \vee \xi_a.ru_x \overset{G.\mathsf{ob}}{\to} \xi_b.pl_x$ holds, and since these are both disjuncts of the expression we wish to show, the result follows from this and the semantics of disjunctions.

Inductive Case: $n = m + 1$ **for** $m > 0$
RTS. (6.12)
Assume $(a,b) \in G'.(\mathsf{hb_r}; \mathsf{hb_{SI}}^n)$ and $\xi_a \in \mathcal{W}$. The first assumption is equivalent to $(a,b) \in G'.(\mathsf{hb_r}; (\mathsf{hb_{SI}}^1; \mathsf{hb_{SI}}^m))$, which in turn is equivalent to saying there exists some e, ξ_e such that $(a,e) \in G'.\mathsf{hb_r}$ and $(e,b) \in G'.(\mathsf{hb_{SI}}^1; \mathsf{hb_{SI}}^m)$, with $e \in \xi_e$ as well as $\xi_e \neq \xi_a$ and $\xi_e \neq \xi_b$. We seek to show that there exist some c, d such that:

$$\xi_a.wu_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\texttt{loc}(d)} \vee \xi_a.wu_{\texttt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\texttt{loc}(d)} \qquad (6.15)$$

From Lemma 5 and the fact that $(a,e) \in G'.\mathsf{hb_r}$, we know that there exist c', d' such that $\xi_a.wu_{\texttt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\texttt{loc}(d')}$. In addition, from the definition of $\mathsf{hb_{SI}}^1$, we can see that it can be expressed as $\mathsf{hb_r}^1 \cup \mathsf{rb_{SI}}$, and

so we consider two cases: either (i) $(e,b) \in G'.(\mathsf{hb_r}^1; \mathsf{hb_{SI}}^m)$; or (ii) $(e,b) \in G'.(\mathsf{rb_{SI}}; \mathsf{hb_{SI}}^m)$.

In case (i), since $G'.\mathsf{hb_r}^1 \subseteq G'.\mathsf{hb_r}$, we can use the inductive hypothesis to show that there exist c'', d'' such that: either (a) $\xi_e.wu_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$; or (b) $\xi_e.wu_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$; or (c) $\xi_e.ru_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$; or (d) $\xi_e.ru_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$. The first two cases come from 6.12 when $\xi_e \in \mathcal{W}$, and the latter two cases come from 6.13 when $\xi_e \in \mathcal{R}$.

In (i.a), we have $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{po}}{\to} \xi_e.wu_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$, and so using REDEFINED-OB, we get $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{ob}}{\to} \xi_e.wu_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$. Therefore, $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$, and using the semantics of disjunctions we have $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')} \vee \xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, from which the desired result holds. In (i.b), we have $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{po}}{\to} \xi_e.wu_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, and so using REDEFINED-OB, we get $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{ob}}{\to} \xi_e.wu_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$. From this, we have $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, and so using the semantics of disjunctions we have $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')} \vee \xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$; the desired result holds from this. In (i.c), we have $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{po}}{\to} \xi_e.ru_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$, and so using REDEFINED-OB, we get $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{ob}}{\to} \xi_e.ru_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$. Simplifying this, we get $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$, and then using the semantics of disjunctions we have $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')} \vee \xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, from which the desired result holds. In (i.d), we can see that $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{po}}{\to} \xi_e.ru_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, and so using REDEFINED-OB, we get $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{ob}}{\to} \xi_e.ru_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$. From this, we get $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, and then using the semantics of disjunctions we have $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')} \vee \xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, from which the desired result holds. Therefore, we have been able to show this result holds for case (i).

In case (ii), we can see from using the inductive hypothesis on 6.14 that the same four cases hold as in case (i); hence, by the same case analysis we find that $\xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')} \vee \xi_a.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, and the desired result holds from this.

RTS. (6.13)
Assume $(a,b) \in G'.(\mathsf{hb_r}; \mathsf{hb_{SI}}^n)$ and $\xi_a \in \mathcal{R}$. As before, the first assumption is equivalent to $(a,b) \in G'.(\mathsf{hb_r}; (\mathsf{hb_{SI}}^1; \mathsf{hb_{SI}}^m))$, which in turn is equivalent to saying there exists some e, ξ_e such that $(a,e) \in G'.\mathsf{hb_r}$ and $(e,b) \in G'.(\mathsf{hb_{SI}}^1; \mathsf{hb_{SI}}^m)$, with $e \in \xi_e$ as well as $\xi_e \neq \xi_a$ and $\xi_e \neq \xi_b$. We seek to show that there exist some c, d such that:

$$\xi_a.ru_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d)} \vee \xi_a.ru_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d)} \tag{6.16}$$

From Lemma 5 and the fact that $(a,e) \in G'.\mathsf{hb_r}$, we know that there exist c', d' such that $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')}$. In addition, from the definition of $\mathsf{hb_{SI}}^1$, we can see that it can be expressed as $\mathsf{hb_r}^1 \cup \mathsf{rb_{SI}}$, and so we consider two cases: either (i) $(e,b) \in G'.(\mathsf{hb_r}^1; \mathsf{hb_{SI}}^m)$; or (ii) $(e,b) \in G'.(\mathsf{rb_{SI}}; \mathsf{hb_{SI}}^m)$.

In case (i), since $G'.\mathsf{hb_r}^1 \subseteq G'.\mathsf{hb_r}$, we can use the inductive hypothesis to show that there exist c'', d'' such that: either (a) $\xi_e.wu_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$; or (b) $\xi_e.wu_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$; or (c) $\xi_e.ru_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$; or (d) $\xi_e.ru_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$. The first two cases come from 6.12 when $\xi_e \in \mathcal{W}$, and the latter two cases come from 6.13 when $\xi_e \in \mathcal{R}$.

In (i.a), we have $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{po}}{\to} \xi_e.wu_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$, and so using REDEFINED-OB, we get $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{ob}}{\to} \xi_e.wu_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$. Therefore, $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$, and using the semantics of disjunctions we have $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')} \vee \xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, from which the desired result holds. In (i.b), we have $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{po}}{\to} \xi_e.wu_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, and so using REDEFINED-OB, we get $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{ob}}{\to} \xi_e.wu_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$.

From this, we have $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, and so using the semantics of disjunctions we have $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')} \vee \xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$; the desired result holds from this. In (i.c), we have $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{po}}{\to} \xi_e.ru_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$, and so using REDEFINED-OB, we get $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{ob}}{\to} \xi_e.ru_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$. Simplifying this, we get $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')}$, and then using the semantics of disjunctions we have $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')} \vee \xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, from which the desired result holds. In (i.d), we can see that $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{po}}{\to} \xi_e.ru_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, and so using REDEFINED-OB, we get $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_e.rl_{\mathtt{loc}(d')} \overset{G.\mathsf{ob}}{\to} \xi_e.ru_{\mathtt{loc}(c'')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$. From this, we get $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, and then using the semantics of disjunctions we have $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')} \vee \xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, from which the desired result holds. Therefore, we have been able to show this result holds for case (i).

In case (ii), we can see from using the inductive hypothesis on 6.14 that the same four cases hold as in case (i); hence, by the same case analysis we find that $\xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d'')} \vee \xi_a.ru_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d'')}$, and the desired result holds from this.

RTS. (6.14)
Assume $(a,b) \in G'.(\mathsf{rb}_{\mathsf{SI}}; \mathsf{hb}_{\mathsf{SI}}{}^n)$. This is equivalent to $(a,b) \in G'.(\mathsf{rb}_{\mathsf{SI}}; (\mathsf{hb}_{\mathsf{SI}}{}^1; \mathsf{hb}_{\mathsf{SI}}{}^m))$, which in turn is equivalent to saying there exists some e, ξ_e such that $(a,e) \in G'.\mathsf{rb}_{\mathsf{SI}}$ and $(e,b) \in G'.(\mathsf{hb}_{\mathsf{SI}}{}^1; \mathsf{hb}_{\mathsf{SI}}{}^m)$, with $e \in \xi_e$ as well as $\xi_e \neq \xi_a$ and $\xi_e \neq \xi_b$. We wish to show that there exist some c, d such that:

$$\begin{aligned} &\xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d)} \vee \xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d)} \\ \vee\ &\xi_a.ru_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d)} \vee \xi_a.ru_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d)} \end{aligned} \tag{6.17}$$

Firstly, since $(a,e) \in G'.\mathsf{rb}_{\mathsf{SI}}$, we know that there exist some $r \in R, w \in W$ such that $(r,w) \in G.\mathsf{rb}$, with $r \in \xi_a$ and $w \in \xi_e$. We can take $\mathtt{loc}(r) = \mathtt{loc}(w) = x$ for some $x \in \mathrm{Loc}$. We know from the definition of $\mathsf{hb}_{\mathsf{SI}}{}^1$ that it can be expressed as $\mathsf{hb}_{\mathsf{r}}{}^1 \cup \mathsf{rb}_{\mathsf{SI}}$, and so we consider two cases: either (i) $(e,b) \in G'.(\mathsf{hb}_{\mathsf{r}}{}^1; \mathsf{hb}_{\mathsf{SI}}{}^m)$; or (ii) $(e,b) \in G'.(\mathsf{rb}_{\mathsf{SI}}; \mathsf{hb}_{\mathsf{SI}}{}^m)$.

In order to show the result holds in case (i), we first note that from Lemma 2, we know that either (a) $x \in \mathtt{WS}_{\xi_a}$ and $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_e.rl_x$; or (b) $x \notin \mathtt{WS}_{\xi_a}$ and $\xi_a.ru_x \overset{G.\mathsf{ob}}{\to} \xi_e.pl_x$. We also note that $\xi_e \in \mathcal{W}$ because $w \in \xi_e$. We now proceed by case analysis.

In case (i.a), since we know $\xi_e \in \mathcal{W}$, we can use the inductive hypothesis from 6.12 to show that there exist c', d' such that: either (1) $\xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d')}$; or (2) $\xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d')}$. In case (i.a.1), we have $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_e.rl_x \overset{G.\mathsf{po}}{\to} \xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d')}$. Using REDEFINED-OB, we have $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_e.rl_x \overset{G.\mathsf{ob}}{\to} \xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d')}$, and so $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d')}$. Then, since this is one of the disjuncts in 6.17, we can see that the desired result holds from the semantics of disjunctions. In case (i.a.2), we know that $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_e.rl_x \overset{G.\mathsf{po}}{\to} \xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d')}$. Using REDEFINED-OB, we have $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_e.rl_x \overset{G.\mathsf{ob}}{\to} \xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d')}$, and so $\xi_a.wu_x \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d')}$. Now, since this is one of the disjuncts in 6.17, we have that the desired result holds from the semantics of disjunctions. Therefore, we have shown that the result holds in case (i.a) overall.

In case (i.b), again from the fact that $\xi_e \in \mathcal{W}$, we can use the inductive hypothesis from 6.12 to show that there exist c', d' such that: either (1) $\xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d')}$; or (2) $\xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d')}$. In case (i.b.1), we have $\xi_a.ru_x \overset{G.\mathsf{ob}}{\to} \xi_e.pl_x \overset{G.\mathsf{po}}{\to} \xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d')}$. From REDEFINED-OB, we have $\xi_a.ru_x \overset{G.\mathsf{ob}}{\to} \xi_e.pl_x \overset{G.\mathsf{ob}}{\to} \xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d')}$, and so $\xi_a.ru_x \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d')}$. Then, since this is one of the disjuncts in 6.17, we can see that the desired result holds from the semantics of disjunctions. In case (i.b.2), we know that $\xi_a.ru_x \overset{G.\mathsf{ob}}{\to} \xi_e.pl_x \overset{G.\mathsf{po}}{\to} \xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d')}$. From REDEFINED-OB, we have $\xi_a.ru_x \overset{G.\mathsf{ob}}{\to} \xi_e.pl_x \overset{G.\mathsf{ob}}{\to} \xi_e.wu_{\mathtt{loc}(c')} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d')}$, and so $\xi_a.ru_x \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d')}$. Now, due to the fact that this is one of the disjuncts in 6.17, we can see that the desired result holds from the semantics of

disjunctions. Therefore, the result holds in case (i.b) and so it holds in the entirety of case (i).

In case (ii), we immediately notice that since $(a, e) \in G'.\mathsf{rb}_{\mathsf{SI}}$, we must have $e \in W$ by the definition of $\mathsf{rb}_{\mathsf{SI}}$, and so case (ii) cannot ever occur as it would require $e \in R$. Therefore, the result holds vacuously in case (ii).

□

Lemma 7. *(Soundness – Persistency Axioms.) For all Px86-consistent execution graphs G of the implementation and their counterpart DSI execution graphs G' constructed as above:*

$$G'.\mathsf{hb}_{\mathsf{SI}} \cap (D \times D) \subseteq G'.\mathsf{nvo} \tag{6.18}$$
$$dom(G'.[D]; \mathsf{st}; [P]) \subseteq G'.P \subseteq G'.T \tag{6.19}$$

PROOF. Pick an arbitrary Px86-consistent execution G of the implementation and its corresponding DSI execution graph G', constructed as given above.

RTS. (6.18)
We start by picking an arbitrary a, b such that $(a, b) \in G'.\mathsf{hb}_{\mathsf{SI}}$ and $a, b \in G'.D$, and so from this we have that $a, b \in W$ since all of the durable events of G' are writes. We now have the following three cases to consider: (i) $a \in G'.I$; or (ii) $a \in G'.Rec$; or (iii) $a \in G'.Run$.

In case (i), we can see from the construction of G' that the initialisation events in $G'.I$ have no incoming $G'.\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb}$ edges. From this and the definition of $G'.\mathsf{hb}_{\mathsf{SI}}$, we can deduce that these initialisation events have no incoming $G'.\mathsf{hb}_{\mathsf{SI}}$ edges, and so we know that $b \notin G'.I$. In addition, from the definition of $G'.\mathsf{nvo}$ we have that $G'.I \times ((G'.E \setminus G'.I) \cap D) \subseteq G'.\mathsf{nvo}$. Therefore, we have $a \in G'.I$ and $b \in ((G'.E \setminus G'.I) \cap D)$, and so $(a, b) \in G'.\mathsf{nvo}$ as required.

In case (ii), we can see from the construction of G' that the only outgoing $G'.\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb}$ edges of events in Rec are to events in $Rec \cup Run$. Therefore, we must have $b \in G'.Rec \cup Run$. If $b \in G'.Rec$, then using the facts that $(a, b) \in G'.D \cap Rec$ and $(a, b) \notin G'.\mathsf{st}$, as well as our initial assumption that $(a, b) \in G'.\mathsf{hb}_{\mathsf{SI}}$, we have $(a, b) \in G'.\mathsf{nvo}$ from the definition of $G'.\mathsf{nvo}$. On the other hand, if $b \in G'.Run$, then due to the fact that $((Rec \cap D) \times (Run \times D)) \subseteq G'.\mathsf{nvo}$, as well as knowing that $a \in (Rec \cap D)$ and $b \in (Run \times D)$, we have $(a, b) \in G'.\mathsf{nvo}$, as required.

In case (iii), given the construction of G', we know that the only outgoing $G'.\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb}$ edges of events in Run are to events in Run. Therefore, we must have $b \in G'.Run$. From this and axiom 6.9, we have $a, b \notin G'.\mathsf{st}$, and so there exist transactions ξ_a, ξ_b such that $\xi_a \neq \xi_b$, $a \in \theta'(\xi_a)$ and $b \in \theta'(\xi_b)$. We know that since $a \in W$, having $(a, b) \in G'.\mathsf{hb}_{\mathsf{SI}}$ is equivalent to having $(a, b) \in G'.(\mathsf{hb}_{\mathsf{r}}; \mathsf{hb}_{\mathsf{SI}}{}^n)$ for some $n \in \mathbb{N}$ (where hb_{r} and $\mathsf{hb}_{\mathsf{SI}}{}^n$ are as defined in Lemma 5 and Lemma 6). This is because $a \notin G'.\mathsf{rb}_{\mathsf{SI}}$ due to the fact that $a \notin R$. In addition, since $a \in \xi_a$, we know that $\xi_a \in \mathcal{W}$, and so from the first conjunct of Lemma 6 we know that there exist some c, d such that: either (a) $\xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d)}$; or (b) $\xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.pl_{\mathtt{loc}(d)}$. We proceed by case analysis.

In case (iii.a), we know that $\mathtt{imp}(a), \mathtt{imp}(b) \in W$ since $a, b \in W$. From this and the construction of G, we know that there exist $sf \in SF$ and $fo \in FO$ such that $\mathtt{loc}(fo) = \mathtt{loc}(\mathtt{imp}(a))$ and $sf, fo \in \xi_a$. This implies that if we assume there exist some abstract events c, d, we have $\mathtt{imp}(a) \overset{G.\mathsf{po}}{\to} fo \overset{G.\mathsf{po}}{\to} sf \overset{G.\mathsf{po}}{\to} \xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\to} \xi_b.rl_{\mathtt{loc}(d)} \overset{G.\mathsf{po}}{\to} \mathtt{imp}(b)$. Firstly, we can rewrite the ob-related events as tso-related events instead since the two ordering relations are equivalent. Therefore, we have $\mathtt{imp}(a) \overset{G.\mathsf{po}}{\to} fo \overset{G.\mathsf{po}}{\to} sf \overset{G.\mathsf{po}}{\to} \xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{tso}}{\to} \xi_b.rl_{\mathtt{loc}(d)} \overset{G.\mathsf{po}}{\to} \mathtt{imp}(b)$. Using TSO-W-FO and the fact that $\mathtt{loc}(fo) = \mathtt{loc}(\mathtt{imp}(a))$, we have $(\mathtt{imp}(a), fo) \in G.\mathsf{tso}$. In addition, from TSO-SF we have $fo \overset{G.\mathsf{tso}}{\to} sf \overset{G.\mathsf{tso}}{\to} \xi_a.wu_{\mathtt{loc}(c)}$. Furthermore, since $\xi_b.rl_{\mathtt{loc}(d)} \in L$ and $\mathtt{imp}(b) \in W$, we have $\xi_b.rl_{\mathtt{loc}(d)} \overset{G.\mathsf{tso}}{\to} \mathtt{imp}(b)$ from TSO-PO-LOCKS. Therefore, combining this information together we get $\mathtt{imp}(a) \overset{G.\mathsf{tso}}{\to} fo \overset{G.\mathsf{tso}}{\to} sf \overset{G.\mathsf{tso}}{\to} \xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{tso}}{\to} \xi_b.rl_{\mathtt{loc}(d)} \overset{G.\mathsf{tso}}{\to} \mathtt{imp}(b)$. We can condense this into $\mathtt{imp}(a) \overset{G.\mathsf{tso}}{\to} fo \overset{G.\mathsf{tso}}{\to} sf \overset{G.\mathsf{tso}}{\to} \mathtt{imp}(b)$. From NVO-LOC and using the fact that $\mathtt{loc}(fo) = \mathtt{loc}(\mathtt{imp}(a))$, we can see $\mathtt{imp}(a) \overset{G.\mathsf{nvo}}{\to} fo$. Since $fo \overset{G.\mathsf{tso}}{\to} sf \overset{G.\mathsf{tso}}{\to} \mathtt{imp}(b)$, we must also have

$fo \overset{G.\mathsf{tso}}{\rightarrow} \mathtt{imp}(b)$, and using this in conjunction with NVO-FOFL-D we get $fo \overset{G.\mathsf{nvo}}{\rightarrow} \mathtt{imp}(b)$. Therefore, we have $\mathtt{imp}(a) \overset{G.\mathsf{nvo}}{\rightarrow} fo \overset{G.\mathsf{nvo}}{\rightarrow} \mathtt{imp}(b)$, and so $(\mathtt{imp}(a), \mathtt{imp}(b)) \in G.\mathsf{nvo}$. As a result, from the definition of $G'.\mathsf{nvo}$ we have $(a, b) \in G'.\mathsf{nvo}$, as required.

In case (iii.b), we proceed in a similar way. We know that $\mathtt{imp}(a), \mathtt{imp}(b) \in W$ since $a, b \in W$. From this and the construction of G, we know that there exist $sf \in SF$ and $fo \in FO$ such that $\mathtt{loc}(fo) = \mathtt{loc}(\mathtt{imp}(a))$ and $sf, fo \in \xi_a$. This implies that if we assume there exist some abstract events c, d, we have $\mathtt{imp}(a) \overset{G.\mathsf{po}}{\rightarrow} fo \overset{G.\mathsf{po}}{\rightarrow} sf \overset{G.\mathsf{po}}{\rightarrow} \xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{ob}}{\rightarrow} \xi_b.pl_{\mathtt{loc}(d)} \overset{G.\mathsf{po}}{\rightarrow} \mathtt{imp}(b)$. Firstly, we can rewrite the ob-related events as tso-related events instead since the two ordering relations are equivalent. Therefore, we have $\mathtt{imp}(a) \overset{G.\mathsf{po}}{\rightarrow} fo \overset{G.\mathsf{po}}{\rightarrow} sf \overset{G.\mathsf{po}}{\rightarrow} \xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{tso}}{\rightarrow} \xi_b.pl_{\mathtt{loc}(d)} \overset{G.\mathsf{po}}{\rightarrow} \mathtt{imp}(b)$. From TSO-W-FO and the fact that $\mathtt{loc}(fo) = \mathtt{loc}(\mathtt{imp}(a))$, we have $(\mathtt{imp}(a), fo) \in G.\mathsf{tso}$. Next, from TSO-SF we have $fo \overset{G.\mathsf{tso}}{\rightarrow} sf \overset{G.\mathsf{tso}}{\rightarrow} \xi_a.wu_{\mathtt{loc}(c)}$. In addition, since $\xi_b.pl_{\mathtt{loc}(d)} \in L$ and $\mathtt{imp}(b) \in W$, we have $\xi_b.pl_{\mathtt{loc}(d)} \overset{G.\mathsf{tso}}{\rightarrow} \mathtt{imp}(b)$ from TSO-PO-LOCKS. Therefore, putting all of these tso-related events together, we get $\mathtt{imp}(a) \overset{G.\mathsf{tso}}{\rightarrow} fo \overset{G.\mathsf{tso}}{\rightarrow} sf \overset{G.\mathsf{tso}}{\rightarrow} \xi_a.wu_{\mathtt{loc}(c)} \overset{G.\mathsf{tso}}{\rightarrow} \xi_b.pl_{\mathtt{loc}(d)} \overset{G.\mathsf{tso}}{\rightarrow} \mathtt{imp}(b)$. This can be condensed into $\mathtt{imp}(a) \overset{G.\mathsf{tso}}{\rightarrow} fo \overset{G.\mathsf{tso}}{\rightarrow} sf \overset{G.\mathsf{tso}}{\rightarrow} \mathtt{imp}(b)$. Using NVO-LOC and the fact that $\mathtt{loc}(fo) = \mathtt{loc}(\mathtt{imp}(a))$, we have $\mathtt{imp}(a) \overset{G.\mathsf{nvo}}{\rightarrow} fo$. Since $fo \overset{G.\mathsf{tso}}{\rightarrow} sf \overset{G.\mathsf{tso}}{\rightarrow} \mathtt{imp}(b)$, we must also have $fo \overset{G.\mathsf{tso}}{\rightarrow} \mathtt{imp}(b)$, and using this along with NVO-FOFL-D we get $fo \overset{G.\mathsf{nvo}}{\rightarrow} \mathtt{imp}(b)$. Therefore, we have $\mathtt{imp}(a) \overset{G.\mathsf{nvo}}{\rightarrow} fo \overset{G.\mathsf{nvo}}{\rightarrow} \mathtt{imp}(b)$, and so $(\mathtt{imp}(a), \mathtt{imp}(b)) \in G.\mathsf{nvo}$. As a result, from the definition of $G'.\mathsf{nvo}$ we have $(a, b) \in G'.\mathsf{nvo}$, as required.

RTS. (6.19)
First, we pick an arbitrary a, b such that $(a, b) \in G'.[D]; \mathsf{st}; [P]$. This can be equivalently expressed as $(a, b) \in G'.\mathsf{st}$, with $a \in G'.D$ and $b \in G'.P$. Therefore, in order to show that $dom(G'.[D]; \mathsf{st}; [P]) \subseteq G'.P$, it suffices to show that $a \in G'.P$.

In order to show that $a \in G'.P$, we use the knowledge that $b \in G'.P$ and proceed by case analysis on the three subcomponents of $G'.P$. We also know there exists some ξ_1 such that $a, b \in \xi_1$. From this, we have: either (i) $b \in G'.I$; or (ii) $b \in PRec$; or (iii) $b \in \bigcup_{\xi \in T^i} \mathrm{p}(\xi)$. In case (i), we know that $\mathtt{tx}(b) = 0$, and so $\xi_1 = 0$ in this case. Therefore, we have $\mathtt{tx}(a) = 0$ as well and so $a \in G'.I$, which implies that $a \in G'.P$ from its construction. In case (ii), using similar reasoning we find that $b \in PRec$ implies that $b \in Rec$, given $\theta^p_{\mathtt{init}(i)} = \theta_{\mathtt{init}(i)}$ and $\theta_{\mathtt{init}(i)}.E \cap D \subseteq G.P$ also hold. Therefore, we have $\xi_1 \in T^i_{rec}$, and $b \in \theta'_{i-1}(\xi_1).E$. But then $a \in \theta'_{i-1}(\xi_1).E$ as well since a and b are both in ξ_1, and since the previous given conditions still hold, we have $a \in Rec$ as well. Therefore, $a \in PRec$ and so we have $a \in G'.P$. In case (iii), we can see that $\xi_1 \in T^i$ and $b \in \mathrm{p}(\xi_1)$. From the definition of $\mathrm{p}(\xi_1)$, this means that $b \in \theta'_i(\xi_1).E$, given $\theta_i(\xi_1).E \cap D \subseteq G_i.P$. But then $a \in \theta'_i(\xi_1).E$ as well since a and b are both in ξ_1, and since the previous given conditions still hold, we have $a \in \mathrm{p}(\xi_1)$, and so $a \in G'.P$ as required. Therefore, we have shown that $dom(G'.[D]; \mathsf{st}; [P]) \subseteq G'.P$ holds.

We now need to show that for any arbitrary $a \in G'.P$, we also have $a \in G'.T$. We recall that $G'.T$ represents the events of complete transactions in G', and we also know that there exists some ξ_1 such that $a \in \xi_1$. Therefore, we know that: either (i) $a \in G'.I$; or (ii) $a \in PRec$; or (iii) $a \in \bigcup_{\xi \in T^i} \mathrm{p}(\xi)$. In case (i), the result holds immediately from the fact that the initialisation transaction ($\xi = 0$) is always assumed to execute to completion instantaneously. In case (ii), the conditions for $PRec$ to be non-empty are that $\theta^p_{\mathtt{init}(i)} = \theta_{\mathtt{init}(i)}$ and $\theta_{\mathtt{init}(i)}.E \cap D \subseteq G.P$. From the first condition, we know that no crash occurred during the execution of $\theta^p_{\mathtt{init}(i)}$, meaning it is a full prefix of $\theta_{\mathtt{init}(i)}$, and from the second condition we have that all durable events in $\theta_{\mathtt{init}(i)}$ were able to persist. Since $PRec = Rec$ when these conditions are met, and Rec ranges over the transactions recovered in the i^{th} era, we must have that ξ_1 executes to completion and so $a \in G'.T$. In case (iii), $\mathrm{p}(\xi_1)$ is only non-empty if $\theta_i(\xi_1).E \cap D \subseteq G_i.P$ holds. With $\xi_1 \in T^i$, this can only hold if ξ_1 has been able to execute to completion and so $a \in G'.T$. Therefore, we have shown that $G'.P \subseteq G'.T$ holds. □

6.3.4 Soundness

Finally, we present the theorem of soundness for the implementation of DSI in Px86 by combining the results of Lemma 4 and Lemma 7.

Theorem 1. *(Implementation Soundness.) For all Px86-consistent execution graphs G of the implementation and their counterpart DSI execution graphs G' constructed as above:*

$$G'.((\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb}) \cap \mathsf{st}) \subseteq G'.\mathsf{po} \tag{6.20}$$

$$G'.\mathsf{hb}_{\mathsf{SI}} \textit{ is irreflexive} \tag{6.21}$$

$$G'.\mathsf{hb}_{\mathsf{SI}} \cap (D \times D) \subseteq G'.\mathsf{nvo} \tag{6.22}$$

$$dom(G'.[D]; \mathsf{st}; [P]) \subseteq G'.P \subseteq G'.T \tag{6.23}$$

PROOF. (6.20) and (6.21) follow immediately from Lemma 4. (6.22) and (6.23) follow immediately from Lemma 7. □

7 | A Non-Prescient Implementation of DSI

We now introduce an alternative implementation of DSI in Px86. The difference between this implementation and the one presented in Chapter 5 is that this implementation is *non-prescient*, meaning that we discard the assumption that the read and write sets of a given transaction are known in advance of the implementation being executed (which we previously held for the prescient DSI implementation). We use this chapter to introduce the non-prescient DSI implementation and highlight the differences between the two DSI implementations introduced in this thesis.

7.1 An Alternative Implementation of DSI

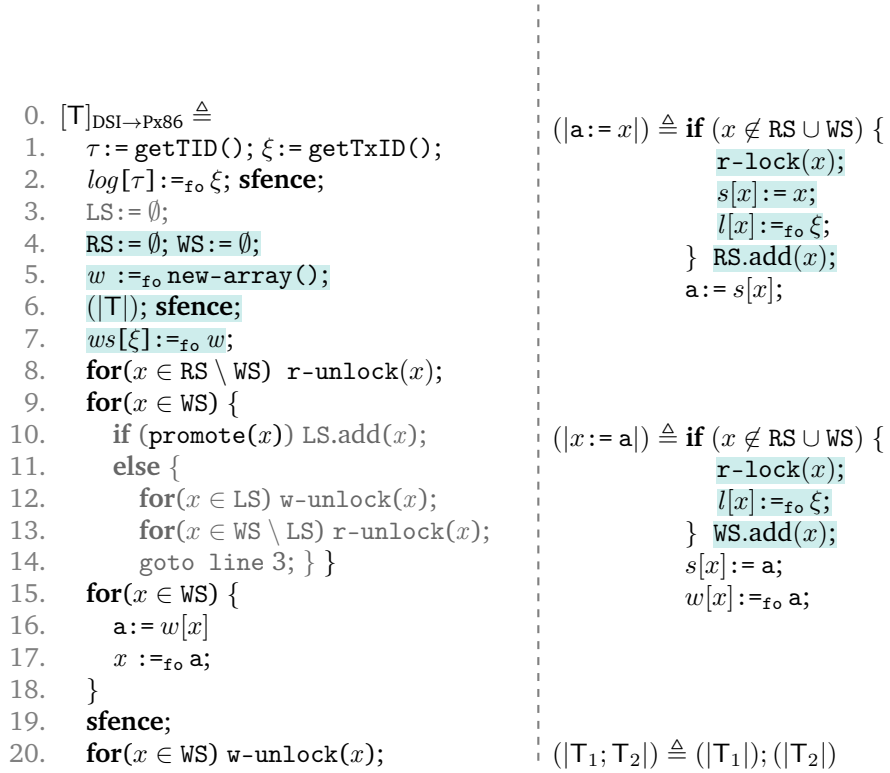

Figure 7.1: Non-prescient DSI implementation of transaction [T] in Px86, where grey code ensures deadlock avoidance. Highlighted code indicates difference between this implementation and the prescient DSI implementation from Chapter 5.

In Fig. 7.1, we present a non-prescient implementation of DSI. The non-prescience of this implementation means we no longer assume that the read set RS and write set WS of the given transaction T are provided in advance; instead, we now add locations to RS and WS "on the fly". Note that there is no recovery mechanism included in Fig. 7.1; this is because the recovery mechanism for the two implementations is identical, including the committed predicate, and so we omit these here. We spend the rest of this section

outlining the differences between the prescient and non-prescient DSI implementations by going through the code that has been highlighted in blue.

7.1.1 Local Read & Write Sets

We notice that RS and WS are now being initialised to $\emptyset$ on line 4, which was not present in the prescient implementation. This is because RS and WS are no longer assumed to be global arrays that are available throughout the execution of the implementation for T, and are instead being updated continuously throughout the local execution of T (see Section 7.1.2).

7.1.2 Absence of `snapshot`

There is no `snapshot` method defined in this implementation, since it takes RS as an argument in the prescient implementation and must therefore know what the values of RS are in advance of the implementation being executed, which is not the case here by definition of this implementation's non-prescience. However, there is still an s array in this non-prescient implementation, and we notice that this gets updated during the local execution of transactional reads; these are described in detail in Section 7.1.4. We continue to refer to s as the snapshot array, even though a snapshot is no longer being taken via one single method prior to T's local execution.

7.1.3 Local Reads of T

Transactional reads locally execute in a very different way in this non-prescient implementation. In the prescient implementation, the local execution of a read of the form a := x simply reads $s[x]$ from the snapshot taken before the local execution of T. The local execution of a read of the form a := x under this non-prescient implementation (top-right of Fig. 7.1) is as follows when the location x has not yet been added to RS or WS (i.e. $x \notin \texttt{RS} \cup \texttt{WS}$): (1) a reader lock on x is acquired by the current thread; (2) the value of x is read into the snapshot array at $s[x]$; (3) the last transaction ξ to acquire a lock on x is stored in $l[x]$; (4) x is added to RS for this transaction; and (5) the value from the snapshot array $s[x]$ is read into a. Therefore, we notice that (1) and (3) correspond to lines 5 and 6 of the prescient implementation respectively; (2) corresponds to the `snapshot` method from the prescient implementation; and (4) is a new instruction for adding x to RS since it is being read from by a. We note that (4) was unneeded in the prescient implementation since RS and WS were not being modified in that case. (5) executes regardless of whether x is already in $\texttt{RS} \cup \texttt{WS}$ or not. It is the only instruction in the local read that is not highlighted, as it coincides with the single instruction that was executed as part of local reads in the prescient implementation.

7.1.4 Local Writes of T

Transactional writes also locally execute in a different way in this non-prescient implementation. In the prescient implementation, the local execution of a write of the form x := a writes the value a to both the snapshot array s and the write log w. The local execution of a write of the form x := a under this non-prescient implementation (middle-right of Fig. 7.1) is as follows when the location x has not yet been added to RS or WS (i.e. $x \notin \texttt{RS} \cup \texttt{WS}$, as before): (1) a reader lock on x is acquired by the current thread; (2) the last transaction ξ to acquire a lock on x is stored in $l[x]$; (3) x is added to WS for this transaction; and (4) the value a is written to both the snapshot array s and the write log w. We notice that (1) and (2) correspond to lines 5 and 6 of the prescient implementation respectively; and (3) is a new instruction for adding x to WS since it is being written to. We note that (3) was unneeded in the prescient implementation since RS and WS were not being modified in that case. (4) executes regardless of whether x is already in $\texttt{RS} \cup \texttt{WS}$ or not. It corresponds to the two instructions in the local write that are not highlighted, as it coincides with the instructions that were executed as part of local writes in the prescient implementation.

7.1.5 Positioning of $(|\texttt{T}|)$

We now consider lines 5 to 7 of the non-prescient implementation. These correspond to lines 17-19 of the prescient implementation. We notice that in the case of the non-prescient implementation, this set of instructions has now moved to be *before* the deadlock avoidance mechanism in grey, whereas previously in the prescient implementation it was after this mechanism. This is due to the fact that the deadlock avoidance mechanism relies on the knowledge of which locations are in RS and WS, and so in the

non-prescient case, these values must be calculated on the fly before the deadlock avoidance code can execute. Otherwise, `RS` and `WS` will both remain as their initialised value $\emptyset$, and so no deadlock avoidance will actually be enforced. Note that in the non-prescient case, this means that deadlock avoidance takes place after the local execution of T. This is valid behaviour, since neither the local reads nor local writes are modifying the true memory locations accessed by T; therefore, write locks are promoted after the local execution of T has completed.

7.2 Soundness of Non-Prescient Implementation

We note that the soundness proof for the non-prescient DSI implementation is analogous to that of the prescient implementation, and so we omit this proof here. We discuss the extent to which we can guarantee correctness for both prescient and non-prescient implementations further in Section 8.1.

8 | Evaluation

In this chapter, we provide an evaluation of the work achieved as part of this project. We start by discussing the extent to which the DSI model can be viewed to be correct, before explaining why the work is practically useful. Finally, we comment on the novelty of the work done in this project in comparison with the state of the art in this area.

8.1 Correctness

The soundness of the prescient DSI implementation (Chapter 5) has been proved in this project. This ensures *correct* Px86-to-DSI compilation, which means that the implementation provides the guarantees of the declarative DSI model (Chapter 4). This proof was non-trivial, as indicated by its length (21 pages) as well as the amount of time needed to complete it. In addition, although the soundness proof for the non-prescient implementation (Chapter 7) was not included in this report, modifying the existing execution trace and soundness proof (i.e. those for the prescient implementation) to account for the changes incurred by the non-prescience of the second implementation would be quite simple. This is due to the fact that there is almost no difference between which *types* of events are included in the prescient and non-prescient implementations; in other words, the two implementations have roughly the same set of events in their execution traces, but the position of some of these events within the execution trace may differ depending on which implementation is being considered. Therefore, the execution trace (and, as a result, soundness proof) of the non-prescient implementation would not be very difficult to construct based on the events we have used for the soundness proof in this report.

Due to the time constraints of this project, it was not possible to prove completeness for the DSI model. This proof is non-trivial and would have been more difficult than the soundness proof, which was already a challenge. Although proving completeness would have been useful in conjunction with the soundness proof for showing equivalence between the declarative DSI model and the Px86 DSI implementation, we leave this as future work, which is discussed more in Chapter 9.

8.2 Practicality

The opportunity for practically using DSI is apparent from the design decisions that were made when developing the *implementations* for DSI that have been introduced in this report. Below, we outline some of these design decisions and their impact on the practicality of these implementations.

8.2.1 Use of Intel-x86

We chose to implement DSI using Intel-x86, a real, widely-used architecture that has been at the forefront of supporting advances in non-volatile memory in recent years. Using a mainstream architecture for the DSI implementations presented in this report has made them realistic and practical, and the way we have presented them (i.e. by replacing low-level commands with more readable shorthand notation) makes them intuitive to understand and accessible to a wider audience (e.g. programmers who do not have a background in formal methods). In particular, we note that beyond having a basic understanding of Intel-x86's persistency primitives, which are described intuitively in Section 5.1, anyone experienced with a high-level programming language should be able to understand the intuition of these implementations and how they represent durable snapshot isolation in a practical sense.

We note that previous transactional consistency models in the literature (e.g. those for snapshot isolation in [29]) have employed simple theoretical languages for presenting the implementations of their models. As a result, such works have also had to pick a consistency model for representing the non-transactional code of their implementations, and this choice can often seem arbitrary. By choosing to use a real-world architecture (Intel-x86) and its corresponding formal persistency model (Px86), these choices did not need to be arbitrary and have resulted in many practical benefits, which we outlined above.

8.2.2 Concurrency Control Mechanism

Throughout this report, we have employed a multiple-readers-single-writer (MRSW) lock library for providing concurrency control within our DSI implementations. We recall that under the snapshot isolation model, concurrent transactions are allowed to read from the same memory location simultaneously. This property of SI motivated the use of MRSW locks in our implementations, which allow multiple transactions reading from a given location to own the same lock on a given location. This provides better performance than standard locking procedures, under which a lock on a given location could only ever be owned by at most one transaction at a time, regardless of whether the location was being read from or written to.

In addition, while we have assumed the existence of this MRSW lock library without delving into the implementation details of its functions for the sake of simplifying our DSI implementations and soundness proof, we believe it would not be too difficult to implement these functions in Px86 (and update the soundness proof accordingly).Indeed, this has been done before as part of the PSER model for Px86 in [5], using the **FAA** and **CAS** atomic primitives in Intel-x86. Therefore, with the improved performance over standard locking procedures as well as the ability to implement these functions easily in Px86, the MRSW lock library is a practical and efficient way of ensuring the concurrency guarantees of durable snapshot isolation hold, and thus makes it clear that our implementations are feasible and usable.

8.2.3 Prescient & Non-Prescient Implementations

We have developed both *prescient* (Chapter 5) and *non-prescient* (Chapter 7) implementations of DSI in Px86. We use this section to outline the practical uses of both types of implementation.

The prescient implementation (Fig. 5.2) serves as a good starting point for those who wish to gain an intuition for how snapshot isolation works in the context of non-volatile memory, since the implementation is simplified by the assumption that the read and write sets of any given transaction are known in advance. Specifically, the notion of "taking a snapshot of memory" (line 9 of prescient implementation) before using this snapshot throughout the local execution of a given transaction is simpler to understand, even for those who may not be familiar with the SI model. Another benefit is that the instructions for locally executing reads and writes in a transaction (top right of Fig. 5.2) are very simple in the prescient implementation, only being made up of reads from and writes to the (volatile) snapshot array, as well as writes to the persistent write log. Furthermore, the prescient implementation may be better suited for developing a program logic for DSI transactions[29].

However, we note that although the prescient implementation is clearly useful, the non-prescient implementation is generally more practical for use on a real machine. One common result of the assumption of prescience in the context of databases is that the read and write sets of a transaction are actually *calculated* via one "pass" through the transaction. Only once this has been done for all transactions in a program can the prescient implementation of SI be executed via a second pass through. In contrast, the non-prescience of our second DSI implementation means that the read and write sets of a transaction T are calculated "on the fly" during the local execution of T, and so the assumption that the read and write sets of T are available in advance can be discarded. This removes the need for a minimum of two passes through each transaction, thus resulting in an implementation with better performance. These ideas can analogously be applied to the context of this project and DSI. Therefore, we have provided two types of implementation for DSI, each with its own share of benefits, thus accounting for the different use cases programmers may need with regards to these implementations of DSI.

8.3 Novelty

We note that this project covers *new ground* in the literature for persistency models. Specifically, the snapshot isolation transactional consistency model has not been formally explored before in the context of non-volatile memory to the best of our knowledge. The closest related work in this area has been the PSER ("persistent serialisability") model, developed by Raad et al. in [7] for the ARMv8 architecture and later in [5] for the Intel-x86 architecture. This model was undoubtedly a milestone in the literature in that it was the first formal transactional model to be developed in the context of non-volatile memory[7], and it is capable of accounting for the persistency guarantees of the given architecture while also providing ease of use to programmers via its high-level transactional interface. However, the main drawback of this model is that it enforces the transactional consistency model of serialisability, which is on the strongest end of the spectrum for these types of models. In particular, since serialisability enforces a total order on all transactions, it does not allow *any* conflicts to occur between transactions executing at the same time, thus incurring a significant slowdown in performance. In contrast, since SI allows transactions to execute concurrently and commit unless they have a write-write conflict, any read-write or read-read conflicts between concurrently executing transactions are allowed under SI and transactions are no longer forced to abort and restart whenever any conflict occurs. This leads to improved performance under SI when compared with serialisability, since transactions now have a higher chance of executing to completion and committing successfully.

9 | Conclusion

In this thesis, we have formalised the semantics of the transactional snapshot isolation model in the context of non-volatile memory by developing the durable snapshot isolation (DSI) model. We achieved this by first developing a declarative (axiomatic) model for DSI, which was done by extending an existing SI consistency model[29] with durability guarantees. We then developed a prescient reference implementation and recovery mechanism for DSI in Px86, thus demonstrating that DSI could be practically implemented on a mainstream architecture. This involved using Intel-x86-specific persistency primitives and multiple-readers-single-writer locks to make sure the persistency and consistency guarantees of DSI were maintained.

Next, we proved that the prescient DSI implementation is sound by proving a number of lemmas to show that our implementation provides both the consistency and persistency guarantees of DSI, thus demonstrating correct Px86-to-DSI compilation. Finally, we developed a non-prescient version of our DSI implementation, which is even more practically useful than the prescient implementation as it discards the assumption that the read and write sets of a transaction are known in advance. As a result of this work, we are able to present the DSI model as a persistent transactional library which is correct and also more efficient than the state of the art.

In the remainder of this chapter, we discuss possible directions for future work that builds on what was achieved in this project. We then discuss the ethical issues relating to this project.

9.1 Future Work

Transactional memory and non-volatile memory are both active research areas, meaning there are many interesting directions this project can be taken in. We discuss a few of these directions for future work below.

- **Completeness proof for DSI.** This is the clearest direction for future work, having proven soundness as part of this project. Proving completeness for the DSI model would involve showing that for every DSI-consistent abstract execution graph G', we can construct a corresponding Px86-consistent execution graph G. This would again involve constructing an execution trace, but this time it would be for the abstract graph G', which is only made up of transactional reads and writes. This trace, along with the constructed relations $G.\mathsf{rf}$, $G.\mathsf{mo}$, etc. for the implementation-level graph (which could be constructed from $G'.\mathsf{rf}$, $G'.\mathsf{mo}$, etc. respectively from the abstract-level graph) could then be used to prove completeness. It would then be possible to use the result of this completeness proof in conjunction with our soundness proof to show that the two DSI characterisations we have presented in this thesis (i.e. the declarative model and the reference implementation) are *equivalent*.
- **Weaker persistency guarantees.** The DSI model developed in this thesis guarantees strict persistency, meaning that the persistency order coincides with the consistency order (i.e. write instructions propagate to memory in the same order they become visible to other threads). It would be interesting to investigate weaker persistency guarantees for the DSI model, for which we could begin by modifying the existing declarative DSI model to allow for weaker behaviours. Specifically, the (DSI-NVO) axiom in this report's DSI model enforces a relationship between the 'happens-before' relation $\mathsf{hb}_{\mathsf{SI}}$ and the 'non-volatile order' nvo to stipulate that concurrent transactions appear to persist one after the other in the same order that they appear to have executed in, which is what enforces strict persistency for DSI. Therefore, this axiom could be investigated and modified in order to weaken the persistency guarantees of the model; this might be useful to investigate since it may allow DSI to make use of hardware-level

persistency optimisations such as out-of-order persistence, depending on the architecture that the corresponding implementation for this model would be written in.

- **Litmus tests using the Alloy solver.** Another interesting extension would be to encode the declarative DSI model in the Alloy solver[36], which we could then use to *automatically* generate persistency litmus tests (as done in [5]). While automatically generating litmus tests for *consistency* models is well understood in the literature, there are additional challenges when generating litmus tests for persistency models. Namely, the 'non-volatile order' nvo cannot be observed directly without specialist hardware that allows the memory system to be monitored. Therefore, we would need to rewrite the DSI axioms involving nvo (i.e. (DSI-NVO) and (DSI-ATOMIC)), most likely in terms of the set of persisted events P (as done for the declarative Px86 model in [5]), before trying to generate persistency litmus tests automatically.

9.2 Ethical Discussion

There are not many ethical issues to consider in this project since it is theoretical in nature. In particular, there was no use of personal data or any user testing performed as part of this project. Even so, one possible ethical issue is possible use of this work in a military application. However, this work has not been developed with any particular focus on military applications, and it would be quite unlikely for this work to be misused, so this should not pose any problems.

Another possible ethical issue, which is more relevant in the context of this project, is that this work is tied to verification and ensuring correctness, meaning that any mistakes in this work could result in real-world bugs if they are not spotted and corrected. For this reason, we have done our best to explain the reasoning behind the proofs in Chapter 6 on a fine-grained level, by omitting almost no proof cases which were analogous to cases we had already proven previously, and linking every step of our proof with the relevant axiom(s) from the declarative Px86 models, when applicable. Hence, this should not pose an issue.

A | Declarative Models for Px86

In this appendix, we provide the axioms for two declarative models that have been developed to formalise the persistency semantics of Intel-x86. The *consistency* of the first model, known as the Px86 model, is ensured by the existence of a strict order known as tso ("total store order"), and was introduced by Raad et al. [5] in the persistent setting (based on the original work of Sewell et al. [18]). Intuitively, tso describes the order in which events are made visible to other threads (i.e. the store order). Meanwhile, the consistency of the second model, which was developed later and is known as the $\text{Px86}_{\text{axiom}}$ model, is ensured by an order known as ob ("ordered-before"), and was introduced by Cho et al. in [8] in the persistent setting (again based on earlier work, by Alglave et al. [37] in this case). We note that tso is *existentially quantified*, whereas the analogous ob is *constructed*; this becomes clearer upon inspection of each model's axioms.

The equivalence of the Px86 model and the $\text{Px86}_{\text{axiom}}$ model has been proved in [8], and so tso and ob can both be used to show Px86-consistency. Due to the simple and succinct nature of the $\text{Px86}_{\text{axiom}}$ model, as well as its reduced non-determinism as a result of ob being constructed rather than being existentially quantified, we use this model as the basis of most of the proofs in this chapter. However, for a small number of proofs which involve the non-volatile order, nvo, we use the Px86 model and tso to prove our results, since the $\text{Px86}_{\text{axiom}}$ model does not model nvo explicitly, whereas the Px86 model does.

A.1 The Px86 Model (tso)

We start by providing an overview of the Px86 model[5], which relies on the tso order. Firstly, we note that the declarative Px86 model is an instantiation of the general framework for declarative persistency models presented in Chapter 3. The Px86 programming language is an extension of the general concurrent programming language provided in Fig. 3.1. The *Px86 primitive commands* are given by the following grammar:

$$\begin{aligned}\textsc{PCom}_{\text{Px86}} \ni \mathsf{c} ::= &\mathbf{load}(x) \mid \mathbf{store}(x, \mathsf{e}) \mid \mathbf{CAS}(x, \mathsf{e}, \mathsf{e}') \mid \mathbf{FAA}(x, \mathsf{e}) \\ &\mid \mathbf{mfence} \mid \mathbf{sfence} \mid \mathbf{flush}_{\text{opt}}\ x \mid \mathbf{flush}\ x\end{aligned}$$

These commands can be described as follows: $\mathbf{load}(x)$ denotes an atomic read from location x; $\mathbf{store}(x, \mathsf{e})$ denotes an atomic write to location x; $\mathbf{CAS}(x, \mathsf{e}, \mathsf{e}')$ denotes the atomic 'compare-and-swap' operation, which compares x with e and sets the value of x to e' if they match (returning 1) and leaves x unchanged otherwise (returning 0); and $\mathbf{FAA}(x, \mathsf{e})$ analogously denotes the atomic 'fetch-and-add' operation, which increments x by e and returns the old value of x. **CAS** and **FAA** are collectively known as RMW ('read-modify-write') or atomic update instructions.

We note that the $\mathbf{flush}_{\text{opt}}$, **flush** and **sfence** persistency commands were described intuitively in Section 5.1.1 and Section 5.1.2.

Next, we define the set of Px86 *labels* as follows:

$$\begin{aligned}\textsc{Lab}_{\text{Px86}} \triangleq\ &\{\mathtt{R}(x, v), \mathtt{W}(x, v), \mathtt{U}(x, v, v'), \mathtt{MF}, \mathtt{SF}, \mathtt{FO}(x), \mathtt{FL}(x) \mid x \in \textsc{Loc} \wedge v, v' \in \textsc{Val}\} \\ \cup\ &\{\mathtt{RU}(x), \mathtt{RL}(x), \mathtt{WU}(x), \mathtt{WL}(x), \mathtt{PL}(x) \mid x \in \textsc{Loc}\}\end{aligned}$$

The top line of labels given above is the original set of labels provided as part of the Px86 model. Each label has the following meaning: $\texttt{R}(x, v)$ denotes reading v from location x; $\texttt{W}(x, v)$ denotes writing v to location x; $\texttt{U}(x, v, v')$ denotes a successful update (RMW) instruction modifying x to v' when its value matches v; `MF` or `SF` denote the execution of the **mfence** or **sfence** instruction, respectively; and $\texttt{FO}(x)$ or $\texttt{FL}(x)$ denote the execution of $\textbf{flush}_{\text{opt}}\ x$ or **flush** x, respectively. The bottom line of labels denotes the extension of this original set of labels with those of the MRSW lock library given in Section 6.1.

Following on from this, we define the sets of associated events with each label provided above. The set of *read* events is: $R \triangleq \{e \in E \mid \exists x, v.\, \texttt{lab}(e) = \texttt{R}(x, v)\}$. The sets of *write* ($W$), *RMW* ($U$), *memory fence* ($MF$), *store fence* ($SF$), *optimised flush* ($FO$) and *flush* ($FL$) events are defined analogously. The set of *lock* events is defined as: $L \triangleq RU \cup RL \cup WU \cup WL \cup PL$. The set of *durable* events in E is: $D \triangleq W \cup U \cup FO \cup FL$.

In addition, under the Px86 model, the definition of the rf relation is extended from that of the general framework to account for *updates* as well. Concretely, we now define $\mathsf{rf} \subseteq (W \cup U) \times (R \cup U)$, which is total and functional on its new range (i.e. every read or update is related to exactly one write or update).

$\mathsf{mo} \subseteq \mathsf{tso}$	(TSO-MO)
tso is total on $E \setminus R$	(TSO-TOTAL)
$\mathsf{rf} \subseteq \mathsf{tso} \cup \mathsf{po}$	(TSO-RF1)
$\forall x \in \text{Loc}. \forall (w, r) \in \mathsf{rf}_x. \forall w' \in W_x \cup U_x.\, (w', r) \in \mathsf{tso} \cup \mathsf{po} \implies (w, w') \notin \mathsf{tso}$	(TSO-RF2)
$([W \cup U \cup R]; \mathsf{po}; [W \cup U \cup R] \setminus (W \times R)) \subseteq \mathsf{tso}$	(TSO-PO)
$([E]; \mathsf{po}; [MF]) \cup ([MF]; \mathsf{po}; [E]) \subseteq \mathsf{tso}$	(TSO-MF)
$([E \setminus R]; \mathsf{po}; [SF]) \cup ([SF]; \mathsf{po}; [E \setminus R]) \subseteq \mathsf{tso}$	(TSO-SF)
$([W \cup U \cup FL]; \mathsf{po}; [FL]) \cup ([FL]; \mathsf{po}; [W \cup U \cup FL]) \subseteq \mathsf{tso}$	(TSO-FL-WUFL)
$\forall X \in \text{CL}. ([FL_X]; \mathsf{po}; [FO_X]) \cup ([FO_X]; \mathsf{po}; [FL_X]) \subseteq \mathsf{tso}$	(TSO-FL-FO)
$([U]; \mathsf{po}; [FO]) \cup ([FO]; \mathsf{po}; [U]) \subseteq \mathsf{tso}$	(TSO-FO-U)
$\forall X \in \text{CL}. ([W_X]; \mathsf{po}; [FO_X]) \subseteq \mathsf{tso}$	(TSO-W-FO)
$\forall x \in \text{Loc}. \mathsf{tso}\vert_{D_x} \subseteq \mathsf{nvo}$	(NVO-LOC)
$\forall X \in \text{CL}. [W_X \cup U_X]; \mathsf{tso}; [FO_X \cup FL_X] \subseteq \mathsf{nvo}$	(NVO-WU-FOFL)
$[FO \cup FL]; \mathsf{tso}; [D] \subseteq \mathsf{nvo}$	(NVO-FOFL-D)
$[R]; \mathsf{po}; [SF] \subseteq \mathsf{tso}$	(TSO-R-SF)
$[R]; \mathsf{po}; [FO \cup FL] \subseteq \mathsf{tso}$	(TSO-SIM)

Figure A.1: The Px86 model

Now, we can provide the axioms for the Px86 model. These are given in Fig. A.1. The first six axioms (TSO-MO) – (TSO-MF) are consistency axioms for this model, and are those of the original x86-TSO consistency model defined by Sewell et al. [18]. Next, the (TSO-SF) – (TSO-W-FO) and (TSO-R-SF) – (TSO-SIM) axioms are also consistency axioms for the Px86 model, but involve events in $FO \cup FL \cup SF$ and were introduced by Raad et al. [5]. Finally, the remaining axioms (NVO-LOC) – (NVO-FOFL-D) describe the persistency semantics of Px86, and were also introduced by Raad et al..

We now extend the (TSO-PO) axiom to account for lock events, as given in Fig. A.2. This extended model is what we use in our proofs later on when dealing with the notion of Px86-consistency.

Intuitively, the (TSO-PO-LOCKS) axiom ensures that lock events cannot be reordered with respect to *any* earlier or later instructions. If this behaviour was not guaranteed, then undesirable reorderings could occur, such as e.g. a flush being reordered with a lock.

(axioms of Px86 (Fig. A.1))

$([L]; \mathsf{po}; [E]) \cup ([E]; \mathsf{po}; [L]) \subseteq \mathsf{tso}$ (TSO-PO-LOCKS)

Figure A.2: The Px86 model, extended to account for MRSW lock events

A.2 The $\text{Px86}_{\text{axiom}}$ Model (ob)

We now consider the $\text{Px86}_{\text{axiom}}$ model[8], which uses the ob order to define Px86-consistency. This model can also be viewed as an instantiation of the general declarative framework given in Chapter 3, except it does not model nvo and relies only on P (the set of persisted events) to ensure that the persistency guarantees of Intel-x86 hold.

For this model, we need to define two further relations, which are as follows:

- $\mathsf{pf} \subseteq (W \cup U) \times (FL \cup FO)$, which is the *"persists-from"* relation. This relates every flush to the mo-latest store for every location persisted by the flush. This relation is analogous to the rf relation, with the main difference being that while rf only relates a load (read) to a *single* store (write/update), pf may relate a flush to multiple stores (one for each location) on the same cache line.
- $\mathsf{pb} \triangleq \mathsf{pf}^{-1}; \mathsf{mo}$, which is the *"persists-before"* relation. This is analogous to rb and relates a flush to mo-later stores.

The programming language, labels and events for the $\text{Px86}_{\text{axiom}}$ model are the *same* as those for the Px86 model given in Appendix A.1. In addition, we note that the original model given in [8] uses a different naming convention for some of the ordering relations. For example, the modification order mo is equivalently known as the coherence order co in the paper. In our presentation of the $\text{Px86}_{\text{axiom}}$ model, we follow the naming convention we have used throughout this thesis in order to represent the model using familiar terms.

We are now ready to define the declarative $\text{Px86}_{\text{axiom}}$ model. This is given in Fig. A.3.

$$
\begin{aligned}
\mathsf{obs} &= \mathsf{mo} \cup \mathsf{rf}_e \cup \mathsf{rb}_e \\
\mathsf{dob} &= ([W \cup U \cup R]; \mathsf{po}; [W \cup U \cup R]) \setminus (W \times R) \\
\mathsf{bob} &= [W \cup U \cup R]; \mathsf{po}; [MF]; \mathsf{po}; [W \cup U \cup R] \\
\mathsf{fob} &= [W \cup U \cup R]; \mathsf{po}; [FL] \\
&\quad \cup ([U \cup R] \cup ([W]; \mathsf{po}; [MF \cup SF])); \mathsf{po}; [FO] \\
&\quad \cup [W]; (\mathsf{po}; [FL])^?; (\mathsf{po} \cap \mathsf{CL}); [FO] \\
\mathsf{ob} &= \mathsf{obs} \cup \mathsf{dob} \cup \mathsf{bob} \cup \mathsf{fob} \cup \mathsf{pf} \cup \mathsf{pb}
\end{aligned}
$$

$(\mathsf{rf}; \mathsf{po}^?)$ irreflexive (CO-RW)

$(\mathsf{rb}; \mathsf{po})$ irreflexive (CO-WR)

ob acyclic (EXTERNAL)

$\mathsf{pf} \subseteq (\mathsf{obs} \cup \mathsf{dob} \cup \mathsf{bob} \cup \mathsf{fob} \cup \mathsf{pb})^+$ (PF-MIN)

$$P = dom(\mathsf{pf}; ([FL] \cup ([FO]; \mathsf{po}; [MF \cup SF \cup U])))$$

$\forall l. \exists w.\, SM(l) = \mathtt{wval}(w) \wedge (P \times \{w\}) \cap \mathrm{Loc} \subseteq \mathsf{mo}^?$ (PERSIST)

Figure A.3: The $\text{Px86}_{\text{axiom}}$ model

Intuitively, the inclusion of the obs, dob and bob relations as part of ob, along with the (CO-RW), (CO-WR) and (EXTERNAL) axioms, ensure that the consistency guarantees of $\text{Px86}_{\text{axiom}}$ are upheld. The persistency guarantees of Px86 are taken care of by the inclusion of the fob, pf and pb relations in ob, as well as the (PERSIST) axiom. The (PF-MIN) axiom is optional, but also handles the persistency side of this model.

We extend the $\text{Px86}_{\text{axiom}}$ model to account for lock events (as we did for the Px86 model) by redefining the dob relation as follows:

$$
\begin{aligned}
&(\text{axioms of } \text{Px86}_{\text{axiom}} (\text{Fig. } A.3)) \\
&\mathsf{lob} = [L]; \mathsf{po}; [E]) \cup ([E]; \mathsf{po}; [L]) \\
\mathsf{ob} =& \mathsf{obs} \cup \mathsf{dob} \cup \mathsf{bob} \cup \mathsf{fob} \cup \mathsf{pf} \cup \mathsf{pb} \cup \mathsf{lob}
\end{aligned}
\qquad \text{(REDEFINED-OB)}
$$

Figure A.4: The $\text{Px86}_{\text{axiom}}$ model, extended to account for MRSW lock events

We use this extended model extensively throughout the proofs in this thesis.

Bibliography

[1] Raad A, Lahav O, Vafeiadis V. On Parallel Snapshot Isolation and Release/Acquire Consistency. In: Ahmed A, editor. Programming Languages and Systems. Cham: Springer International Publishing; 2018. p. 940–967.

[2] The Clojure Language: Refs and Transactions; 2022. Available from: https://clojure.org/reference/refs.

[3] Haskell: Software Transactional Memory package; 2022. Available from: https://hackage.haskell.org/package/stm-2.2.0.1.

[4] ScalaSTM; 2022. Available from: https://nbronson.github.io/scala-stm/.

[5] Raad A, Wickerson J, Neiger G, Vafeiadis V. Persistency Semantics of the Intel-X86 Architecture. Proc ACM Program Lang. 2019 dec;4(POPL). Available from: https://doi.org/10.1145/3371079.

[6] Pelley S, Chen PM, Wenisch TF. Memory Persistency. SIGARCH Comput Archit News. 2014 jun;42(3):265–276. Available from: https://doi.org/10.1145/2678373.2665712.

[7] Raad A, Wickerson J, Vafeiadis V. Weak Persistency Semantics from the Ground up: Formalising the Persistency Semantics of ARMv8 and Transactional Models. Proc ACM Program Lang. 2019 oct;3(OOPSLA). Available from: https://doi.org/10.1145/3360561.

[8] Cho K, Lee SH, Raad A, Kang J. Revamping Hardware Persistency Models: View-Based and Axiomatic Persistency Models for Intel-X86 and Armv8. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2021. New York, NY, USA: Association for Computing Machinery; 2021. p. 16–31. Available from: https://doi.org/10.1145/3453483.3454027.

[9] Raad A, Vafeiadis V. Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model. Proc ACM Program Lang. 2018 oct;2(OOPSLA). Available from: https://doi.org/10.1145/3276507.

[10] Boehm HJ, Chakrabarti DR. Persistence Programming Models for Non-Volatile Memory. SIGPLAN Not. 2016 jun;51(11):55–67. Available from: https://doi.org/10.1145/3241624.2926704.

[11] Chakrabarti DR, Boehm HJ, Bhandari K. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. SIGPLAN Not. 2014 oct;49(10):433–452. Available from: https://doi.org/10.1145/2714064.2660224.

[12] Intel Optane; 2021. Available from: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[13] Lee BC, Ipek E, Mutlu O, Burger D. Architecting Phase Change Memory as a Scalable Dram Alternative. SIGARCH Comput Archit News. 2009 jun;37(3):2–13. Available from: https://doi.org/10.1145/1555815.1555758.

[14] Kawahara T, Ito K, Takemura R, Ohno H. Spin-transfer torque RAM technology: Review and prospect. Microelectronics Reliability. 2012;52(4):613–627. Advances in non-volatile memory technology. Available from: https://www.sciencedirect.com/science/article/pii/S002627141100446X.

[15] Strukov D, Snider G, Stewart D, Williams S. The Missing Memristor Found. Nature. 2008 06;453:80–3.

[16] Memory Consistency (Lecture Slides). University of Edinburgh School of Informatics; 2018. Available from: http://www.inf.ed.ac.uk/teaching/courses/pa/Notes/lecture07-sc.pdf.

[17] Adve SV, Boehm HJ. In: Padua D, editor. Memory Models. Boston, MA: Springer US; 2011. p. 1107–1110. Available from: https://doi.org/10.1007/978-0-387-09766-4_419.

[18] Sewell P, Sarkar S, Owens S, Nardelli FZ, Myreen MO. X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors. Commun ACM. 2010 jul;53(7):89–97. Available from: https://doi.org/10.1145/1785414.1785443.

[19] Pulte C, Flur S, Deacon W, French J, Sarkar S, Sewell P. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8. Proc ACM Program Lang. 2017 dec;2(POPL). Available from: https://doi.org/10.1145/3158107.

[20] Batty M, Owens S, Sarkar S, Sewell P, Weber T. Mathematizing C++ Concurrency. SIGPLAN Not. 2011 jan;46(1):55–66. Available from: https://doi.org/10.1145/1925844.1926394.

[21] Lahav O, Vafeiadis V, Kang J, Hur CK, Dreyer D. Repairing Sequential Consistency in C/C++11. SIGPLAN Not. 2017 jun;52(6):618–632. Available from: https://doi.org/10.1145/3140587.3062352.

[22] Raad A. Sequential Consistency (Lecture Slides). The Theory & Practice of Concurrent Programming. Imperial College London; 2021.

[23] Raad A. Total Store Ordering (Lecture Slides). The Theory & Practice of Concurrent Programming. Imperial College London; 2021.

[24] Condit J, Nightingale EB, Frost C, Ipek E, Lee B, Burger D, et al. Better I/O through Byte-Addressable, Persistent Memory. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. SOSP '09. New York, NY, USA: Association for Computing Machinery; 2009. p. 133–146. Available from: https://doi.org/10.1145/1629575.1629589.

[25] Bernstein PA, Hadzilacos V, Goodman N. Concurrency Control and Recovery in Database Systems. USA: Addison-Wesley Longman Publishing Co., Inc.; 1986.

[26] What is a Transaction?. Microsoft Documentation; 2021. Available from: https://docs.microsoft.com/en-gb/windows/win32/ktm/what-is-a-transaction?redirectedfrom=MSDN.

[27] Litz H, Dias RJ, Cheriton DR. Efficient Correction of Anomalies in Snapshot Isolation Transactions. ACM Trans Archit Code Optim. 2015 jan;11(4). Available from: https://doi.org/10.1145/2693260.

[28] Berenson H, Bernstein P, Gray J, Melton J, O'Neil E, O'Neil P. A Critique of ANSI SQL Isolation Levels. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data. SIGMOD '95. New York, NY, USA: Association for Computing Machinery; 1995. p. 1–10. Available from: https://doi.org/10.1145/223784.223785.

[29] Raad A, Lahav O, Vafeiadis V. On the Semantics of Snapshot Isolation. In: Enea C, Piskac R, editors. Verification, Model Checking, and Abstract Interpretation. Cham: Springer International Publishing; 2019. p. 1–23.

[30] Concurrency Anomalies. Telerik OpenAccess Classic Documentation; 2013. Available from: https://docs.telerik.com/help/openaccess-classic/concurrency-control-anomalies.html.

[31] Cerone A, Gotsman A. Analysing Snapshot Isolation. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing. PODC '16. New York, NY, USA: Association for Computing Machinery; 2016. p. 55–64. Available from: https://doi.org/10.1145/2933057.2933096.

[32] Kolli A, Gogte V, Saidi A, Diestelhorst S, Chen PM, Narayanasamy S, et al. Language-Level Persistency. SIGARCH Comput Archit News. 2017 jun;45(2):481–493. Available from: https://doi.org/10.1145/3140659.3080229.

[33] Gogte V, Diestelhorst S, Wang W, Narayanasamy S, Chen PM, Wenisch TF. Persistency for Synchronization-Free Regions. SIGPLAN Not. 2018 jun;53(4):46–61. Available from: https://doi.org/10.1145/3296979.3192367.

[34] Adya A, Liskov B, O'Neil P. Generalized isolation level definitions. In: Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073); 2000. p. 67–78.

[35] Intel® 64 and IA-32 Architectures Software Developer Manuals; 2022.

[36] Milicevic A, Near JP, Kang E, Jackson D. Alloy*: A General-Purpose Higher-Order Relational Constraint Solver. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 1; 2015. p. 609–619.

[37] Alglave J, Maranget L, Tautschnig M. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. ACM Trans Program Lang Syst. 2014 jul;36(2). Available from: https://doi.org/10.1145/2627752.