Deciding Reachability under Persistent x86-TSO

PAROSH AZIZ ABDULLA, Uppsala University, Sweden
MOHAMED FAOUZI ATIG, Uppsala University, Sweden
AHMED BOUAJJANI, University of Paris, France
K. NARAYAN KUMAR, Chennai Mathematical Institute and CNRS UMI RelaX, India
PRAKASH SAIVASAN, The Institute of Mathematical Sciences, India

We address the problem of verifying the reachability problem in programs running under the formal model Px86 defined recently by Raad et al. in POPL'20 for the persistent Intel x86 architecture. We prove that this problem is decidable. To achieve that, we provide a new formal model that is equivalent to Px86 and that has the feature of being a well structured system. Deriving this new model is the result of a deep investigation of the properties of Px86 and the interplay of its components.

CCS Concepts: • **Theory of computation → Verification by model checking**; • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: model checking, program verification, TSO memory model, persistent memories

ACM Reference Format:
Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. 2021. Deciding Reachability under Persistent x86-TSO. *Proc. ACM Program. Lang.* 5, POPL, Article 56 (January 2021), 32 pages. https://doi.org/10.1145/3434337

1 INTRODUCTION

Emerging *Non-Volatile Random-Access Memories (NVRAM)* provide the best of two worlds, namely the efficiency of DRAM, and the data persistency across failures of a non-volatile store [Intel 2019c; Liu et al. 2020]. In particular, NVRAMs offer byte addressability, i.e., they can be addressed directly by the CPU through read and write operations, thus giving programs direct and low-latency access to persistent data, without any assistance from the operating system. Upon a crash, the volatile state of the system, including the DRAM content and process registers are lost while, in contrast, the NVRAM state is preserved. This is why NVRAMs are often referred to as *Persistent Memories (PMs)*. The combination of efficiency and persistency is a very attractive feature, breeding interest in PMs, and spurring research both in industry and academia [ARM 2018; Intel 2019b; Liu et al. 2020; Raad et al. 2020, 2019]. Leading chip manufacturers such as Intel and ARM have started to integrate the technology in their chips [ARM 2018; Intel 2019b]. Developers are creating systems that directly manipulate PMs such as data bases [Arulraj and Pavlo 2017], key-value stores [Xia et al. 2017], and custom programs [Cohen et al. 2018].

Authors' addresses: Parosh Aziz Abdulla, Uppsala University, Sweden, parosh@it.uu.se; Mohamed Faouzi Atig, Uppsala University, Sweden, mohamed_faouzi.atig@it.uu.se; Ahmed Bouajjani, University of Paris, France, abou@irif.fr; K. Narayan Kumar, Chennai Mathematical Institute and CNRS UMI RelaX, India, kumar@cmi.ac.in; Prakash Saivasan, The Institute of Mathematical Sciences, India, prakashs@imsc.res.in.

2475-1421/2021/1-ART56
https://doi.org/10.1145/3434337

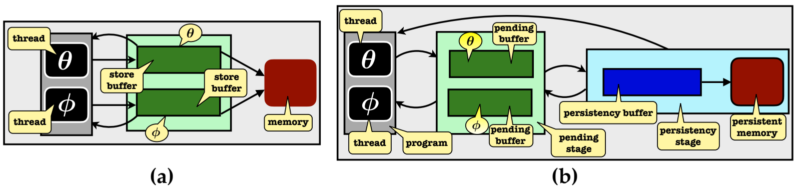


Fig. 1. (a) The classical TSO semantics. (b) The Px86 (persistency) semantics [Raad et al. 2020].

The support for recoverability offered by PMs comes at a price, namely, that the programmer needs to provide a *recovery code* which retrieves the data that has persisted since before the crash, and use this data to recreate a state from which the computation can continue. Consistent recovery making the occurrence of crashes completely transparent to the applications is hard to ensure, and in general, while PMs allow to recover some data, the content of the recovery state may not be consistent with the program view. In fact, many systems that implement persistency allow write operations to *persist*, i.e., become persistent, in an order which may differ from the order in which they are issued by the program, which may lead to observing inconsistent states. To help ensure consistent recovery, specific *persistency barriers* are used to restrict the persistency order, i.e., to oblige the write operations to persist in a given order.

The consistency issue makes the use of PM architectures extremely hard, and therefore, there is a crucial need of formal methods for reasoning in a principled way about their behaviors, as well as of tools for automated program verification, in order to help programmers develop their applications and ensure their correctness.

A first step towards this end is to define abstract *memory models* of PM's that prescribe faithfully their visible behaviors in reaction to any sequence of write and read operations performed by their users. A PM architecture is typically designed as an extension of some existing architecture with an additional persistency mechanism. Then, the memory model of a PM (or its semantics) is determined by a combination of both the *consistency model* corresponding to the semantics of the original architecture, and the *persistency model* that defines the order in which write instructions may persist [Pelley et al. 2014]. A case in point is the recent Intel chip that augments the x86 architecture with persistent memory [Intel 2019a], resulting in an extension of the classical Total Store Order (TSO) semantics [Sewell et al. 2010]. The manner in which data is persisted in the chip does not conform to TSO, and program runs, along which crashes occur, follow a weaker memory model than TSO. This is reflected in the operational model defined in [Raad et al. 2020], called there Px86. Fig. 1 (a) and (b) depicts the classical and persistency x86-TSO models respectively.

In the classical TSO model, FIFO store buffers, one per thread, are used to store write operations before they are committed to the main memory. Write operations in a store buffer are visible only to the thread that issued them, and they become visible to all threads only when they are committed to the memory. This has the effect of delaying their execution, and allows read operations (on different variables) to overtake them. This is the only type of re-ordering allowed in that model. Issuing a memory barrier by a thread has the effect of blocking the execution of that thread until its buffer is empty, ensuring that all prior writes are made visible before issuing any further instructions.

The Px86 model defined in [Raad et al. 2020] is much more complex than TSO, with a hierarchical memory system, and different flush and barrier operations that may be used to constrain instruction re-orderings in various ways. Its architecture uses a two-stage buffer system, a *pending stage* and a *persistency stage*, in addition to the persistent memory. In a similar manner to the classical model, the pending stage consists of buffers, one for each thread, carrying write operations issued by the corresponding thread and yet not visible to the other threads. Once an operation reaches the end of a pending buffer, it can cross to the persistency stage. The latter consists of a single buffer that is visible to all the threads. When a crash occurs, the content of the memory is preserved while all operations in the buffers of both stages are lost. In contrast with TSO, the Px86 model has the following features that make it particularly complex:

• The buffers in both stages may contain, in addition to write operations, other types of operations such as flush and fence instructions that are used to constrain allowed re-orderings in different ways. The number of operations of each type can be unbounded.

• Several types of re-orderings may occur inside the buffers in both stages, and hence these buffers do not behave according to the FIFO discipline. For instance, while write instructions are not re-ordered with each other in the pending stage, they might be re-ordered with flush instructions, and once they cross to the persistency stage, write instructions on different variables can be re-ordered.

• The memory system in the persistency model is *hierarchical* in the sense that the write operations in the pending buffers are not used to update the shared memory directly. A thread may need to consult the persistency stage, or even the persistent memory, to fetch the value of a shared variable.

Once formal models of PMs have been defined, the crucial issue is to develop algorithmic approaches for formal verification of programs running on PMs. For that, investigating the decidability and complexity of fundamental verification problems such as verifying safety properties (or dually state reachability), is extremely important.

Program verification under weak memory models has proven to be a challenging task. Indeed, the ability of these models to re-order operations leads in general to a high computational power. Intuitively, this re-ordering amounts to using various types of unbounded buffers, and therefore verification problems may become in general either highly complex or even undecidable. For instance, it has been shown that the state reachability problem is decidable for (the classical) TSO [Abdulla et al. 2018a; Atig et al. 2010, 2012], while it is undecidable for other models such as Power [Abdulla et al. 2020]. The models obtained by combining the consistency models of the original architecture with persistency models become much more complicated, with several additional sources of re-orderings. As far as we know, the decidability and complexity issues have never been addressed in the presence of persistency. This paper takes the first step in investigating these issues. We address the state reachability problem for finite-state concurrent programs running on the top of the persistency TSO model defined in [Raad et al. 2020]. Notice that in [Raad et al. 2020], two slightly different versions of the persistency TSO model are defined called Px86$_{man}$ and Px86$_{sim}$. The Px86$_{man}$ conforms to the behaviors described (informally) in the manual of the persistent x86 architecture, but actually allows more behaviors than the real architecture, and does not capture precisely the intents of its designers. Px86$_{sim}$ is the formal model that is faithful to the behaviors of the architecture. In this paper we focus on the latter model, and from now on, we implicitly refer to Px86$_{sim}$ when we talk about the *persistency x86/TSO model* (or simply Px86).

Despite the complexity of the Px86 model, we prove in this paper that the reachability problem for finite-state programs under that model is decidable. We show that this problem is reducible to a decidable reachability problem in the well-known framework of *well-structured transition systems* (WSTS) [Abdulla et al. 1996; Finkel and Schnoebelen 2001]. This is a generic framework for proving decidability of reachability problem of infinite-state systems. The fundamental theorem established in this framework is that, for any given system, if (1) it is possible to prove that there is a *well-quasi ordering* (WQO) on the states (i.e., an ordering such that for every infinite set of states there are necessarily at least two comparable elements) for which it is possible to prove that the transition relation of the system is *monotone* (i.e., if a state s_1 has a transition by some action a to a state s'_1, then every larger state s_2 has a transition by a to some state s'_2 that is larger than s_2), and if (2) computing predecessors of any state can be done effectively, then the standard iterative backward reachability analysis algorithm is guaranteed to terminate, and therefore the reachability problem of a given state (and more generally to upward-closed sets of states w.r.t. to the considered WQO) is decidable [Abdulla et al. 1996; Finkel and Schnoebelen 2001]. This framework has been applied to many types of infinite-state systems in the literature, including FIFO-channel systems for instance. (Of course, well-structuredness is a sufficient condition for decidability, not a necessary one.)

However, the application of the framework of WSTS to the persistency semantics is not easy. Indeed, a good strategy for establishing well-stucturedness in this context is in general based on the quest of a number of desirable *"criteria of well-behavior"*, that are not satisfied by the persistency semantics. The main criterion is, as we have mentionned above, that the transition system induced by the program is *monotone* with respect to a WQO on the data structure [Abdulla and Jonsson 1993]. This means in our case that if we insert an additional message in a given buffer then, from the new (larger) configuration, the system is able to perform at least the same sequence of transitions as from the old (smaller) configuration. Clearly, the behavior of the persistency semantics (according to notions of states and transitions of [Raad et al. 2020]) is not monotone since adding write, flush or fence instructions to buffers may restrict the behavior of the system.

A second important criterion for showing well-structuredness is that buffers are FIFO, since this allows to use the natural sub-word well-quasi ordering on these structures. However, as we have seen, the buffers in Px86 are *not* FIFO; they allow selective re-orderings of messages, and they may contain unbounded numbers of messages of different types (write, flushes, and fences). Another criterion of well-behavior that is important for showing monotonicity is *locality*, in the sense that read operations are performed locally on each buffer, without interactions with other buffers. This is not satisfied in the persistency model due to the memory hierarchy.

In the classical TSO model, buffers are FIFO and reads are local. The decidability proof in that case can be done using a translation of the semantics to an equivalent monotone semantics w.r.t. the sub-word ordering. In the case of Px86, such a single step translation is extremely difficult, if not impossible, due to the additional difficulties mentioned above (non-FIFO buffers, and the non-locality of reads). Therefore the challenge is to provide a new formal model that is equivalent to the persistency TSO semantics defined in [Raad et al. 2020], and which falls under the class of systems to which the WSTS framework can be applied. Notice that, in order to achieve decidability, it is necessary to investigate semantical aspects and define a model with particular properties that make the proof of well-structuredness possible, and our approach for that is to enforce (while preserving equivalence) the criteria of well-behavior mentioned above. Besides decidability, this investigation has also the interest to provide a better insight about the features of this model, and its outcome is valuable also from the semantical point of view as it provides a formal model that could be of simpler use for reasoning about programs.

We perform a sequence of translations, ultimately obtaining a well-structured model that is equivalent to the Px86 model of [Raad et al. 2020]. The new model is called the *scheduling semantics* and is obtained in two steps to make the presentation of the semantics, and the proof of its soundness and completeness, easier. First we tackle the issues of locality and FIFO order in the buffers (introducing what we call the *basic* scheduling semantics), and then, we show how it is possible to obtain monotonicity (by providing what we call the *refined* scheduling semantics).

Locality is obtained by introducing an intermediate memory between the pending and the persistency stages. The information in the persistency stage is needed only when a crash occurs to determine the re-starting state of each thread and the contents of the intermediary memory.

A more challenging issue is to define an equivalent model where only FIFO buffers are used, possibly with a finitely encodable additional information. Concerning the pending buffers, to deal with the fact that flush instructions can be re-ordered forwards and backwards, we introduce two flags, namely a *Delay* and a *Promise* flag for each thread and variable indicating whether a flush has been delayed (moved backwards) or promised (moved forwards) in the corresponding pending buffer. We prove that, surprisingly, using only these additional Boolean flags, it is possible to simulate precisely the original model in the presence of an unbounded number of flush reorderings. Concerning the persistency stage, we split the persistency buffer into a system of several initial persistency buffers, one per variable, and an additional final persistency buffer where write instructions are drained from the initial buffers before being committed to the persistent memory. Having one buffer per variable allows to re-order write instructions on different variables, and the final buffer allows to simulate correctly the persistency barriers semantics. The architecture of the so defined model, called basic scheduling semantics, is pictured in Figure 2.

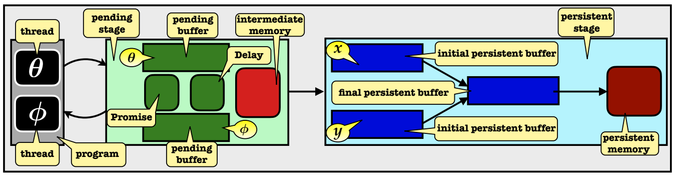


Fig. 2. The architecture of the basic scheduling semantics.

Then, the second step of our reduction is to translate the basic model described above to a monotone one. For that, each stage needs to be handled differently. Concerning the pending stage, the idea is to use a (dual) load-buffer semantics based on FIFO load buffers instead of store buffers following the approach introduced in [Abdulla et al. 2018a], although technically the reduction is quite different since we have to deal with other types of messages in the buffers such as flush instructions. In the load-buffer semantics, write instructions are executed in the moment they are issued to the main memory, and then, they flow to the load buffers from which threads can fetch values to read. In addition, we maintain information about at most one flush instruction per variable in each thread load buffer, and show that this suffices to simulate the basic scheduling semantics. This fact is crucial for obtaining a monotone model. Concerning the persistency stage, getting monotonicity is somehow easier. In fact, the source of non-monotonicity comes only from the final persistency buffer, not from the initial persistency buffers. To handle this case, we transform write instructions in the final persistency buffer into *memory snapshots*, in the spirit of the techniques employed in [Atig et al. 2010, 2012]. From this reduction, we deduce that the verification problem of the reachability problem in finite-state programs is decidable. We can also show that the complexity of this problem is non-primitive recursive by a reduction from the same problem for TSO, for which the complexity has been established in [Atig et al. 2010].

To summarize, we define a new operational formal semantics that is equivalent to the Px86 model [Raad et al. 2020] up to state reachability, and which has the property of defining a well-structured system. This model is based on establishing a clear separation between the pending and persistency stages. This compositional view of the Px86 model, and the ability of using only FIFO buffers in the modelling, are the key for proving the decidability of the state reachability problem.

Related Work. Formal models of PMs have been proposed recently in, e.g., [Raad and Vafeiadis 2018; Raad et al. 2020, 2019]. These works follow the line of research investigating formal models of weakly consistent systems, defining axiomatic and/or operational models for hardware architectures [Alglave et al. 2014; Flur et al. 2016; Mador-Haim et al. 2012; Sewell et al. 2010], concurrent programming languages [Batty et al. 2011; Kang et al. 2017; Lahav et al. 2016; Nienhuis et al. 2016; Podkopaev et al. 2019], and distributed systems [Burckhardt 2014].

Decidability and complexity of program verification under weak consistency have been investigated in the last decade for hardware architectures models [Abdulla et al. 2020; Atig et al. 2010, 2012], concurrent programming languages models [Abdulla et al. 2019], and for distributed replicated data structures and data bases [Lahav and Boker 2020]. Many other works addressing effective program verification under weak consistency have been carried out, e.g., [Abdulla et al. 2017a, 2018b; Alglave et al. 2013b,a; Atig et al. 2011; Demsky and Lam 2015; Gotsman et al. 2016; Kokologiannakis et al. 2018; Kokologiannakis and Vafeiadis 2020]. All the mentioned works do not address the case of PMs. So far, works on automated verification of programs running on PMs are rare. The papers [Liu et al. 2020, 2019] propose a method for detecting *cross-failures*, i.e., checking whether persistent data remain consistent across a failure. The papers identify necessary conditions for consistent recoveries, i.e., to decide whether the data that persist *before* a failure, is used to take the program to a consistent state *after* the failure.

The framework of well structured systems is a general framework for establishing decidability results for infinite-state systems. It has been traditionally applied to various types of systems such as Petri nets, unreliable communication channels, register systems, parametrized networks of processes, etc. It has been also applied to weak memory models in [Atig et al. 2010, 2012]. It has never been applied so far to the case of PM models.

A new equivalent model to Px86 model [Raad et al. 2020] has been developed independently by Khyzha and Lahav [Khyzha and Lahav 2021].

2 OVERVIEW

We will give a step-by-step introduction to the scheduling semantics, using a sequence of simple examples. We will start by recalling the persistency x86/TSO semantics of [Raad et al. 2020]. For the latter, we also highlight the aspects of the semantics that make it hard to prove decidability of the reachability problem. For simplicity, we will assume that cache lines are not shared by variables. In Sec. 2.4 we explain briefly how we can extend our framework, in a straightforward manner, to deal with the general case where cache lines are shared by multiple variables. We define the scheduling semantics in two steps, namely the *basic*, followed by the *refined* (full) semantics.

2.1 The Persistency Semantics

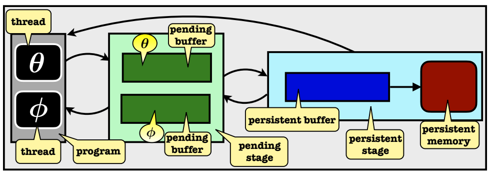


Fig. 3. The architecture of the persistency semantics as defined in [Raad et al. 2020].

The operational semantics of persistency TSO was recently proposed by Raad et. al [Raad et al. 2020]. The semantics, which we simply refer to as the *persistency semantics* in the sequel, generalizes the classical TSO semantics. Fig. 3 depicts the architecture of the persistency semantics for the case of a program with two threads θ and ϕ. The semantics uses a two-stage buffer system, which we refer to as the *pending stage*, and the *persistency stage*. The pending stage consists of unbounded buffers, which we call the *pending buffers*, one for each thread. The pending buffer of a thread θ carries messages representing instructions that have been executed by θ, but that are yet not visible to the other threads. Messages can be of different types, e.g., write, flush, or fence messages, reflecting the instructions that have generated them. A message in the pending may eventually cross to the persistency stage, possibly overtaking other messages ahead of it in the buffer. The latter consists of a single buffer, which we call the *persistency buffer*. The persistency buffer is shared by (and is visible to) all the threads and it sends its messages to the *persistent memory*. We say that a write message *persists* when it reaches the persistent memory. A message that persists will survive a system crash, unless it is overwritten in the persistent memory by another message. On the other hand, all messages traveling in the buffers will be lost if a crash takes place. After crash, a *recovery* procedure, provided by the programmer, can retrieve the persistent data and use it to re-start the program from an appropriate configuration.

2.1.1 Configurations. Fig. 4 illustrates a typical configuration in the persistency semantics. For ease of reading, we will omit the local states of the threads from our figures in this section. The pending buffer of θ contains a *store fence* message sf, followed[1] by a write message assigning 1 to x, denoted x^1. The pending buffer of ϕ contains a write message assigning 1 to y, followed by a flush message on the variable x, denoted fo^x. The persistency buffer contains a barrier message on x, denoted per^x, followed by a write message on y. The values of x and y in the persistent memory are 3 and 0 respectively. If a crash occurs in this configuration, then these two values will be retained. We will explain the roles of sf, fo, and per later.

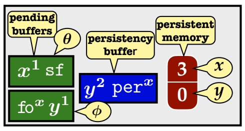


Fig. 4. A configuration in the persistency semantics.

2.1.2 Memory Hierarchy. The memory system of the persistency semantics is *hierarchical* in the sense that write messages in the pending buffers are not used to update the shared memory directly. This makes read instructions *non-local.* A read instruction can traverse the whole memory hierarchy: when reading from a variables x, the thread first inspects its own pending buffer and reads the value of the last buffered write to x if such a write exists (this is similar to the classical TSO semantics); otherwise, it searches the persistency buffer for the value of the last buffered write to x if such a write exists; otherwise, it reads x from the persistent memory. In Fig. 4, read instructions by the thread θ on the variables x and y would obtain the values 1 (from its pending buffer), resp. 2 (from the persistency buffer). Analogously, the thread ϕ would obtain the values 3 (from the persistent memory) and 1 (from its pending buffer).

2.1.3 Re-orderings. Write messages exhibit a mixture of FIFO and non-FIFO behaviors in the persistency semantics. They are ordered inside the pending buffers but, once inside the persistency buffer, write messages on different variables can be re-ordered. Fig. 5 illustrates simple program, and a run of the program under the persistency semantics. In γ_1, the thread has performed its two instructions, and the corresponding messages, i.e., x^1 and y^1 are in the pending buffer. The two messages cannot be re-ordered inside the pending buffer, and hence they cross to the persistency buffer in the same order, i.e., x^1 followed by y^1 (configuration γ_2). However, once inside the persistency buffer, the messages can be re-ordered. This means that, although x^1 was generated before y^1, the latter can persist first. The assertion at the end of program says when the value of y is 1 in the persistent memory then the value of x can be 0 or 1.

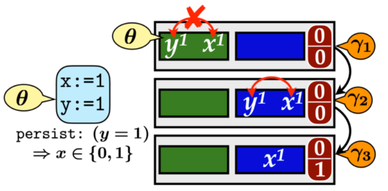


Fig. 5. Re-ordering of write messages in the persistency buffer.

2.1.4 Flush Operations. To limit the re-orderings that allow write instructions to persist differently from the order in which they were issued, the persistency architecture provides different kinds of flush/fence instructions. Fig. 6 illustrates the flush instruction $\mathsf{flush}_{\mathsf{opt}}$, referred to as *cache line flush optimized* in the Intel Manual [Intel 2019a; Raad et al. 2020]. When a thread θ executes $\mathsf{flush}_{\mathsf{opt}} x$, a corresponding message fo^x is added to the pending buffer of θ. When the message crosses to the persistency buffer, it is transformed to a message per^x. The latter acts as a *barrier* in the

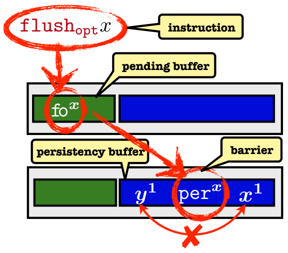


Fig. 6. The $\mathsf{flush}_{\mathsf{opt}}$ instruction.

[1]Later, we will formally represent a write message as a triple $\langle \mathsf{wr}, x, 1\rangle$. To simplify the figures in this section, we depict it as x^1. We will use a similar notation for other types of messages when we introduce the persistency and scheduling semantics in the subsequent sub-sections.

sense that it prevents write messages on x that lie before it in the persistency buffer to be overtaken by write messages on other variables that lie after it in the buffer (although per^x can overtake a write message on different variable y.) The message per^x *disappears* when it reaches the end of the persistency buffer without affecting the persistent memory.

2.1.5 Overtaking flush$_{\text{opt}}$*.* While the fo message transforms to a barrier per in the persistency buffer, it behaves weakly in the pending buffer, in the sense that it can be overtaken by (and also overtake) other messages. Consequently, the flush$_{\text{opt}}$ instruction on a variable x is too weak to ensure that a write instruction on x persists before a later write instruction on another variable y. In γ_1 of Fig. 7, θ has executed all of its three instructions, namely two write instructions on x resp. y, with a flush$_{\text{opt}}$ instruction on x in between; and has put the corresponding messages in the pending buffer. A write message is allowed to *overtake* an fo message in the pending buffer. This means that we can reach the configuration γ_2 where the write messages on x and y have both crossed to the persistency buffer, while the fo^x is still in the pending buffer. Since write messages on different variables can overtake each other inside the persistency buffer, the message y^1 can persist before the message x^1 (configuration γ_3).

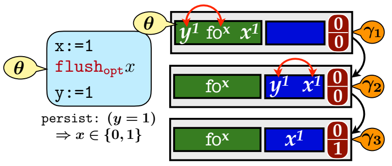


Fig. 7. Combining flush$_{\text{opt}}$ with plain write instructions.

2.1.6 Strengthening flush$_{\text{opt}}$*.* Given the weak behavior of the flush$_{\text{opt}}$ instruction in the pending stage, we need to combine it with other instructions such as the *store fence* sfence instruction. The sfence and flush$_{\text{opt}}$ combine together to enforce persistence in a given order as follows. The sfence instruction generates the message sf in the pending buffer. The sf message *disappears* when it reaches the end of the pending buffer. However, it acts as a barrier *inside* the pending buffer. Since fo generates a barrier per in the persistency buffer, the two messages can enforce that persistency between write message occurs in a given order. This is illustrated in the program in Fig. 8, which we get from the program of Fig. 7 by inserting a store fence before the flush$_{\text{opt}}$ instruction. In γ_1, the thread has executed its instructions and output the corresponding messages in its pending buffer. In the persistency semantics, the message fo^x cannot overtake a write message on the same variable x. Given this, and the fact that the message sf cannot be re-ordered with any other messages in the pending buffer means that the messages move form the pending to the persistency buffer in the order shown in γ_2. Hence the addition of the sfence after the flush$_{\text{opt}}x$ has enforced that writes persist in the same order as they are issued by the thread.

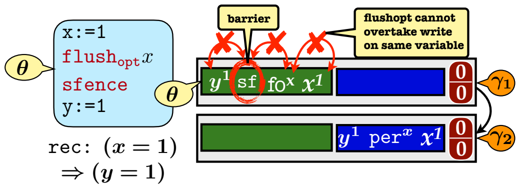


Fig. 8. Strengthening the flush$_{\text{opt}}$ instruction.

Another way to tackle the weakness of the flush$_{\text{opt}}$ instruction is to replace with the stronger flush instruction flush$_{\text{opt}}x$, which generates a message fl^x (Fig. 9). The latter cannot be overtaken by write instructions and generates a barrier message per^x in the persistency buffer, thus enforcing the correct order of persistency.

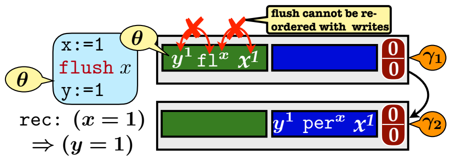


Fig. 9. Strengthening the flush$_{\text{opt}}$ instruction.

2.1.7 Overtaking Other Instructions. In the program of Fig. 10, the write instruction z:=1 can persist before y := 1 (both issued by the thread ϕ), only if the instruction flush$_{\text{opt}}y$ *overtakes* the write instruction `x:=1` (both issued by the thread θ). The explanation is as follows. In order for the instruction `z:=1` to execute (and persist), we need the write messages generated by the write instructions to cross from the pending stage to the persistency stage in the order y^1, x^1, x^2, and finally z^1. To see this, we make the following observations:

- x^1 and x^2 are in the same pending buffer and hence they cannot be reordered.
- The condition `(a==0)` and `(b==2)` implies that the messages x^1 and x^2 must reach the persistency buffer between the time points where ϕ executes the read instructions `a:=x` and `b:=x`.
- The memory fence instruction `mfence` can be executed only when the pending buffer of ϕ is empty, and hence the message y^1 reaches the persistency stage before the instruction `a:=x` is executed.
- The thread write instruction `z:=1` is executed after the read instruction `b:=x`, and hence also the message z^1 is put in the pending buffer after the execution of `b:=x`.

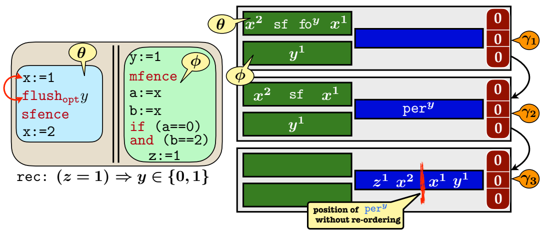


Fig. 10. `flush`$_{\texttt{opt}}$ overtaking other instructions.

Let us now consider two cases, namely (i) $\texttt{fo}^y$ overtakes x^1. (ii) $\texttt{fo}^y$ does not overtake x^1. Case (i) is depicted in Fig. 10, where $\texttt{fo}^y$ crosses first to the persistency buffer and transforms to the barrier $\texttt{per}^y$ (the transition from γ_1 to γ_2). Afterwards the barrier can be removed from the buffer and the rest of the messages can cross to the persistency buffer in the above mentioned order. Since write messages on different variables can be re-ordered in the persistency buffer, the message z^1 can persist before y^1. In case (ii), since the message `sf` acts as a barrier (as explained in Sec. 2.1.6), it prevents the message x^2 from being re-ordered with $\texttt{fo}^y$, so $\texttt{fo}^y$ will enter the persistency buffer between x^1 and x^2 . Consequently, as indicated in Fig. 10, the barrier $\texttt{per}^y$, created by $\texttt{fo}^y$, will separate z^1 from y^1, which means that z^1 cannot persist before y^1. Notice that, in this example, the order in which two writes in the thread ϕ persist depends on re-orderings of instructions in θ.

2.1.8 *Unbounded Re-orderings.*

In general, the $\texttt{fo}^x$ instruction can be involved in several types of re-orderings inside the pending buffer. First, it can *overtake* a write or a `fo` message on any variable different from x, and also overtake any other `fo` message. Second, it may be *overtaken* by any write or `fo` message, regardless of the variable, and by a `fl` message on any variable different from x. For example, assume that a program has a run in which messages are transferred from the program to the pending buffer in the sequence shown[2] in B_1 of Fig. 11. Using the re-orderings shown in the figure, the persistency semantics allows the messages to cross to the persistency stage in different orders, e.g., in the order B_2', or B_2. Since the pending buffers are unbounded, there is *a priori* no bound on the number of orderings that may occur before messages are transferred to the persistency stage, along a given run of the program.

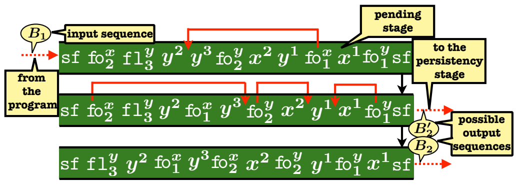

B1
input sequence
pending stage
sf fo₂^x fl₃^y y² y³ fo₂^y x² y¹ fo₁^x x¹ fo₁^y sf
from the program
to the persistency stage
sf fo₂^x fl₃^y y² fo₁^x y³ fo₂^y x² y¹ x¹ fo₁^y sf
B2'
B2
possible output sequences
sf fl₃^y y² fo₁^x y³ fo₂^x x² fo₂^y y¹ fo₁^y x¹ sf

Fig. 11. Re-ordering wrt. `flush`$_{\texttt{opt}}$.

2.1.9 *The Reachability Problem.*

We are interested in the *reachability problem*. Given a program, together with a recovery procedure, we would like to check whether the program violates a given assertion, characterized as a set of *bad thread (process) states*. Our goal is to prove the decidability of the reachability problem by applying the framework of Well-Structured Transition Systems

[2]To distinguish the different occurrences of `fo` and `fl` messages, we mark them by subscripts in the figure.

(WSTS), instantiated to the case of (finite-state) threads operating on (unbounded) FIFO buffers. The application of the framework to the persistency semantics is not easy. Classically, when applying the framework of WSTS to systems with FIFO-like data structures, we expect the system to satisfy four *"criteria of well-behavior"*, that are not satisfied by the persistency semantics, namely:

(1) *Monotonicity.* The behavior of the system should be monotone wrt. the standard sub-word ordering on FIFO buffers. We give the formal definition of monotonicity in Sec. 7. Roughly speaking, it means that if we insert an additional message in a given buffer then, from the new (larger) configuration, the system is able to perform at least the same sequence of transitions as from the old (smaller) configuration. The behavior of the persistency semantics is not monotone. For instance, inserting flush and fence messages in the buffers will restrict the re-ordering of messages, and hence may prevent the the system from performing the same sequence of transitions. For instance, inserting an `sf` message in the pending buffer of configuration γ_1 in Fig. 7, between the messages y^1 and fo^x will prevent the re-ordering indicated in the figure, and thus restrict the behavior of the program.
(2) *Locality.* The system should perform its operations *locally* on each buffer, without interaction with the rest of the buffers. As explained above, read operations are not local in the persistency semantics due to the memory hierarchy.
(3) *FIFO.* The buffers should not re-order messages. As we saw, both write messages (in the persistency buffer), and flush messages (in the pending buffers) can be re-ordered.
(4) *Normality.* The buffers should only allow normal (standard) operations, which can be:
   - Update operations, i.e., enqueueing and dequeueing messages.
   - Other operations, provided that they can be implemented using a finite data structure. Typical examples of such operations are reading the latest write operation by a thread on a variable, reading the value of a variable in the shared memory, or removing a distinguished message from inside (not necessarily at the head of) a buffer.

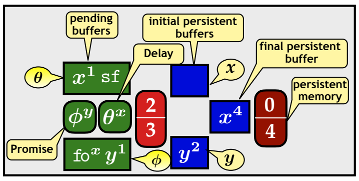
   In the persistency semantics, there are several examples where dequeueing is not made from the head of the buffer. This is a side-effect of the fact that messages overtake each other. For instance, in Fig. 5, the write message y^1 can be dequeued from the persistency buffer in γ_2.

Necessity vs Sufficiency of the Criteria of Well-behavior. We emphasize these criteria are sufficient rather than necessary conditions for obtaining decidability. In particular, they help bring us close to "FIFO-like" queues with monotonic behaviors which can be handled in the WSTS framework.

2.2 The Basic Scheduling Semantics

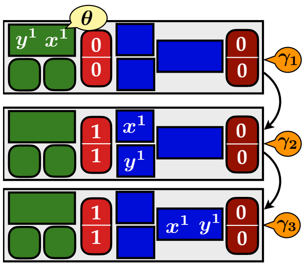
The basic scheduling semantics satisfies the criteria of locality and FIFO, but not monotonicity. The operational model, whose architecture, illustrated in Fig. 2, is equivalent to the persistency semantics wrt. the reachability problem. As in the persistency semantics, it consists of a *pending* and a *persistency* stage. These stages implement features to satisfy the criteria of well-behavior (except monotonicity). To ensure locality of read operations, we add an *intermediate* memory to the pending stage (Sec. 2.2.2). To simulate the re-ordering of write instructions in the persistency stage, we replace the (only) persistency buffer, by a two-stage FIFO-buffer system (Sec. 2.2.3). First, we have a set of FIFO buffers, one for each variable x, called the *x-initial persistency buffer*; followed by a single FIFO buffer, called the *final persistency buffer*. We get rid of re-orderings caused by `per` messages (recall that a `per` message can overtake write messages on different variables), by eliminating the `per` message altogether from the semantics. Instead, we put an extra condition that constrains how we remove `fo` messages from the pending buffers (Sec. 2.2.4). Getting rid of `per` messages has the additional advantage that it eliminates one source of non-monotonicity in the semantics (recall that `per` messages block certain re-orderings). To simulate the re-ordering of flush messages in the pending buffers, we use a *scheduling protocol* that regulates the flow of messages to the persistency stage. The protocol uses two *finite* sets of *Boolean* flags, called the *delay* and *promise* flags, that fully capture the effect of (unbounded numbers of) flush message re-orderings, without losing precision (Secs. 2.2.6, and 2.2.7).

2.2.1 Configurations. The figure (on the right) illustrates a configuration in the scheduling semantics with two threads θ and ϕ, and two shared variables x and y. The pending buffers contain the same messages as in Sec. 2.1.1. The values of x and y in the intermediate memory are 2 and 3; and in the persistent memory 0 and 4, respectively. The initial persistency buffer of x is empty, while that of y contains a single write message assigning 2 to y. The final persistency buffer contains a single write message, assigning 4 to x. The modules `Promise` and `Delay` are sets of Boolean flags, one for each variable and thread. In the figures, we show the sets of flags that are true. In the figure, the promise flag of ϕ and y is true, while all the other promise flags are false. Similarly, the delay flag of the thread θ and x is true, while all the other delay flags are false.



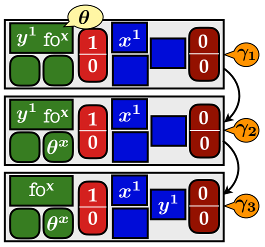
2.2.2 Intermediate Memory. We add an explicit *intermediate memory* after the pending buffers. When a write message reaches the end of the pending buffer of a thread θ, it can non-deterministically be deleted and used to update the intermediate memory. When θ performs a read instruction on a variable x, and its pending buffer does not contain write messages on x, then θ fetches the value of x from the intermediate memory without consulting the persistency stage. This makes read instructions local to the pending stage, and eliminates the hierarchical structure of the memory. In the figure of Sec. 2.2.1 the thread θ sees the value 1 of x (from its pending buffer), resp. 3 of y (from the intermediate memory). The thread ϕ sees the values 2 (from the intermediate memory) resp. 1 (from its pending buffer).

2.2.3 Simulating Write Re-orderings. Re-ordering of write messages is simulated by letting these messages move non-deterministically from the initial persistency buffers to the final persistency buffer. The figure (on the right) describes how this is done in case of the program of Sec. 2.1.3. Write instructions are appended to the end of the pending buffer belonging to the thread (configuration γ_1). The pending stage can non-deterministically select a write message at the head of some pending buffer and, at the same time, remove it, use it to update the intermediate memory, and move it to the persistency stage. A write messages on a variable x is forwarded to the x-initial persistency buffer (configuration γ_2). Messages are selected *non-deterministically* from the heads of the different initial buffers, and transferred to the final persistency buffer. This allows to re-order the write messages x^1 and y^1 (configuration γ_3). This example might give the impression that the final persistency buffer is not needed, since we can achieve the same effect by simply letting the messages from the initial buffers to update the persistent memory directly, without making them pass through the final buffer. However, as described in Sec. 2.2.5, we need the final buffer to avoid using the `per` messages in the semantics.

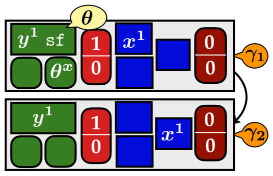


2.2.4 Delaying the `flush`$_{\texttt{opt}}$ *Instruction.* We implement a mechanism to allow other instructions to overtake the `flush`$_{\texttt{opt}}$ instruction without compromising the FIFO policy of the pending buffer. Our solution is to *delay* a message fo^x that is at the head of the buffer. To capture delays, we use a Boolean flag, called the `Delay`-flag, for each thread and variable. If the x-`Delay`-flag is true for a given a thread θ, then this tells us that we are currently delaying (at least) one fo^x in θ.

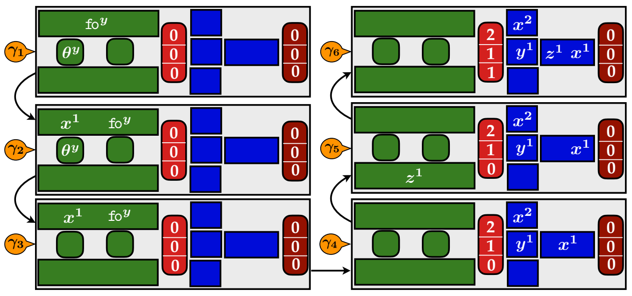
In the figure (on the right) we show how this allows us to simulate the scenario of Sec. 2.1.5. In γ_1, the thread θ has executed all its three instructions. The message x^1 has already crossed to the x-initial persistency buffer. When the fo^x message is fetched from the pending buffer, we will not let it cross immediately to the persistency stage. Instead, we update the corresponding `Delay`-flag to `true` (configuration γ_2). This allows the operation to be potentially overtaken by other operations. The message y^1 is still in the pending buffer. In γ_3, the message y^1 has crossed to the y-initial persistency buffer, thus overtaking the fo^x message, and also from there it has crossed to the final persistency buffer, thus persisting before the message x^1. Later in the run, the fo^x message can be *released*, i.e., be allowed to take effect. This is done by resetting the x-`Delay`-flag to `false` (under a certain condition described in Sec. 2.2.5).



2.2.5 Removing the `per` *Messages.* To simulate the effect of $\mathtt{per}^x$ messages, we require that a delayed message fo^x in thread θ can be released, *only if* the x-initial persistency buffer is empty. In the sequel, we call this operation *emptiness testing*. In this manner, we create an implicit barrier without the need to explicitly adding the message $\mathtt{per}^x$ (as we did in the persistency semantics). This is illustrated in the figure (on the right) which simulates the scenario of Sec. 2.1.6. The configuration γ_1 is similar to the configuration γ_2 in Sec. 2.2.4, except that the thread θ has now also executed the `sfence` instruction, and put the associated message `sf` in its pending buffer. The fact that store fences cannot overtake $\mathtt{flush}_{\mathtt{opt}}$ instructions, is modelled by forbidding the removal of `sf` from the buffer until the delay is released. Finally, as described above, the delay can only be released when the x-initial persistency buffer becomes empty, i.e., after the message x^1 has moved to the final persistency buffer (configuration γ_2). Since the latter is a FIFO buffer, the message y^1 cannot catch up with x^1, and hence the latter persists first.

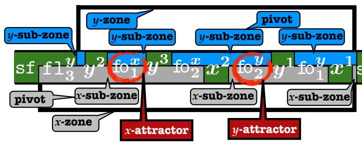


2.2.6 Promising the $\mathtt{flush}_{\mathtt{opt}}$ *Instruction.* Similarly to delays, we implement a mechanism to allow $\mathtt{flush}_{\mathtt{opt}}$ overtake other instructions while keeping the FIFO policy of the pending buffers. For that, we allow a thread θ to *promise* that it will perform a $\mathtt{flush}_{\mathtt{opt}} x$ instruction. Concretely, we insert the message fo^x at the end the pending buffer of θ, and set the x-`Promise`-flag of θ to `true`. In the figure (on the right), we simulate the scenario of Sec. 2.1.7 in the scheduling semantics. In the configuration γ_1, the thread has made a promise on y. As long as the flag is set, the thread is only allowed to perform instructions that can be overtaken by the instruction $\mathtt{flush}_{\mathtt{opt}} y$. For instance, in γ_1, the thread θ is allowed to perform a write instruction on x if such an instruction is available, but not a write instruction on y. In γ_2, the thread has performed its first instruction, i.e., `x:=1`. Notice that the promise has allowed the instruction $\mathtt{flush}_{\mathtt{opt}} y$ to overtake `x:=1`. Next, θ performs the $\mathtt{flush}_{\mathtt{opt}} y$ instruction which allows to discharge the promise. This is indicated by resetting the y-`Promise`-flag to `false` (configuration γ_3). Now, θ can perform the rest of its instructions, one after one, and store the corresponding messages in its pending buffer, while thread ϕ can execute its first instruction `y:=1`, and store the corresponding message in its pending buffer. In the next steps, the two threads propagate their buffer contents to initial



persistency buffers and, in the case of the message x^1 further to the final persistency buffer, while updating the intermediate memory accordingly (configuration γ_4). Now, the thread ϕ executes the instruction `z:=1` (configuration γ_5). The message z^1 moves (in two steps) to the final persistency buffer (configuration γ_6). The instruction `z:=1` will thus persist before `y:=1`.

2.2.7 Unbounded Re-orderings. We explain how the scheduling semantics[3] captures the scenario where multiple re-orderings involving fo^x messages in the persistency semantics (Sec. 2.1.8). In fact, the scheduling semantics forbids such re-orderings. Instead, it uses the `Promise`- and `Delay`-flags to implement a *scheduling protocol* that eliminates re-orderings but still maintains exactness wrt. the reachability problem. For any *input* sequence of messages provided by the program to the pending stage, the semantics guarantees that it generates at least one *output* sequence of write messages that follows the protocol and crosses from the pending stage to the persistency stage. We will define the protocol in several steps: (i) We divide the input sequence, wrt. each variable x, into contiguous segments which we call *x-zones*. (ii) We show that the persistency semantics satisfies an *attraction* property: for any input sequence of messages to the pending buffer, it is possible to generate an output sequence such that, inside each x-zone, all occurrences of the fo^x message are consecutive (they are "attracted" to each other inside the zone.) (iii) The scheduling semantics produces the effect of at most one copy of the per^x message per x-zone, exploiting the fact that consecutive occurrences of the per^x can be merged to a single copy without affecting the set of reachable configurations. (iv) We show that the scheduling protocol needs to be applied only to one *active* zone at a time, and hence one `Promise`-flag and one `Delay`-flag per variable and thread suffices to ensure exactness. To illustrate these ideas, we consider Fig. 12. The sequences B_1 and B_2 are identical to the input and output sequences in the example of Sec. 2.1.8. Given the output sequence B_2, the sequence of messages that will cross to the persistency buffer is given by D_2.



Zones. For a sequence of messages B and a variable x, we say that B is an *x-zone* if it contains only messages that can overtake fo^x. Each x-zone is divided into a sequence of *x-sub-zones*. An x-sub-zone contains only messages that can be overtaken by fo^x. This means that each pair of x-sub-zones is separated by a write message on x. The pivot x-sub-zone is the left-most x-sub-zone containing an occurrence of the fo^x message. The figure (on the right) shows the sequence B_2 from Sec. 2.1.8. The Figure shows an x-zone, with three sub-zones, and a y-zone with four sub-zones. The pivot x-sub-zone is the first one from the left, and pivot y-sub-zone is the third one from the left.

Attraction. The operation exploits the fact a message fo^x can be re-ordered with any other message inside an x-sub-zone, and be moved to the left any number of steps inside the x-zone. It selects an fo^x message in the pivot x-zone, called the

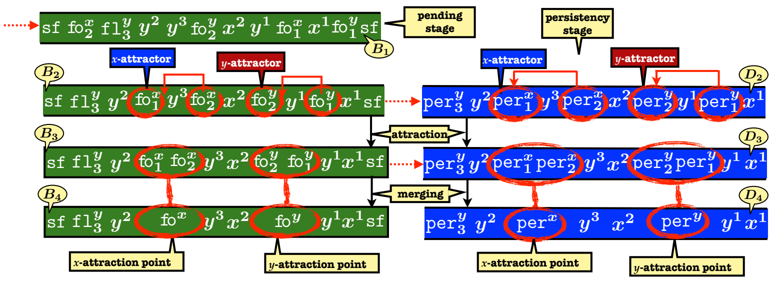

sf fo$_2^x$ fl$_3^y$ y^2 y^3 fo$_2^y$ x^2 y^1 fo$_1^x$ x^1 fo$_1^y$ sf
B_1
pending stage
persistency stage
x-attractor
y-attractor
x-attractor
y-attractor
B_2
sf fl$_3^y$ y^2 fo$_1^x$ y^3 fo$_2^x$ x^2 fo$_2^y$ y^1 fo$_1^y$ x^1 sf
D_2
per$_3^y$ y^2 per$_1^x$ y^3 per$_2^x$ x^2 per$_2^y$ y^1 per$_1^y$ x^1
attraction
B_3
sf fl$_3^y$ y^2 fo$_1^x$ fo$_2^x$ y^3 x^2 fo$_2^y$ fo$_1^y$ y^1 x^1 sf
D_3
per$_3^y$ y^2 per$_1^x$ per$_2^x$ y^3 x^2 per$_2^y$ per$_1^y$ y^1 x^1
merging
B_4
sf fl$_3^y$ y^2 fo^x y^3 x^2 fo^y y^1 x^1 sf
D_4
per$_3^y$ y^2 per^x y^3 x^2 per^y y^1 x^1
x-attraction point
y-attraction point
x-attraction point
y-attraction point

Fig. 12. Zones and uniform runs.

[3] This sub-section explains applications of the model, using the features already introduced in the previous sub-sections, rather than adding new features.

attractor, and moves the rest of the fo^x messages in the whole zone, called the x-*movers*, to a position, called the x-*attraction point*, beside the attractor. In Fig. 12, we apply attraction to B_2 twice to obtain B_3. First, we apply attraction wrt. x where fo_1^x is the x-attractor, and fo_2^x is the (only) x-mover; and then we apply attraction wrt. y where fo_2^y is the y-attractor, and fo_1^y is the (only) y-mover. This means that the sequence of messages that will cross to the persistency buffer is of the form D_3. Notice that applying attraction in the pending stage will change the sequence of messages passing through the persistency buffer in a similar fashion: the corresponding $\mathtt{per}$ messages are attracted to each other, resulting in corresponding attraction points. (cf. D_2 and D_3 in Fig. 12). Attraction does not reduce the set of allowed re-orderings of write messages in the persistency buffer. The reason is that the set of constraints implied by the movers in their new positions, is identical to the set of constraints implied by the attractor. However, the set of allowed re-orderings may increase, as the constraints by the movers in their original positions will now disappear. In D_2, the only allowed re-ordering is between x^1 and y^1 while D_3 allows, in addition, the re-ordering of x^2 and y^3. This means that applying attraction to an x-zone allows to reach the same set of persistent memory states as before (but possibly more). Since attraction corresponds to re-orderings that are always allowed by the persistency semantics, its application does not comprise precision, as it does not change the set of reachable persistent memory states (and consequently it does not change the set of reachable configurations).

Merging. Given an input sequence of messages to the persistency stage, the scheduling semantics produces an output sequence on which it has applied attraction. Notice that the number of fo^x messages in an x-zone can still be unbounded. Therefore, the scheduling semantics applies an additional operation, called *merging*, which collapses the consecutive copies of the fo^x resp. $\mathtt{per}^x$ message into one copy. Applying compression to B_3 and D_3 leads to B_4 resp. D_4. Merging preserves *exactly* the set of allowed re-orderings of write messages in the persistency buffer: multiple copies of $\mathtt{per}^x$ create an *identical* barrier as a single copy of $\mathtt{per}^x$. In both cases, they prevent later write messages from overtaking earlier write messages on x. In Fig. 12, the set of re-orderings of write messages allowed in D_3 and D_4 are identical.

Compression. The operation consists in applying attraction followed by merging. The net effect of compression is to collect all the fo^x messages (and hence also all the $\mathtt{per}^x$ messages) in the x-attraction point, and merge them into a single message. For instance, B_4 and D_4 are compressions of B_2 resp. D_2. Compression *preserves* the set of reachable configurations since both attraction and merging do.

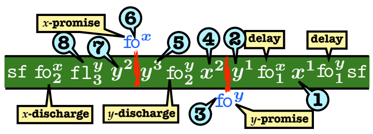

x-promise
6
fo^x
8
7
5
4
2
delay
delay
$\mathtt{sf}\ \mathsf{fo}_2^x\ \mathtt{fl}_3^y\ y^2\ y^3\ \mathsf{fo}_2^y\ x^2\ y^1\ \mathsf{fo}_1^x\ x^1\ \mathsf{fo}_1^y\ \mathtt{sf}$
x-discharge
y-discharge
3
fo^y
1
y-promise

Fig. 13. The scheduling protocol.

Implementing Compression in the Scheduling Semantics. The scheduling protocol uses $\mathtt{Delay}$- and $\mathtt{Promise}$-flags to guarantee that at least one compression is generated for each x-zone. We give a high-level description of the scheduling protocol using Fig. 13. Suppose that we are given the input sequence B_1 from Fig. 12. We show how the scheduling semantics allows simulate the output sequence D_4. The indices ①, . . . , ⑧ give the order in which the messages leave the buffer. Recall that, in the scheduling semantics, a write message that reaches the head of a pending buffer will eventually be forwarded to the persistency stage, while a fo message is used to perform the emptiness checking operation on the relevant initial persistency buffer (rather than producing a $\mathtt{per}$ message). The main ingredients of the protocol are the following.

• If the next input message cannot overtake the fo^x message then this signals the start of a new *active* x-zone. The previous x-zone will become passive. In Fig.13, the reception of the first message sf creates both a new active x-zone and a new active y-zone. In particular, several zones may be active at the same time but at most one per variable.

• At some point inside the x-zone, the protocol guesses the position of the x-attractor. This is done by (i) setting the x-Promise-flag to true, and (ii) putting a message fo^x in the buffer. In Fig. 13, this is done at position 6 for x and at position 3 for y. A message fo^x (the x-promise) is inserted after receiving y^3 and before receiving y^2, and analogously, a message fo^y (the y-promise) is inserted after receiving y^1 and before receiving x^2. The promises are artificial messages that are inserted by the semantics, and do not correspond to actual messages received from the thread. In fact, the promised messages are the only fo messages from the current zones that will be used to perform the emptiness testing operation. The "actual" fo messages are all deleted either when they generated or when they reach the head of the pending buffer. Thus, the protocol will test the emptiness of the initial x-initial persistency buffer at most once for any x-zone. After the promise is made, we are inside the pivot x-sub-zone, and hence, the thread is not allowed to provide write messages on x, until the promise is discharged.

• The fo messages that reach the head of the buffer before the promise is made (fo_1^x and fo_1^y) are *delayed*, i.e., they are simply fetched and removed when reaching the head of the buffer, while setting the corresponding Delay-flag to true (or keeping the value of the flag if already true).

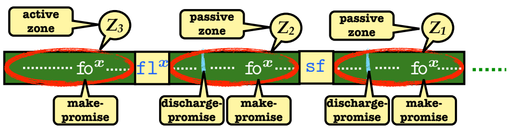
• Write messages are appended and received from the buffer as usual. This means x^1 will leave the buffer first followed by x^1.

• The fo's that are received from the thread after the promise has been made, are all discarded. However, the very last one is used to discharge the promise, i.e., resetting the corresponding Promise-flag to false. In our example, fo_2^x and fo_2^y are used to discharge the x- and the y-promise respectively.

• When the promised message for x reaches the head of the buffer, it is eventually used to check the emptiness of the x-initial persistency buffer and simultaneously reset the x-Delay-flag to false. At position ③, the promised y-message reaches the head of the buffer, can will be used to preform the emptiness check.

Overall, in our example, has produced the compression B_4 (of Fig. 12).

Handling Multiple Zones. In general, a pending buffer may contain several x-zones, separated by messages that cannot overtake fo^x (see the figure on the right). However, at any point of time, all these zones will be passive except, possibly, the left-most one. The reason is that, by construction, we activate a new zone only after we have made the previous one passive (discharged its promise). The value of the x-promise-flag is true exactly when the left-most x-zone is active. In such a manner, we can keep track of an unbounded number of over-takings using two Boolean flags for any thread and variable.

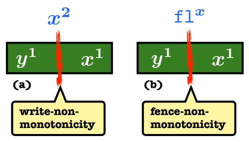


2.3 The Refined Scheduling Semantics

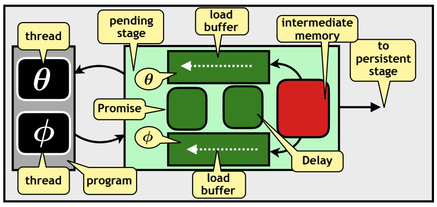
The basic scheduling semantics satisfies the locality and FIFO criteria, but fails to meet the monotonicity criterion. Monotonicity fails exactly in two places, namely the pending buffers, and in the final persistency buffer. Therefore, we replace these buffers by yet new types of buffers, resulting in the *refined scheduling semantics*. The refined scheduling semantics is equivalent to the basic scheduling semantics up to the reachability problem, and it satisfies the criteria of well-behavior including monotonicity. This allows us to show the well-structuredness of the entire system.

2.3.1 Load Pending Buffers. There are two sources of non-monotonicity in the pending buffers of basic scheduling semantics, namely *write-non-monotonicity* and *flush/fence-non-monotonicity*.

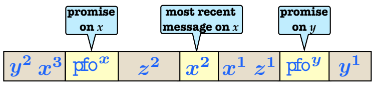
Write-non-monotonicity is illustrated in the figure (on the right), where the pending buffer contains a write message $\langle \mathtt{wr}, x, 1\rangle$ followed by a write message $\langle \mathtt{wr}, y, 1\rangle$. Inserting an extra write message $\langle \mathtt{wr}, x, 2\rangle$ in between these two messages in the buffer, as depicted in the figure, would prevent other threads from observing the intermediate memory state in which $x = 1$ and $y = 1$. An example of flush/fence-non-monotonicity is depicted in the figure, where inserting an $\mathtt{fl}^x$ message would prevent the message y^1 from persisting before x^1.



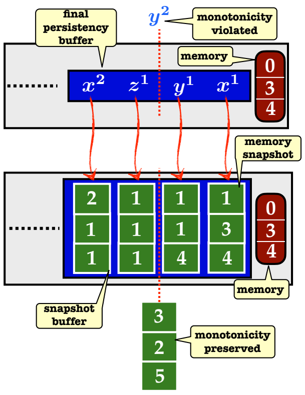
The purpose of the load-buffer semantics, depicted in the figure (on the right), is to eliminate the two sources of non-monotonicity. It is inspired from (but is more involved than) a similar semantics designed to eliminate write-non-monotonicity in the classical TSO semantics [Abdulla et al. 2018a]. We replace the pending store buffers of the threads by *pending load buffers*. The load buffers contain pending read messages, i.e., values that will potentially be used by forthcoming read instructions. In addition, as we describe below, a buffer may contain a bounded number of *distinguished messages*. The flow of information will now be in the reverse direction. Write instructions are performed atomically on the intermediate memory, while simultaneously propagating the corresponding messages to the persistency stage. The values of the variables are propagated non-deterministically from the memory to the load buffers of the threads. When a thread θ performs a read instruction on a variable $x \in \mathbb{X}$, it will first try to fetch the value of x from a distinguished message inside its buffer, called the *most-recent* message on x. The message represents the latest write instruction of θ on x. If no such message exists in the buffer, θ can read from the first message in its buffer, but only if that message is on x.



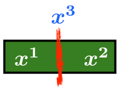
We get rid of flush/fence-non-monotonicity through two arrangements. First, we propagate flush and sfence instructions to the persistency stage immediately after they are issued, without ever putting them in the pending buffer. The second measure concerns the fo^x messages. Although compression leaves each x-zone with at most one fo^x message, there is in general no bound on the number of x-zones in a given run of the system. The refined semantics solves this problem by keeping a *single* distinguished *promise message* pfo^x per variable x in the load buffer, corresponding to the active x-zone. Since the flow is backward, a promise sends a $\mathtt{per}^x$ to the persistency buffer. Therefore, it is sufficient to keep a single copy of pfo^x that indicates whether there is currently an un-discharged promise on x, as well as the position, relative to the other messages, where the promise was made. The figure (on the right) depicts a configuration of a load buffer with three distinguished messages. Like store buffers, the number of normal (non-distinguished) read messages is potentially unbounded. However, the buffer is now monotone wrt. the normal read messages. More precisely, any additional read messages that are added to the buffer, may simply be neglected by the thread and therefore such messages will not constrain the behavior of the thread. The transition relation is not monotone wrt. to the distinguished messages. However, the number of these messages is finite (bounded by $2 \cdot |\mathbb{X}|$). This means that operations on the distinguished message do not violate the locality criterion. Furthermore, the existence of a bounded number of distinguished messages that violate monotonicity will not affect well-structuredness of the system. Technically, their existence can be encoded by factorizing the buffer into a finite sequence of sub-buffers that are separated by the distinguished messgaes. The distinguished messages themselves are manipulated using a finite-state component that is separated from the buffer.



2.3.2 The Snapshot Buffer. The final persistency buffer suffers from write-non-monotonicity. To see that consider the example shown in the figure (on the right), where the final persistency buffer contains a sequence of four write messages. By propagating the first three messages in the sequence, we can obtain a memory state where $x = 1$, $y = 1$, and $z = 1$. However, inserting an additional write message y^2 in the position indicated in the figure will make such a state unreachable. Therefore, we replace the final persistency buffer with a new equivalent buffer, called the *snapshot buffer*. The buffer contains a sequence of *memory snapshots*, i.e., each message is a function that assigns a value to each variable. Intuitively, the snapshot buffer simulates the final persistency buffer by calculating, for each message, the state of the persistent memory at the moment the current message hits the memory (see the figure). The snapshot buffer is monotone since inserting new snapshot messages does not prevent the same sequence of memory states to arise, even though these states may not occur contiguously: they may be separated by other states.



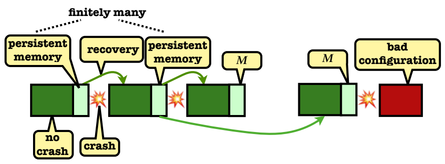
On the other hand, the initial persistency buffers are all monotone. The reason is that the different variables are separated, and therefore inserting an additional write message does not violate monotonicity as follows. In the figure (on the right), as soon as the message x^3 crosses to the snapshot buffer, we can let x^1 cross in the next step and hence the system will be able to perform the same transitions as before.



2.4 Semantical Equivalence, Decidability, and Complexity

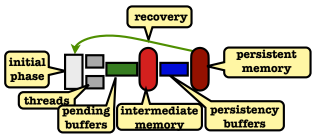
Semantical Equivalence. The scheduling semantics is equivalent to the persistency semantics, up to the reachability of process states. Formally, Lemma 5.1 shows that the basic semantics is equivalent to the scheduling semantics. Lemma 6.1 shows that refined semantics is equivalent to the basic semantics. Therefore, Lemma 5.1 and Lemma 6.1 together imply that the basic persistency and scheduling semantics are equivalent. As we saw above, the refined scheduling semantics fulfills all the criteria of well-behavior. This allows to solve the reachability problem, in which we ask whether we can reach a given set of *bad* thread states. Due the equivalence of the persistency and scheduling semantics (up to state reachability), the decidability of the reachability problem for the latter implies the problem for the former.

Crashes and Recovery. Observe that the scheduling semantics only considers program runs that are crash-free. However, solving the crash-free reachability problem allows to solve the full reachability problem in the presence of crashes. As usual with persistent memories (e.g., [Raad et al. 2020]), we assume that we are given a *recovery procedure*: a function takes the content of the persistent memory as input, and returns the re-starting states of the threads after a crash. A run of the system can be viewed as a sequence of crash-free segments, each of which ends with a crash. After a crash, the recovery procedure computes the re-starting states, and the execution of the system is resumed. A key observation here is that for each run leading to bad configuration, there is another run where the persistent memory states just prior to the crashes are all different. The reason is that if two such states are identical, then we can short-cut the run, as shown in the figure (on the right), and obtain a shorter run. This means that, we need only to consider runs where we bound the number of crashes that occur along



the run by the number of different memory states which is $|\mathbb{X}|^{|\mathbb{D}|}$ where $\mathbb{X}$ is the set of variables and $\mathbb{D}$ is the data domain (both of which are finite). Notice that there is still no bound on the length of each crash-free segment. In this manner, we have reduced the full reachability problem to a finite set of instances of crash-free reachability problem. Therefore, the decidability of the latter implies the decidability of the former.

Complexity. The reachability problem for the classical TSO model can be reduced to the crash-free reachability problem. This can be done by adding an initialization phase, performed by an extra thread, that re-initializes the threads, the intermediate memory (using atomic read-write operations). The recovery procedure can then be defined to always re-start the system from the initialization phase. Thus, any run of the system will consist of a set of runs under the classical model, each of which starts from the initial configuration. Since the reachability problem for the classical TSO model is non-primitive-recursive [Atig et al. 2010], the same applies to the crash-free reachability problem.



Cache Lines. The model of [Raad et al. 2020] uses the concept of cache lines that are shared by groups of variables Variables from the same cache line are indistinguishable wrt. reordering in the pending stage. Furthermore, the persistency barriers of variables on the same cache line are indistinguishable in the persistency stage. However, writes on different variables may overtake each other in the persistency stage even if they are in the same cache line. We can address cache lines via two modifications to our constructions. To adapt our pending stage to incorporate the cache lines, we simply treat fo^x and $\mathsf{fo}^{x'}$ as identical for x and x' in the same cache line. In the persistency stage we continue to have one initial persistency buffer per variable, but to receive a per^x, we require that the initial persistency buffers of all variables in x's cache line be empty.

3 EVENT AUTOMATA

We will give the formal definitions of the basic and refined scheduling semantics using *event automata* (or simply *automata*). Such an automaton can in general be infinite-state, and each transition is labeled by a pair of events, called the *input* resp. *output* events. The events represent the interaction of the automaton with the environment. We will consider automata from the *external* and *internal* point of view. For the external view, we study the *histories*, i.e., the sequences of input and output events, that are generated by the automaton. For the internal view, we consider the set of configurations that are reachable from the initial configuration. In the later section, we will define automata for describing the program, the pending stage, and the persistency stage. The last two automata are fixed for given sets of variables and threads, and do not depend on the actual definition of the program.

3.1 Notation

For a set A, we use A^* (A^+) to denote the set of (non-empty) finite words over A. For a word $w \in A^*$, we use $|w|$ to denote the length of w, and for $i : 1 \leq i \leq |w|$, we use $w[i]$ to denote the i^{th} element of w. For $a \in A$, we write $a \in w$ to denote that $w[i] = a$ for some $i : 1 \leq i \leq |w|$. For words $w_1, w_2 \in A^*$, we use $w_1 \cdot w_2$ to denote the concatenation of w_1 and w_2. We write $w_1 \preceq w_2$ if w_1 is a (not nececessarily contiguous) sub-word of w_2. For sets A and B, we use $[A \longrightarrow B]$ to denote the set of functions from A to B and write $f : A \longrightarrow B$ to denote that $f \in [A \longrightarrow B]$. We use $f[a \leftarrow b]$ to denote the function f' where $f'(a) = b$, and $f'(a') = f(a')$ if $a' \neq a$. For a function $f : A_1 \longrightarrow [A_2 \longrightarrow A_3]$ we define $f[a_1 \leftarrow a_2 \leftarrow a_3]$ to be the function f' such $f'(a_1)(a_2) = a_3$, and $f'(a_1')(a_2') = f(a_1')(a_2')$ if $a_1' \neq a_1$ or $a_2' \neq a_2$.

3.2 Automata

We assume a finite alphabet Σ which we refer to as the set of *events*. We assume that Σ contains a special *empty event*, denoted $-$, that is not visible to the environment. For a word $h \in \Sigma^*$, we say that h is a *history* over Σ if $- \notin h$, i.e., it does not contain the empty event. For a word $w \in \Sigma^*$, we define w^- to be the maximal subword h of w that is a history, i.e., h is the word we get from w by removing all occurrences of $-$. A *behavior* b over Σ is pair $\langle h, h' \rangle$ of histories over Σ. For sets of behaviors B and B', we define their *composition* $B \otimes B' := \{\langle h, h' \rangle \mid \exists h''.\ (\langle h, h'' \rangle \in B) \wedge (\langle h'', h' \rangle \in B')\}$.

An *(event) automaton* $\mathcal{A}$ is a tuple $\langle \Gamma, \Gamma_{init}, \Sigma_{in}, \Sigma_{out}, \rightarrow \rangle$ where Γ is a set of *configurations*, $\Gamma_{init} \subseteq \Gamma$ is the set of *initial* configurations, Σ_{in} is the set of *input events*, Σ_{out} is the set of *output events*, and $\rightarrow \subseteq \Gamma \times \Sigma_{in} \times \Sigma_{out} \times \Gamma$ is the *transition relation*. We write $\gamma \xrightarrow{e/e'} \gamma'$ to denote that $\langle \gamma, e, e', \gamma' \rangle \in \rightarrow$. We write $\gamma_1 \rightarrow \gamma_2$ to denote that $\gamma_1 \xrightarrow{e_1/e_2} \gamma_2$, for some $e_1 \in \Sigma_{in}$ and $e_2 \in \Sigma_{out}$; and use $\xrightarrow{*}$ to denote the reflexive transitive closure of $\rightarrow$. For a configuration $\gamma \in \Gamma$, we write $\langle \mathcal{A}, \gamma \rangle \models \gamma'$ to denote that $\gamma \xrightarrow{*} \gamma'$. For a set of configurations $G \subseteq \Gamma$, we write $\langle \mathcal{A}, G \rangle \models \gamma'$ to denote that $\langle \mathcal{A}, \gamma \rangle \models \gamma'$ for some $\gamma \in G$. Finally, we write $\mathcal{A} \models \gamma$ to denote that $\langle \mathcal{A}, \Gamma_{init} \rangle \models \gamma$. We extend the definition to $\mathcal{A} \models G$ where G is a set as expected. A *run* ρ of $\mathcal{A}$ is a sequence $\gamma_0 \xrightarrow{\langle e_1, e_1' \rangle} \gamma_1 \xrightarrow{\langle e_2, e_2' \rangle} \gamma_2 \xrightarrow{\langle e_3, e_3' \rangle} \cdots \xrightarrow{\langle e_n, e_n' \rangle} \gamma_n$, where $\gamma_0 \in \Gamma_{init}$. We use $\mathsf{Runs}(\mathcal{A})$ to denote the set of runs of $\mathcal{A}$. We define $[\![\rho]\!] := \left\langle (e_1, e_2, e_3 \cdots e_n)^-, \left(e_1', e_2', e_3' \cdots e_n'\right)^- \right\rangle$, and define $[\![\mathcal{A}]\!] := \{[\![\rho]\!] \mid \rho \in \mathsf{Runs}(\mathcal{A})\}$. In other words, $[\![\mathcal{A}]\!]$ is the set of behaviors generated by $\mathcal{A}$.

Given, two automata $\mathcal{A}_1 = \left\langle \Gamma^1, \Gamma^1_{init}, \Sigma^1_{in}, \Sigma^1_{out}, \rightarrow_1 \right\rangle$ and $\mathcal{A}_2 = \left\langle \Gamma^2, \Gamma^2_{init}, \Sigma^2_{in}, \Sigma^2_{out}, \rightarrow_2 \right\rangle$, we define their composition $\mathcal{A}_1 \otimes \mathcal{A}_2$ to be the automaton $\mathcal{A} = \langle \Gamma, \Gamma_{init}, \Sigma_{in}, \Sigma_{out}, \rightarrow \rangle$, defined as follows:

- $\Gamma = \Gamma_1 \times \Gamma_2$. • $\Gamma_{init} = \Gamma^1_{init} \times \Gamma^2_{init}$. • $\Sigma_{in} = \Sigma^1_{in}$. • $\Sigma_{out} = \Sigma^2_{out}$.
- $\langle \gamma_1, \gamma_2 \rangle \xrightarrow{e_1/e_2} \langle \gamma_3, \gamma_4 \rangle$ if one of the following conditions is satisfied:
  - $\gamma_1 \xrightarrow{e_1/-}_1 \gamma_3$, $e_2 = -$, and $\gamma_2 = \gamma_4$. – $\gamma_2 \xrightarrow{-/e_2}_2 \gamma_4$, $e_1 = -$, and $\gamma_1 = \gamma_3$.
  - $\gamma_1 \xrightarrow{e_1/e_3}_1 \gamma_3$ and $\gamma_2 \xrightarrow{e_3/e_2}_2 \gamma_4$ for some $e_3 \neq -$.

Notice that $[\![\mathcal{A}_1 \otimes \mathcal{A}_2]\!] = [\![\mathcal{A}_1]\!] \otimes [\![\mathcal{A}_2]\!]$. We extend the composition operator to $\mathcal{A}_1 \otimes \mathcal{A}_2 \otimes \cdots \otimes \mathcal{A}_n$ in the obvious manner. Let the resulting automaton be $\mathcal{A}$.

4 CONCURRENT PROGRAMS

We consider concurrent programs, consisting of a finite set Θ of threads that communicate through a finite set $\mathbb{X}$ of shared variable. The values of the variables are fetched from a finite domain $\mathbb{D}$, including the special value 0. A thread contains a finite set R of local variables, also ranging over $\mathbb{D}$. The behavior of a thread is defined by a finite set of instructions each labeled from a finite set L of labels. We use a simple but general programming language to describe the instructions. The syntax of the language as well as its semantics (as an event automaton) are standard, and are omitted here. The language allows standard instructions such as reading, writing, and compare-and-swap instructions on shared and local variables, as well as branching/jumping instructions, and finally flush and fence instructions that are specific to the architecture (as described in Sec. 2). Semantically, a program P induces an automaton, denoted $\mathcal{A}^{\textcircled{P}}$. A configuration of $\mathcal{A}^{\textcircled{P}}$ is a pair $\langle \lambda, R \rangle$ where $\lambda : \Theta \rightarrow L$ defines the label of the current instruction of each thread, and $\mathcal{R} : R \rightarrow \mathbb{D}$ defines the values of the local variables. The set of initial configurations contains a single configuration $\langle \lambda_{init}, \mathcal{R}_{init} \rangle$ where $\lambda_{init}(\theta)$ defines the label from which θ starts executing, and $\lambda_{init}(a) = 0$ for all local variables $a \in R$. The set of input events contains only the empty event, since we assume that the program does not take any input. An output event is either a *write* $\langle \theta, \mathtt{wr}, x, d \rangle$, a *read*

$\langle\theta, \mathtt{wr}, x, d\rangle$, an *rmw* $\langle\theta, \mathtt{rmw}, x, d_1, d_2\rangle$, an *optimized flush* $\langle\theta, \mathtt{fo}, x\rangle$, a *flush* $\langle\theta, \mathtt{fl}, x\rangle$, a *store fence* $\langle\theta, \mathtt{sf}\rangle$, a *memory fence* $\langle\theta, \mathtt{mf}\rangle$, or the empty event $-$, where $\theta \in \Theta$ is thread, $x \in \mathbb{X}$ is a variable, and $d, d_1, d_2 \in \mathbb{D}$ are values taken from the data domain. The transition relation is defined by a set of inference rules as usual.

For a labeling function $\lambda : \Theta \to L$, we define $\mathcal{A}^{\textcircled{P}} \Box \lambda$ to be the automaton we get from $\mathcal{A}^{\textcircled{P}}$ by changing the initial label of each thread θ in P to $\lambda(\theta)$.

5 THE BASIC SCHEDULING SEMANTICS

In this section, we give the formal definition of the basic scheduling semantics, as two automata that describe the pending and persistency stages respectively.

5.1 The Pending Stage

The pending stage automaton[4] (Fig. 14) is of the form $\mathcal{A}^{\textcircled{1}} = \left\langle \Gamma^{\textcircled{1}}, \Gamma^{\textcircled{1}}_{init}, \Sigma^{\textcircled{1}}_{in}, \Sigma^{\textcircled{1}}_{out}, \to_{\textcircled{1}} \right\rangle$, defined as follows. A configuration in $\Gamma^{\textcircled{1}}$ is a tuple $\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle$ of functions, defined as follows. The function $B : \Theta \to G^*$, where $G = (\{\mathtt{wr}\} \times \mathbb{X} \times \mathbb{D}) \cup (\{\mathtt{fo}, \mathtt{fl}\} \times \mathbb{X}) \cup \{\mathtt{sf}\}$ is the *buffer alphabet*, encodes the contents of the pending buffer of each thread. The function $\mathtt{Promise} : \Theta \to [\mathbb{X} \to \mathbb{B}]$ defines the promise flag of each variable in each thread. The function $\mathtt{Delay} : \Theta \to [\mathbb{X} \to \mathbb{B}]$ defines the delay flag of each variable in each thread. The function $M : \mathbb{X} \to \mathbb{D}$ defines the value of each variable in the intermediate memory. The set $\Gamma^{\textcircled{1}}_{init}$ of initial configurations is the singleton $\{\langle B_{init}, \mathtt{Promise}_{init}, \mathtt{Delay}_{init}, M_{init}\rangle\}$ where $B_{init} = \lambda\theta.\epsilon$, $\mathtt{Promise}_{init} = \lambda\theta.\lambda x.\mathtt{false}$, $\mathtt{Delay}_{init} = \lambda\theta.\lambda x.\mathtt{false}$, and $M_{init} = \lambda x.0$. In other words, we start from empty pending buffers, with no promises and delays, and with an intermediate state that assigns 0 to all the variables.

The set of input events is identical to the set output events from program automaton (described in Sec. 4). An output event is either a *write* $\langle \mathtt{wr}, x, d\rangle$, a *barrier* $\langle \mathtt{per}, x\rangle$, or the empty event $-$, where $x \in \mathbb{X}$ and $d \in \mathbb{D}$.

The transition relation $\to_{\textcircled{1}}$ is defined by the set of inference rules in Fig. 14, one set for each thread θ. Before we go through the rules, we give some remarks that make some of the rule premises easier to understand. First, we interpret the emptiness of the pending buffers of a thread θ as the conjunction of two conditions, an *explicit* condition, namely that $B(\theta) = \epsilon$, and an *implicit* condition, namely that the promise and delay flags should be false for all the variables. The implicit condition reflects the fact that a delay indicates that at least one occurrence of the message $\mathtt{fo}$ is (logically) inside the buffer. Second, all the inference rules respect the FIFO policy: messages are added to the end, and removed from the head of the buffers.

In the first group of rules, consisting of four rules, the automaton receives an event from the program that corresponds to read/write instructions. The first three transitions in the group do not generate any output events. In `Write`, the automaton receives an event $\langle\theta, \mathtt{wr}, x, d\rangle$ which means that θ performs a write instruction that assigns the value $d \in \mathbb{D}$ to the variable $x \in \mathbb{X}$. The corresponding write message $\langle \mathtt{wr}, x, d\rangle$ is appended to the end of the pending buffer of θ. A $\mathtt{flush}_{\mathtt{opt}}$ instruction on x cannot overtake a write instruction on x, and hence we require the the flag $\mathtt{Promise}(\theta)(x)$ is false. In `Read-Own-Write`, the automaton receives the event $\langle\theta, \mathtt{rd}, x, d\rangle$ which means that θ performs a read instruction on the variable $x \in \mathbb{X}$, in a configuration where there is at least one write message on x in the pending buffer of θ. In such a case, we take the value from the most recent write message on x in the buffer of θ. The rule `Read-from-Memory` is similar, but there is no write message on x in the pending buffer of θ. We fetch the value from the entry of x in the intermediate memory. Although a $\mathtt{flush}_{\mathtt{opt}}$ instruction cannot overtake a read instruction,

[4]To differentiate the automata we use for the different modules, we use superscripts like ①, ②, etc.

$$\frac{\mathtt{Promise}(\theta)(x) = \mathtt{false} \qquad B' = B[\theta \leftarrow \langle \mathtt{wr}, x, d\rangle \bullet B(\theta)]}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{\langle \theta, \mathtt{wr}, x, d\rangle / -}_{\textcircled{1}} \langle B', \mathtt{Promise}, \mathtt{Delay}, M\rangle}\ \text{Write}$$

$$\frac{B(\theta) = w_1 \bullet \langle \mathtt{wr}, x, d\rangle \bullet w_2 \qquad x \notin w_1}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{\langle \theta, \mathtt{rd}, x, d\rangle / -}_{\textcircled{1}} \langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle}\ \text{Read-Own-Write}$$

$$\frac{\nexists d.\ \langle \mathtt{wr}, x, d\rangle \in B(\theta) \qquad M(x) = d}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{\langle \theta, \mathtt{rd}, x, d\rangle / -}_{\textcircled{1}} \langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle}\ \text{Read-from-Memory}$$

$$\frac{\forall y \in \mathbb{X}.\ \mathtt{Promise}(\theta)(y) = \mathtt{false} \qquad \forall y \in \mathbb{X}.\ \mathtt{Delay}(\theta)(y) = \mathtt{false} \qquad B(\theta) = \epsilon \qquad M(x) = d_1 \qquad M' = M[x \leftarrow d_2]}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{\langle \theta, \mathtt{rmw}, x, d_1, d_2\rangle / \langle \mathtt{wr}, x, d_2\rangle}_{\textcircled{1}} \langle B, \mathtt{Promise}, \mathtt{Delay}, M'\rangle}\ \text{Read-Modify-Write}$$

$$\frac{\mathtt{Promise}(\theta)(x) = \mathtt{false} \qquad B' = B[\theta \leftarrow \langle \mathtt{fo}, x\rangle \bullet B(\theta)] \qquad \mathtt{Promise}' = \mathtt{Promise}[\theta \leftarrow x \leftarrow \mathtt{true}]}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{- / -}_{\textcircled{1}} \langle B', \mathtt{Promise}', \mathtt{Delay}, M\rangle}\ \text{Initiate-Promise}$$

$$\frac{(\langle \mathtt{fo}, x\rangle \in B(\theta)) \vee (\mathtt{Delay}(\theta)(x) = \mathtt{true}) \qquad \mathtt{Promise}(\theta)(x) = \mathtt{true} \qquad \mathtt{Promise}' = \mathtt{Promise}[\theta \leftarrow x \leftarrow \mathtt{false}]}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{\langle \theta, \mathtt{fo}, x\rangle / -}_{\textcircled{1}} \langle B, \mathtt{Promise}', \mathtt{Delay}, M\rangle}\ \text{Discharge-Promise}$$

$$\frac{\mathtt{Promise}(\theta)(x) = \mathtt{true}}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{\langle \theta, \mathtt{fo}, x\rangle / -}_{\textcircled{1}} \langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle}\ \text{Merge-FO}$$

$$\frac{B(\theta) = w \bullet \langle \mathtt{fo}, x\rangle \qquad B' = B[\theta \leftarrow w] \qquad \mathtt{Delay}' = \mathtt{Delay}[\theta \leftarrow x \leftarrow \mathtt{true}]}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{- / -}_{\textcircled{1}} \langle B', \mathtt{Promise}, \mathtt{Delay}', M\rangle}\ \text{Delay-FO}$$

$$\frac{\mathtt{Delay}(\theta)(x) = \mathtt{true} \qquad \mathtt{Delay}' = \mathtt{Delay}[\theta \leftarrow x \leftarrow \mathtt{false}]}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{- / \langle \mathtt{per}, x\rangle}_{\textcircled{1}} \langle B', \mathtt{Promise}, \mathtt{Delay}', M\rangle}\ \text{Release-FO}$$

$$\frac{\mathtt{Promise}(\theta)(x) = \mathtt{false} \qquad B' = B[\theta \leftarrow \langle \mathtt{fl}, x\rangle \bullet B(\theta)]}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{\langle \theta, \mathtt{fl}, x\rangle / -}_{\textcircled{1}} \langle B', \mathtt{Promise}, \mathtt{Delay}, M\rangle}\ \text{FL}$$

$$\frac{\forall y \in \mathbb{X}.\ \mathtt{Promise}(\theta)(y) = \mathtt{false} \qquad B' = B[\theta \leftarrow \mathtt{sf} \bullet B(\theta)]}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{\langle \theta, \mathtt{sf}\rangle / -}_{\textcircled{1}} \langle B', \mathtt{Promise}, \mathtt{Delay}, M\rangle}\ \text{SF}$$

$$\frac{\forall y \in \mathbb{X}.\ \mathtt{Promise}(\theta)(y) = \mathtt{false} \quad \forall y \in \mathbb{X}.\ \mathtt{Delay}(\theta)(y) = \mathtt{false} \quad B(\theta) = \epsilon}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{\langle \theta, \mathtt{mf}\rangle / -}_{\textcircled{1}} \langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle}\ \text{MF}$$

$$\frac{B(\theta) = w \bullet \langle \mathtt{wr}, x, d\rangle \qquad M' = M[x \leftarrow d] \qquad B' = B[\theta \leftarrow w]}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{- / \langle \mathtt{wr}, x, d\rangle}_{\textcircled{1}} \langle B', \mathtt{Promise}, \mathtt{Delay}, M'\rangle}\ \text{Propagate-Write}$$

$$\frac{\mathtt{Delay}(\theta)(x) = \mathtt{false} \qquad B(\theta) = w \bullet \langle \mathtt{fl}, x\rangle \qquad B' = B[\theta \leftarrow w]}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{\langle \theta, \mathtt{fl}, x\rangle / \langle \mathtt{per}, x\rangle}_{\textcircled{1}} \langle B', \mathtt{Promise}, \mathtt{Delay}, M\rangle}\ \text{Prop-agate-FL}$$

$$\frac{\forall y \in \mathbb{X}.\ \mathtt{Delay}(\theta)(y) = \mathtt{false} \qquad B(\theta) = w \bullet \mathtt{sf} \qquad B' = B[\theta \leftarrow w]}{\langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle \xrightarrow{- / -}_{\textcircled{1}} \langle B', \mathtt{Promise}, \mathtt{Delay}, M\rangle}\ \text{Prop-agate-SF}$$

Fig. 14. The pending stage automaton.

in contrast to the `Write` rule, we do not need to put any condition on the promise flags in the premise of the rule. The reason is the following properties. First, in contrast to write instructions, read instructions are instantaneous. More precisely, a write message needs to travel through the whole buffer before it "takes effect" and becomes visible to the other threads. A read instruction is performed atomically. Second, as we will see below (in the rule `Discharge-Promise`), we allow to discharge a promise only if the corresponding fo is still in the buffer or delayed. The net effect of these two properties is that a $\mathtt{flush}_{\mathtt{opt}}$, on a variable $y \in \mathbb{X}$, cannot overtake any read instruction on any variable. In `Read-Modify-Write`, the automaton receives an event $\langle \theta, \mathtt{rmw}, x, d_1, d_2\rangle$ which means that θ reads the value of the variable $x \in \mathbb{X}$. We require that the value of x in the intermediate memory is d_1, and that the pending buffer of θ is empty. As mentioned above, the emptiness of the buffer also implies that the promise and delay flags are all false. In such a case, we assign the value $d_2 \in \mathbb{D}$ to x in the intermediate memory. In contrast to the previous three rules, we generate a write event consisting of the variable x and the written value d_2. Furthermore, we require that the delay flags for all the variables are `false` in θ. This ensures that the `CAS` instruction that has induced the `rmw` does not overtake any $\mathtt{flush}_{\mathtt{opt}}$ instructions.

The next five rules concern the $\mathtt{flush}_{\mathtt{opt}}$ instruction. In `Initiate-Promise`, θ promises a $\mathtt{flush}_{\mathtt{opt}}$ instruction on a variable $x \in \mathbb{X}$. We append a message $\langle \mathsf{fo}, x\rangle$ to the end of the pending buffer of θ, and set the promise flag of x in θ to `true`. The flag remains true until the promise is discharged. This happens when an event $\langle \theta, \mathsf{fo}, x\rangle$ is performed by the program (the rule `Discharge-Promise`), at which point the flag may be reset to false. Relating to the scheduling protocol, described in Sec. 2.2.7, the two previous rules can also be used to add normal (non-artificial) fo messages, when an $\langle \theta, \mathsf{fo}, x\rangle$ is received from θ. This is done by first promising the message before the event, and then immediately discharging the promise when the event is received. The rule

`Merge-FO` allows additional occurrences of $\texttt{flush}_{\texttt{opt}}$ instructions without discharging the promise. As we saw in Sec. 2.2.7, this is necessary to simulation of attraction and merging operations in the scheduling protocol. In `Delay-FO`, a `fo` message has reached the head of the pending buffer. In such a case the automaton will delay the message, allowing other messages to overtake it. Although the message is always delayed, it may not necessarily be overtaken by other messages. The reason is that the `Release-FO` rule can be applied immediately afterwards to inform the persistency stage of the reception of the `fo` message. The latter also generates the corresponding event $\langle \texttt{per}, x\rangle$. As we will see in the next sub-section, when the event $\langle \texttt{per}, x\rangle$ is received by the persistency stage, it will not generate a $\langle \texttt{per}, x\rangle$ message in the initial buffer. The rule `Delay-FO` is enabled even if the delay flag for θ and x is already set. This is to take into account the absorption operation. In particular, we can interpret $\texttt{Delay}(\theta)(x)$ as θ having performed at least one $\texttt{flush}_{\texttt{opt}}$ on x rather than exactly one.

The next three rules deal with the thread issuing other fence instructions than $\texttt{flush}_{\texttt{opt}}$. In `FL`, the automaton receives an event $\langle \theta, \texttt{fl}, x\rangle$ from the program, indicating that the thread $\theta \in \Theta$ performs a `flush` instruction on the variable $x \in \mathbb{X}$, and the corresponding message $\langle \texttt{fl}, x\rangle$ is appended to the end of the pending buffer of the thread. The event is allowed to take place only if the x-`Promise`-flag `false`. The reason is that a $\texttt{flush}_{\texttt{opt}}$ instruction is not allowed to overtake a `flush` instruction on the same variable. The rule `SF` is similar and concerns the store fence instruction. We require that the y-`Promise`-flag is not set for any variable $y \in \mathbb{X}$, since an `sfence` instruction is not allowed to be overtaken a $\texttt{flush}_{\texttt{opt}}$ instruction on any variable. The rule `MF` concerns a memory fence, and the pending buffer of the thread should be empty when the instruction performed. In particular, the promise and delay flags should be `false` for all variables.

The last three rules concern output events that the automaton propagates to the persistency stage. The transitions can all be non-deterministically executed by the automaton whenever they are enabled. There are invisible to the program, and hence the rules do not involve any input events. *Propagate-Write* tells us that when a write message reaches the head of the pending buffer, then it may be fetched and used to update the intermediate memory. At the same time, the persistency stage is notified through the output event $\langle \texttt{wr}, x, d\rangle$. The rule `Propagate-FL` is similar in the sense that the persistency stage is notified through the event $\langle \texttt{per}, x\rangle$. In this case, the delay flag of x should be `false` since the `flush` instruction cannot overtake the $\texttt{flush}_{\texttt{opt}}$ instruction on the same variable. The persistency stage is again notified through the event $\langle \texttt{per}, x\rangle$. The last rule is similar and concerns the store fence instruction. The delay flag of all variables should be `false` since the `sfence` instruction cannot overtake the $\texttt{flush}_{\texttt{opt}}$ instruction on any variable.

For a function $M : \mathbb{X} \to \mathbb{D}$, giving the content of the intermediate memory, we define $\mathcal{A}^{①}[M]$ to be the automaton we get from $\mathcal{A}^{①}$ by changing the initial intermediate memory state to M.

5.2 The Persistency Stage

The automaton, depicted in the figure, receives events from the pending stage but does not output any events. Formally, $\mathcal{A}^{②} = \left\langle \Gamma^{②}, \Gamma^{②}_{init}, \Sigma^{②}_{in}, \Sigma^{②}_{out}, \to_{②} \right\rangle$, where the components defined as follows. A configuration in $\Gamma^{②}$ is a triple of the form $\langle D, E, N\rangle$. The function $D : \mathbb{X} \to (\mathbb{X} \times \mathbb{D})^*$ defines the contents of the initial persistency buffers of all the variables. We require that $B(x)$ contains write messages only on x. In fact, the variable name in a write message is redundant; we keep it only for clarity.

$$\frac{D(x) = \epsilon}{\langle D, E, N\rangle \xrightarrow{\langle \texttt{per}, x\rangle / -}_{②} \langle D, E, N\rangle}\ \text{Per} \qquad \frac{D' = D[x \leftarrow \langle \texttt{wr}, x, d\rangle \bullet D(x)]}{\langle D, E, N\rangle \xrightarrow{\langle \texttt{wr}, x, d\rangle / -}_{②} \langle D', E, N\rangle}\ \text{Write}$$

$$\frac{D(x) = w \bullet \langle \texttt{wr}, x, d\rangle \quad D' = D[x \leftarrow w] \quad E' = E[x \leftarrow \langle \texttt{wr}, x, d\rangle \bullet E]}{\langle D, E, N\rangle \xrightarrow{-/-}_{②} \langle D', E', N\rangle}\ \text{Propagate-Write}$$

$$\frac{E = w \bullet \langle \texttt{wr}, x, d\rangle \quad E' = w \quad N' = N[x \leftarrow d]}{\langle D, E, N\rangle \xrightarrow{-/-}_{②} \langle D, E', N'\rangle}\ \text{Update-Persistent-Memory}$$

The word $E \in (\mathbb{X} \times \mathbb{D})^*$ gives the content of the final persistency buffer. Finally, the function $N : \mathbb{X} \longrightarrow \mathbb{D}$ gives the contents of the persistent memory. We define $\Gamma^{②}_{init} := \{\langle D_{init}, E_{init}, N_{init}\rangle\}$, where $D_{init} = (\lambda x \in \mathbb{X}.\ \epsilon)$, $E_{init} = \epsilon$, and $N_{init} = (\lambda x \in \mathbb{X}.\ 0)$, i.e., initially, all the buffers are empty, and values of the value of each variable in the persistent memory is 0. The set of input events is identical to the set output events $\Sigma^{①}_{out}$. The set $\Sigma^{②}_{out}$ contains only the empty event $-$. The inference rules inducing the transition relation $\rightarrow_{②}$ are as follows. In Per, the automaton receives an event $\langle \mathtt{per}, x\rangle$ from the pending stage. The transition is enabled only if the x-initial persistency buffer is empty. This ensures that we create a barrier between the early write messages on x, and write messages that arrive later from the pending stage. In Write, the automaton receives an event indicating a write operation on a variable $x \in \mathbb{X}$, and the automaton appends the corresponding message to the end of the x-initial buffer. In Propagate-Write, the automaton non-deterministically removes the message at the head of one of the initial buffers, and appends it to tail of the final buffer. This means that the automaton keeps the order of write messages on the same variable, while it can re-order messages on distinct variables. The persistent memory stores the value of the variables to be recovered in case of a system crash. The automaton can non-deterministically fetch the last message in the final buffer, and use it to update the persistent memory (the rule Update-Persistent-Memory).

For a function $N : \mathbb{X} \longrightarrow \mathbb{D}$, giving the content of the persistent memory, we define $\mathcal{A}^{②}[N]$ to be the automaton we get from $\mathcal{A}^{②}$ by changing the initial persistent memory state to N.

5.3 The Reachability Problems

In the rest of this section, we fix a program P, with a finite set Θ of threads, and a set L of instruction labels. Let $\mathcal{A}^{Ⓟ}$ be the automaton induced by the program (see Sec. 4). In the *Crash-Free Reachability Problem*, we are given a configuration γ of $\mathcal{A}^{Ⓟ}$, and the question is whether $\mathcal{A}^{Ⓟ} \otimes \mathcal{A}^{①} \otimes \mathcal{A}^{②} \models \langle \gamma, \gamma_1, \gamma_2\rangle$, for some γ_1 and γ_2.

In the *Full Reachability Problem*, we consider full runs of the system including crashes. A recovery procedure is a function $\mathtt{Rec} : [\mathbb{X} \rightarrow \mathbb{D}] \longrightarrow [\Theta \rightarrow L]$. Intuitively, given the current state of the persistent memory $N : \mathbb{X} \longrightarrow \mathbb{D}$, the value of $\mathtt{Rec}\,(N)$ is labeling function λ that gives the re-starting instructions (new initial labels) of the threads after a crash from a configuration of the system. For a program P and a configuration $\gamma = \langle \gamma_P, \gamma_1, \gamma_2\rangle$ where $\gamma_2 = \langle D, E, N\rangle$, we define $P\Box\gamma$ to be the automaton $\left(\mathcal{A}^{Ⓟ}[\mathtt{Rec}\,(N)]\right) \otimes \left(\mathcal{A}^{①}[N]\right) \otimes \left(\mathcal{A}^{②}[N]\right)$. In the *Full-Reachability Problem*, we are given a configuration γ of $\mathcal{A}^{Ⓟ}$, as well as a recovery procedure Rec. The question is whether there exists a finite sequence $\mathcal{A}_0\gamma_0\mathcal{A}_1\gamma_1 \cdots\ \mathcal{A}_n\gamma_n$ such that the following conditions are satisfied:

- $\mathcal{A}_0 = \mathcal{A}^{Ⓟ} \otimes \mathcal{A}^{①} \otimes \mathcal{A}^{②}$
- $\mathcal{A}_i \models \gamma_i$
- $\mathcal{A}_{i+1} = P\Box\gamma_i$.
- $\gamma_n = \langle \gamma, \gamma_1, \gamma_2\rangle$, for some γ_1, γ_2.

5.4 Correctness

Let $\mathcal{A}^{Ⓞ}$ be the automaton corresponding to the persistency semantics of [Raad et al. 2020]. The full reachability and crash-free reachability problems are defined analogous to scheduling semantics. The crash-free reachability problem asks if a configuration of $\mathcal{A}^{Ⓟ}$ can be reached in $\mathcal{A}^{Ⓟ} \otimes \mathcal{A}^{Ⓞ}$. The full reachability problem asks whether a sequence of crashes and recoveries allow us to reach such a configuration. The following lemma states that each of these problems can be reduced to the corresponding problem in scheduling semantics.

Lemma 5.1. *The crash-free/full reachability problem on $\mathcal{A}^{Ⓟ} \otimes \mathcal{A}^{Ⓞ}$ reduces to the crash-free [full] reachability on $\mathcal{A}^{Ⓟ} \otimes \mathcal{A}^{①} \otimes \mathcal{A}^{②}$.*

Towards the proof of the lemma, notice that the program component of both the systems are the same. Hence it is enough to establish equivalence between $\mathcal{A}^{Ⓞ}$ and $\mathcal{A}^{①} \otimes \mathcal{A}^{②}$. As a first step, we reformulate the persistency semantics as a composition of pending stage and persistency stage (referred to as $\mathcal{A}^{Ⓑ}$ and $\mathcal{A}^{Ⓟ}$ respectively), this we do by introducing an intermediate memory component in the pending stage (i.e. to $\mathcal{A}^{Ⓑ}$). In the new setup, a transfer of a write from the pending stage to the persistence stage is also stored in the intermediate memory, allowing us to service the reads directly from the intermediate memory. This allows us to reformulate $\mathcal{A}^{Ⓞ}$ as $\mathcal{A}^{Ⓑ} \otimes \mathcal{A}^{Ⓟ}$. The equivalence of this reformulation is straightforward. With this, we can structure the proof of the Lemma 5.1 as two separate equivalences between $\mathcal{A}^{①}, \mathcal{A}^{Ⓑ}$ and $\mathcal{A}^{②}, \mathcal{A}^{Ⓟ}$.

Equivalence between $\mathcal{A}^{①}$ and $\mathcal{A}^{Ⓑ}$: Any access of the pending buffers in $\mathcal{A}^{①}$ confirms to the FIFO policy whereas the accesses to pending buffer in $\mathcal{A}^{Ⓑ}$ need not follow this criterion. The $\mathcal{A}^{①}$ uses the promise and delay flags to simulate the behaviours of $\mathcal{A}^{Ⓑ}$ while still confirming to the FIFO policy. Section 2.1 describes, through examples, the key ideas used in this simulation. A careful construction using those ideas leads to a simulation of each run ρ in $\mathcal{A}^{Ⓑ}$ by a run ρ' in $\mathcal{A}^{①}$. It is important to note that this simulating run is defined step by step but is based on the entire run ρ. The simulation satisfies the following at the end of each step:

- The inputs read so far are identical
- The output sequence in the simulation can be obtained from that of the simulated prefix of ρ by the deletion of some of the `per` events.

The actual invariant required for the proof is a significant strengthening of these requirements. This is necessary for it to be inductive and to guarantee that the simulation steps are indeed enabled. Among other things it asserts that the intermediate memory reached at the end of these partial runs is identical and that, for each thread, the contents of the pending buffer is identical if one ignores (i.e. project out) the fo messages. More intricate parts of the invariant pertain to the Delay and Promise flags.

For instance, it asserts that `Promise` $(\theta)(x)$ is set to true only if the tail end of $B(\theta)$ is part of a pivot x-sub-zone and that there is a subsequent $\langle \theta, \mathsf{fo}, x\rangle$ within that zone to release this obligation. It also asserts that `Delay` $(\theta)(x)$ is set to true only if the x-zone at the head of the pending buffer contains $\langle \mathsf{fo}, x\rangle$ which may then be used to discharge it.

The simulation of runs of $\mathcal{A}^{①}$ by $\mathcal{A}^{Ⓑ}$ is again carried out per run ρ of $\mathcal{A}^{Ⓑ}$ but defined step by step on its prefixes. The invariant in this case guarantees:

- The inputs read so far are identical
- The output sequence in the simulation can be obtained by stuttering the `per` events in the output sequence of the simulated prefix.

In summary

(1) If $\langle \alpha, \beta\rangle \in \left[\!\left[\mathcal{A}^{Ⓑ}\right]\!\right]$ then there is a $\langle \alpha, \beta'\rangle \in \left[\!\left[\mathcal{A}^{①}\right]\!\right]$, where β' is obtained by deleting some `per` events from β.

(2) If $\langle \alpha, \beta\rangle \in \left[\!\left[\mathcal{A}^{①}\right]\!\right]$, then there is a $\langle \alpha, \beta'\rangle \in \left[\!\left[\mathcal{A}^{Ⓑ}\right]\!\right]$, where β' which is obtained by stuttering some `per` events from β.

Equivalence between $\mathcal{A}^{②}$ and $\mathcal{A}^{Ⓟ}$: Recall that the persistency buffer in the $\mathcal{A}^{Ⓟ}$ is now replaced in $\mathcal{A}^{②}$ by a two level structure: initial persistency buffers consisting of a collection of FIFO channels, one per variable, which in turn feed the final persistency buffer consisting of a single FIFO channel.

The simulation of $\mathcal{A}^{Ⓟ}$ by $\mathcal{A}^{②}$ is achieved as follows: Once again, our simulation proceeds step by step based on the entire run. The $\mathcal{A}^{Ⓟ}$, on reading a $\langle \mathtt{wr}, x, d\rangle$ from its input, stores $\langle \mathtt{wr}, x, d\rangle$ in its persistency buffer. This is simulated by $\mathcal{A}^{Ⓟ}$by storing $\langle \mathtt{wr}, x, d\rangle$ in the initial buffer corresponding to x. On reading a $\langle \texttt{per}, x\rangle$ the simulation proceeds if the input buffer of x is empty. Finally writes to persistent memory are simulated by transferring the head of the buffer to the persistent memory (and the invariant of the simulation guarantees that the required value resides at the head of this buffer.). Further after each step, the simulation transfers values from the initial to the final persistency buffer provided their turn (based on the order in which they exited the persistency buffer in the run ρ) has arrived ensuring that values leave the final stage in FIFO order as well. We establish an invariant which guarantees after each simulation step that

- The inputs read are the same
- The state of the persistent memory is the same

The actual invariant is a significant strengthening, which includes properties such as: contents of the final buffer appear in the order in which they exited in ρ, the values present in the initial and final buffers are the same as those found in the persistency buffer and so on.

Reasoning along the same lines, a similar result can be established for the other direction also. With this, we have

(3) On any input α, the set of persistent memory configurations reachable under $\mathcal{A}^{Ⓟ}$ and $\mathcal{A}^{②}$ are the same.

We next put together the results obtained for the pending and persistent stages. To take into account the stuttering/deletion of per events in the simulation results for the pending stage we prove the following robustness results for $\mathcal{A}^{Ⓟ}$ and $\mathcal{A}^{②}$.

(4) Let α be an input to the persistent stage and α' obtained from α by stuttering some of the per events. Then, any configuration reachable under $\mathcal{A}^{Ⓟ}$ on input α is also reachable on input α'.
(5) let α be an input to the persistent stage and α' obtained by deleting some of the per events in α. Then, any configuration reachable under $\mathcal{A}^{②}$ on input α is also reachable on input α'.

Combining the above results, we see that on any input the set of persistent memory stages reachable in $\mathcal{A}^{①} \otimes \mathcal{A}^{②}$ and $\mathcal{A}^{Ⓑ} \otimes \mathcal{A}^{Ⓟ}$ are the same. This establishes Lemma 5.1.

6 THE REFINED SCHEDULING SEMANTICS

In this section, we give the automata for the refined scheduling semantics.

6.1 Load Pending Buffers

To describe the contents of the load buffers, we consider words $w \in G^*$ over the buffer alphabet $G = (\{\texttt{rd}, \texttt{srd}\} \times \mathbb{X} \times \mathbb{D}) \cup (\{\texttt{pfo}\} \times \mathbb{X})$. There are three types of messages in the alphabet. A *normal* read message $\langle \texttt{rd}, x, d\rangle$ corresponds to reading from a write instruction that was performed by another thread. A *self-read* message $\langle \texttt{srd}, x, d\rangle$ corresponds to reading from a write instruction that was performed by the thread itself. A *promise* message $\langle \texttt{pfo}, x\rangle$ corresponds to a promised $\texttt{flush}_{\texttt{opt}} x$ instruction. A *distinguished message* is either a self-read or a promise message. We say that w is *load-buffer word*, or simply an *LB-word* if it is of the form $w_1 m_1 w_2 m_2 w_3 \cdots m_{n-1} w_n$, where w_i contains only normal messages, and each m_i is a distinguished message. Furthermore, we require that $m_i \neq m_j$ if $i \neq j$. In other words, for each variable x, the word w contains at most one self-read message on x, and at most one pfo message on x. We define the *factorization* of w $\texttt{factor}(w) := [w_1][m_1][w_2][m_2][w_3] \cdots [m_{n-1}][w_n]$ (the brackets are for readability and have no semantical significance.) We use H to denote the set of LB-words.

The load pending automaton is of the form $\mathcal{A}^{③} = \left\langle \Gamma^{③}, \Gamma^{③}_{init}, \Sigma^{③}_{in}, \Sigma^{③}_{out}, \rightarrow_{③} \right\rangle$, defined as follows. A configuration in $\Gamma^{③}$ is a triple $\langle B, \texttt{Promise}, \texttt{Delay}, M\rangle$. The state of the load buffers is

$$\frac{\mathtt{Promise}(\theta)(x)=\mathtt{false} \quad B'=B[\theta\leftarrow B(\theta)\odot\langle\mathtt{srd},x,d\rangle] \quad M'=M[x\leftarrow d]}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{\langle\theta,\mathtt{wr},x,d\rangle/\langle\mathtt{wr},x,d\rangle}_{③} \langle B',\mathtt{Promise},\mathtt{Delay},M'\rangle}\ \text{Write}$$

$$\frac{B'=B[\theta\leftarrow B(\theta)\cdot\langle\mathtt{rd},x,d\rangle] \quad M'(x)=d}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{-/-}_{③} \langle B',\mathtt{Promise},\mathtt{Delay},M\rangle}\ \text{Back-Propagate-Read}$$

$$\frac{B(\theta)=m\cdot w \quad m\neq\langle\mathtt{pfo},*\rangle \quad B'=B[\theta\leftarrow w]}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{-/-}_{③} \langle B',\mathtt{Promise},\mathtt{Delay},M\rangle}\ \text{Delete-Msg}$$

$$\frac{B(\theta)=w_1\cdot\langle\mathtt{srd},x,d\rangle\cdot w_2}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{\langle\theta,\mathtt{rd},x,d\rangle/-}_{③} \langle B,\mathtt{Promise},\mathtt{Delay},M\rangle}\ \text{Read-Self}$$

$$\frac{B(\theta)=\langle\mathtt{rd},x,d\rangle\cdot w \quad \langle\mathtt{srd},x,*\rangle\notin B(\theta)}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{\langle\theta,\mathtt{rd},x,d\rangle/-}_{③} \langle B,\mathtt{Promise},\mathtt{Delay},M\rangle}\ \text{Read-Other}$$

$$\frac{\forall y\in\mathbb{X}.\,\mathtt{Promise}(\theta)(y)=\mathtt{false} \quad \forall y\in\mathbb{X}.\,\mathtt{Delay}(\theta)(y)=\mathtt{false} \quad B(\theta)=\epsilon \quad M(x)=d_1 \quad M'=M[x\leftarrow d_2]}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{\langle\theta,\mathtt{rmw},x,d_1,d_2\rangle/\langle\mathtt{wr},x,d_2\rangle}_{③} \langle B,\mathtt{Promise},\mathtt{Delay},M'\rangle}\ \text{Read-Modify-Write}$$

$$\frac{\mathtt{Delay}'=\mathtt{Delay}[\theta\leftarrow x\leftarrow\mathtt{true}]}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{\langle\theta,\mathtt{fo},x\rangle/-}_{③} \langle B,\mathtt{Promise},\mathtt{Delay}',M\rangle}\ \text{Delay-FO}$$

$$\frac{\mathtt{Promise}(\theta)(x)=\mathtt{false} \quad \mathtt{Delay}(\theta)(x)=\mathtt{true} \quad \mathtt{Delay}'=\mathtt{Delay}[\theta\leftarrow x\leftarrow\mathtt{false}]}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{-/\langle\mathtt{per},x\rangle}_{③} \langle B,\mathtt{Promise},\mathtt{Delay}',M\rangle}\ \text{Release-FO (1)}$$

$$\frac{\mathtt{Promise}(\theta)(x)=\mathtt{true} \quad \mathtt{Delay}(\theta)(x)=\mathtt{true} \quad B'=B[\theta\leftarrow B(\theta)\bullet\langle\mathtt{pfo},x\rangle] \quad \mathtt{Delay}'=\mathtt{Delay}[\theta\leftarrow x\leftarrow\mathtt{false}]}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{-/\langle\mathtt{per},x\rangle}_{③} \langle B',\mathtt{Promise},\mathtt{Delay}',M\rangle}\ \text{Release-FO (2)}$$

$$\frac{\mathtt{Promise}(\theta)(x)=\mathtt{false} \quad \mathtt{Promise}'=\mathtt{Promise}[\theta\leftarrow x\leftarrow\mathtt{true}] \quad \mathtt{Delay}'=\mathtt{Delay}[\theta\leftarrow x\leftarrow\mathtt{true}]}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{-/-}_{③} \langle B,\mathtt{Promise}',\mathtt{Delay}',M\rangle}\ \text{Make-Promise}$$

$$\frac{(\mathtt{Delay}(\theta)(x)=\mathtt{true})\wedge(\langle\mathtt{pfo},x\rangle\notin B(\theta)) \quad \mathtt{Promise}(\theta)(x)=\mathtt{true} \quad \mathtt{Promise}'=\mathtt{Promise}[\theta\leftarrow x\leftarrow\mathtt{false}]}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{\langle\theta,\mathtt{fo},x\rangle/-}_{①} \langle B,\mathtt{Promise}',\mathtt{Delay},M\rangle}\ \text{Discharge-Promise (1)}$$

$$\frac{B(\theta)=w_1\cdot\langle\mathtt{pfo},x\rangle\cdot w_2 \quad B'=B[\theta\leftarrow w_1\cdot w_2] \quad \mathtt{Promise}'=\mathtt{Promise}[\theta\leftarrow x\leftarrow\mathtt{false}]}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{\langle\theta,\mathtt{fo},x\rangle/-}_{③} \langle B',\mathtt{Promise}',\mathtt{Delay},M\rangle}\ \text{Discharge-Promise(2)}$$

$$\frac{\mathtt{Promise}(\theta)(x)=\mathtt{true}}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{\langle\theta,\mathtt{fo},x\rangle/-}_{③} \langle B,\mathtt{Promise},\mathtt{Delay},M\rangle}\ \text{Merge-FO}$$

$$\frac{\mathtt{Promise}(\theta)(x)=\mathtt{false} \quad \mathtt{Delay}(\theta)(x)=\mathtt{false}}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{\langle\theta,\mathtt{fl},x\rangle/\langle\mathtt{per},x\rangle}_{③} \langle B,\mathtt{Promise},\mathtt{Delay},M\rangle}\ \text{FL}$$

$$\frac{\forall y\in\mathbb{X}.\,\mathtt{Promise}(\theta)(y)=\mathtt{false} \quad \forall y\in\mathbb{X}.\,\mathtt{Delay}(\theta)(y)=\mathtt{false}}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{\langle\theta,\mathtt{sf}\rangle/-}_{③} \langle B,\mathtt{Promise},\mathtt{Delay},M\rangle}\ \text{SF}$$

$$\frac{\forall y\in\mathbb{X}.\,\mathtt{Promise}(\theta)(y)=\mathtt{false} \quad \forall y\in\mathbb{X}.\,\mathtt{Delay}(\theta)(y)=\mathtt{false} \quad B(\theta)=\epsilon}{\langle B,\mathtt{Promise},\mathtt{Delay},M\rangle \xrightarrow{\langle\theta,\mathtt{mf}\rangle/-}_{③} \langle B,\mathtt{Promise},\mathtt{Delay},M\rangle}\ \text{MF}$$

Fig. 15. The load buffer automaton.

defined by $B : \Theta \to H$. The transition system maintains the invariant that each pending load buffer contains an LB-word. The `Promise`, `Delay`, and the M functions define the delay flags resp. the contents of the intermediate memory as before. The set $\Gamma^{③}_{init}$ of initial configurations $\{\langle B_{init}, \mathtt{Promise}_{init}, \mathtt{Delay}_{init}, M_{init}\rangle\}$ is defined as before. In particular, we start with empty load buffers. The sets of input events $\Sigma^{③}_{in}$, and output events $\Sigma^{③}_{out}$ are identical to their counter-parts $\Sigma^{①}_{in}$ resp. $\Sigma^{①}_{out}$. The transition relation $\rightarrow_{③}$ is defined through the inference rules shown in Fig. 15, for a thread $\theta \in \Theta$.

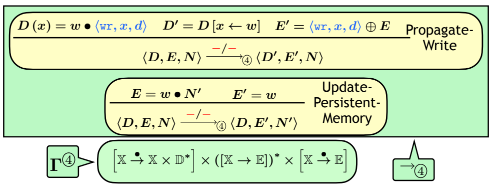
For $w \in H$, representing the content of a load buffer, we define $w \odot \langle \mathtt{srd}, x, d\rangle := w'$ where $w' = w_1 \cdot w_2 \cdot \langle \mathtt{srd}, x, d\rangle$ if $w = w_1 \cdot \langle \mathtt{srd}, x, d'\rangle \cdot w_2$ for some $d' \in \mathbb{D}$, and $w' = w \cdot \langle \mathtt{srd}, x, d\rangle$ if there is no self-read message on x in w. Notice that the operation maintains the invariant that the buffer contains an LB-word. In `Write`, the automaton receives an event $\langle \theta, \mathtt{wr}, x, d\rangle$, and performs three operations simultaneously: (i) it updates the intermediate memory, (ii) it notifies the persistency stage through the event $\langle \mathtt{wr}, x, d\rangle$, and (iii) it sends a self-read message $\langle \mathtt{srd}, x, d\rangle$ to itself. To emphasize the fact that messages travel in the reverse direction, we let the messages be added from the right (which is now considered to be the tail), and fetched from the left (which is now the head) of the load buffer. The transition may be performed only if there is no discharged promise on x, i.e., there is no pfo message on x in the load buffer. At any point, the intermediate memory may non-deterministically select the thread θ and a variable x and send the value of x to θ. This creates a normal read message on x that is appended to the end of the buffer of θ (the rule `Back-Propagate-Read`). Notice that such a message could have been created by the thread itself: we create a self-read message only in the `Write` rule. This simplifies the set of rules and it does not affect the analysis. In `Read-Self`, θ reads from the self-read message on the x, if such a message is in the buffer. If such a message is missing, and if the last read message in the buffer is on x then θ can read the value of x from that message (the rule `Read-Other`). The thread can non-deterministically remove the message at the head of its buffer and throw it away (`Delete-Msg`). The rule `Read-Modify-Write` is as before, i.e., the included read and write operations are carried out simultaneously, and we require that the buffer is empty, and that the delay flags are false.

The next six rules handle promising and delaying the $\texttt{flush}_{\texttt{opt}}$ instruction. When θ performs a $\texttt{flush}_{\texttt{opt}}$ on a variable $x \in \mathbb{X}$ it is delayed immediately, and in contrast to the basic semantics, it is not inserted in the buffer (`Delay-FO`). As in the basic semantics, a delayed instruction can be released non-deterministically to the persistency stage (`Release-FO`). The thread can promise a $\texttt{flush}_{\texttt{opt}}$, by setting the promise flag, while delaying the event rather than sending it immediately to the persistency stage (`Make-Promise`). A promise on a variable x can be discharged when the $\texttt{flush}_{\texttt{opt}} x$ instruction is executed by θ, either by removing the (only) `pfo` message, or checking that the delay flag is `true`. As before, a promise need not be discharged if a $\texttt{flush}_{\texttt{opt}} x$ occurs.

The last three rules concern other flush and fence instructions. In `FL`, the automaton receives an event $\langle \theta, \texttt{fl}, x \rangle$. In a similar manner to a $\texttt{flush}_{\texttt{opt}}$ instruction, we do not put any message in the buffer. However, we still require that there is no delay or promise on the variable x. The rule `SF` and `MF` are straightforward adaptations from the basic semantics.

6.2 Snapshot Buffer

For a write message $m = \langle \texttt{wr}, x, d \rangle$ and a non-empty word $w \in [\mathbb{X} \rightarrow \mathbb{D}]^+$ of memory states (snapshots), with $w = N \cdot w'$, we define $m \oplus w := (N[x \leftarrow d]) \cdot w$, i.e., we construct a new snapshot by considering the last element N in w and update the value of x to d, and then append the new snapshot to w. In the figure, we show only the parts that are different from the basic semantics The `Propagate-Write` rule still removes a write message on x from the initial x-initial buffer. However, instead of simply transferring the message to the final persistency buffer, it computes its addition to the current content of the snapshot buffer and transfers the result to the latter. Furthermore, in `Update-Persistent-Memory`, we copy the entire snapshot to the memory rather than just the value of a single variable.


$D(x) = w \bullet \langle \texttt{wr}, x, d \rangle \quad D' = D[x \leftarrow w] \quad E' = \langle \texttt{wr}, x, d \rangle \oplus E$
$\langle D, E, N \rangle \xrightarrow{-/-}_{④} \langle D', E', N \rangle$
Propagate-Write
$E = w \bullet N' \quad E' = w$
$\langle D, E, N \rangle \xrightarrow{-/-}_{④} \langle D, E', N' \rangle$
Update-Persistent-Memory
$\Gamma^{④}$
$\left[\mathbb{X} \overset{\bullet}{\rightarrow} \mathbb{X} \times \mathbb{D}^*\right] \times ([\mathbb{X} \rightarrow \mathbb{E}])^* \times \left[\mathbb{X} \overset{\bullet}{\rightarrow} \mathbb{E}\right]$
$\rightarrow_{④}$

Lemma 6.1. *The crash-free/full reachability problems from $\mathcal{A}^{\textcircled{P}} \otimes \mathcal{A}^{①} \otimes \mathcal{A}^{②}$ reduce to the corresponding problems for $\mathcal{A}^{\textcircled{P}} \otimes \mathcal{A}^{③} \otimes \mathcal{A}^{④}$.*

Towards the correctness, we show equivalence of $\mathcal{A}^{\textcircled{P}} \otimes \mathcal{A}^{①} \otimes \mathcal{A}^{②}$ and $\mathcal{A}^{\textcircled{P}} \otimes \mathcal{A}^{③} \otimes \mathcal{A}^{④}$. While the equivalence of $\mathcal{A}^{②}$ and $\mathcal{A}^{④}$ is straightforward, we provide the correctness argument for equivalence of $\mathcal{A}^{\textcircled{P}} \otimes \mathcal{A}^{①}$ and $\mathcal{A}^{\textcircled{P}} \otimes \mathcal{A}^{③}$ below. The equivalence of the automaton $\mathcal{A}^{\textcircled{P}} \otimes \mathcal{A}^{①}$ to $\mathcal{A}^{\textcircled{P}} \otimes \mathcal{A}^{③}$ is best seen as a series of transformations involving a number of intermediate semantics. Our multi-step approach allows us to treat the different concerns separately. In the transformations below, we will assume a fixed $\mathcal{A}^{\textcircled{P}}$ and proceed with the explanations. We mainly show the following transformations

- Transformation of $\mathcal{A}^{①}$ into an equivalent enriched TSO model
- Transformation of enriched TSO model into an equivalent enriched Load Buffer model
- Refining the enriched Load Buffer model into $\mathcal{A}^{③}$

Transforming $\mathcal{A}^{②}$ into an Enriched TSO Model . Recall that the standard TSO model consists of a channel per thread into which writes are deposited (at the tail) and the memory is updated from values drawn from the head of these buffers. It also includes memory fence instructions. In $\mathcal{A}^{①}$, we have reads, writes, memory fence (all of which find a place in the TSO model) and in addition there are other instructions such as sfence, $\texttt{flush}_{\texttt{opt}}$, flush as well as the flags promise and delay.

Our idea is to incorporate all these additional items into the TSO as (reads and writes to) new variables. Setting the flag $\texttt{Promise}(x)(\theta)$ (in $\mathcal{A}^{①}$) will now be represented as writing true/false to a new variable $\texttt{Promise}(x)(\theta)$, while testing its value would correspond to a read. The instruction flush/sfence is to be treated as writing (some fixed value) to the variable $\langle x, \texttt{fl}, \theta\rangle / \langle \theta, \texttt{sf}\rangle$ (which has the effect of inserting it in the TSO buffer thus simulating the corresponding move in the scheduling semantics). It turns out that we can use a variable $\texttt{Delay}(x)(\theta)$ to capture both the instruction $\texttt{flush}_{\texttt{opt}}$ as well as the flag $\texttt{Delay}(x)(\theta)$.

Memory updates on these new variables can be used to denote the transmission of values from the pending stage to the persistent stage. For instance, the memory update on the variable $\langle x, \texttt{fl}, \theta\rangle$ captures the transfer of a $\langle \texttt{fl}, x\rangle$ from the pending buffer to the output as a $\langle \texttt{per}, x\rangle$ in $\mathcal{A}^{①}$. To simulate the discharge of a delay (and the output of a per), we incorporate a new special process with the responsibility to reset this variable to false through an rmw instruction.

The main obstacle to such a direct coding into TSO is the requirement in $\mathcal{A}^{①}$ to test side conditions at output transitions: for instance, to transfer sf from the pending buffer to the output we have to verify whether $\texttt{Delay}(x)$ is false for all x. Similarly to transfer a $\langle \texttt{fl}, x\rangle$ as a $\langle \texttt{per}, x\rangle$ we have to verify that $\texttt{Delay}(x)$ is false. To handle such tests we enrich the TSO model, adding power to the memory updates. The new model allows us to specify, for any variable, an enabling boolean condition (on the memory) that must hold for its value to be updated in the memory. We do not have to concern ourselves with the decidability of this enriched model as our aim is to use this merely as a step in the translation eventually to the refined scheduling semantics.

Equivalence of Enriched TSO Model with Enriched Load Buffer Model. The second stage involves moving from this enriched TSO model where writes are stored in buffers while reads are instantaneous (*store buffers*) to one where writes happen instantaneously while reads pick up values from buffers (*load buffers*). In a load buffer model, values are periodically back propagated from the memory to the load buffers so that a thread may read an outdated value from this buffer. We enrich this traditional load buffer model as we enriched the TSO, allowing us to specify, for each variable, an enabling boolean condition to be evaluated on the memory before writing to it. The equivalence between TSO and load buffer (LB)models was shown in [Abdulla et al. 2018a]. The proofs extend easily to an equivalence between the enriched TSO model and enriched load buffer model. Once again, we do not have to concern ourselves with the decidability of the enriched LB model as it is used only in the translation to the refined scheduling semantics.

Refining the Enriched Load Buffer Model into $\mathcal{A}^{③}$. The Enriched Load Buffer model while faithfully simulating $\mathcal{A}^{①}$ still uses additional variables and these appear in the load buffers making it different from $\mathcal{A}^{③}$. The last stage transforms this enriched load buffer model into the refined scheduling semantics by eliminating the need to back propagate these new variables to the load buffer. The elimination of the back propagation of most of the new variables can be handled via very simple observations : for instance, the variables $\langle x, \texttt{fl}, \theta\rangle$, $\langle \theta, \texttt{sf}\rangle$ are not read at all obviating any reason for their back propagation. For $\texttt{Promise}(x)(\theta)$ we make use of the fact that it is modified and read only by θ.

The last and somewhat intricate step is to show that the back propagation of the variables $\texttt{Delay}(x)(\theta)$ can be handled by using at most one $\langle \texttt{pfo}, x\rangle$ in the load buffer of thread θ. A rough intuition is the following: this variable is written to by only two processes, the thread θ which only writes true and the special process that only resets it through a rmw instruction. The special thread accesses this variable only through a rmw instruction. Thus, this variable needs to be back propagated only to thread θ. We would like to eliminate this as well and allow it to read directly from the memory. However, there is a very specific circumstance in which this results in a problem. Since θ only reads the value true attempting to read a wrong value can only disable an otherwise enabled read. And this happens only in a very specific scenario i.e., when it is still reading values from its load buffer that arrived earlier than last time the special process set this variable to false. We are able to capture this situation by inserting the $\langle \mathtt{pfo}, x\rangle$ as a fence into the load buffer at the execution of the `rmw` instruction. This then is essentially a reformulation of $\mathcal{A}^{③}$.

Thus, we are able to demonstrate the equivalence between $\mathcal{A}^{①}$ and $\mathcal{A}^{③}$ models when executed in composition with any program giving us the correctness of Lemma 6.1.

Remark on FIFO Buffers. Some aspects of the semantics may give the impression that the load buffer is not purely FIFO. For instance the rule *Discharge-Promise(2)* seems to perform a non-FIFO operation since it deletes a message from inside the load buffer. However, this is not true. More precisely, we divide the load buffer to a *finite* sequence of sub-buffers, each of which is a FIFO buffer. The buffers are separated by messages from the set of distguished messages which is finite. This is reflected in the factorization operation, which is the basis the ordering relation with respect to which the system is monotone (Sec. 7).

7 DECIDABILITY

In this section, we give an overview of the decidability of the reachability problems, by instantiating the framework of well-structured systems [Abdulla et al. 1996; Finkel and Schnoebelen 2001]. In the rest of the section, we fix an *ordered automaton* $\langle \Gamma, \Gamma_{init}, \Sigma_{in}, \Sigma_{out}, \rightarrow, \sqsubseteq\rangle$, i.e., an automaton that is equipped with a pre-order $\sqsubseteq$ on the set Γ.

Well-structured Automata. For events $e_1, e_2 \in \Sigma_{in} \cup \Sigma_{out}$, we use $\overset{e_1/e_2}{\Longrightarrow}$ to denote $(\xrightarrow{-/-})^* \circ \xrightarrow{e_1/e_2} \circ (\xrightarrow{-/-})^*$. If $\gamma_1 \overset{e_1/e_2}{\Longrightarrow} \gamma_2$ then we can move from γ_1 to γ_2 by performing a transition $\gamma_1 \xrightarrow{e_1/e_2} \gamma_2$, preceded and followed by an arbitrary number of transitions in which the automaton does not interact with the environment during the transition. For a set $G \subseteq \Gamma$ of configurations, we define its *predecessor set* as $Pred\,(e_1/e_2)\,(G) := \left\{\gamma \mid \exists \gamma' \in G.\ \gamma \overset{e_1/e_2}{\Longrightarrow} \gamma'\right\}$. We define the *upward closure* of G as $\widehat{G} := \{\gamma' \mid \exists \gamma \in G.\ \gamma \sqsubseteq \gamma'\}$. We say $\mathcal{A}$ is *well-structured* if it satisfies the following conditions:

- $\sqsubseteq$ is a well quasi-ordering, i.e., for any infinite sequence $\gamma_0, \gamma_1, \gamma_2, \ldots$ of configurations, there are i, j with $i < j$ and $\gamma_i \sqsubseteq \gamma_j$.
- $\rightarrow$ is monotone wrt. $\sqsubseteq$, i.e., given configuration $\gamma_1, \gamma_2, \gamma_3$ such that $\gamma_1 \xrightarrow{e_1/e_2} \gamma_2$ and $\gamma_1 \sqsubseteq \gamma_3$, there is a configuration γ_4 such that $\gamma_3 \overset{e_1/e_2}{\Longrightarrow} \gamma_4$ and $\gamma_2 \sqsubseteq \gamma_4$.
- For a finite set $G \subseteq \Gamma$, we can compute a finite encoding of the predecessor set $Pred\,(e_1/e_2)\left(\widehat{G}\right)$.

Theorem 7.1. *For a well-structured automaton $\mathcal{A}$ and a finite set $G \subseteq \Gamma$, $\mathcal{A} \models \widehat{G}$ is decidable.*

Orderings. The main step in proving well-structuredness of our automata is to provide a well quasi-ordering $\sqsubseteq$, such that the transition relation is monotone wrt. $\sqsubseteq$. Since, we are working with FIFO operations, and all the operations are local, computing the predecessor set amounts to standard operations on finite words [Abdulla and Jonsson 1993]. The set $\Gamma^{Ⓟ}$ of configurations in the automaton $\mathcal{A}^{Ⓟ}$ is finite, and therefore we can trivially define $\sqsubseteq$ to be the equality relation on $\Gamma^{Ⓟ}$. For the automaton $\mathcal{A}^{③}$ we define the ordering $\sqsubseteq$ on the set $\Gamma^{③}$ in two steps. First, we define an ordering $\sqsubseteq$ on the set H of LB-words, using their factorizations. Let $w, w' \in H$, with $\mathtt{factor}\,(w) = [w_1][m_1][w_2][m_2][w_3]\cdots[m_{n-1}][w_n]$, and $\mathtt{factor}\,(w') = [w'_1][m'_1][w'_2][m'_2][w'_3]\cdots[m'_{n-1}][w'_n]$. We write $w \sqsubseteq w'$ to denote that (i) $m_i = m'_i$ for all $i : 1 \leq i < n$, and (ii) $w_i \preceq w'_i$ for all $i : 1 \leq i \leq n$. In other words, we require that w and w' agree on the values and the orderings of the distinguished messages, and furthermore, each segment in w is a sub-word of the corresponding segment in w' (recall the definition of $\preceq$ from Sec. 3.1.) For configurations $\gamma = \langle B, \mathtt{Promise}, \mathtt{Delay}, M\rangle$ and $\gamma' = \langle B', \mathtt{Promise}', \mathtt{Delay}', M'\rangle$ in $\Gamma^{③}$, we write $\gamma \sqsubseteq \gamma'$ to denote that (i) $B(\theta) \sqsubseteq B'(\theta)$ for all $\theta \in \Theta$, (ii) $\mathtt{Promise}' = \mathtt{Promise}$, (iii) $\mathtt{Delay} = \mathtt{Delay}'$, and (iv) $M = M'$. For configurations $\gamma = \langle D, E, N\rangle$ and $\gamma' = \langle D', E', N'\rangle$ in $\Gamma^{④}$, we write $\gamma \sqsubseteq \gamma'$ to denote that (i) $D(x) \preceq D'(x)$ for all $x \in \mathbb{X}$. (ii) $E \preceq E'$. (iii) $N = N'$. The well-structuredness of $\mathcal{A}^{Ⓟ}$, $\mathcal{A}^{③}$, and $\mathcal{A}^{④}$ implies the same for that $\mathcal{A}^{Ⓟ} \otimes \mathcal{A}^{③} \otimes \mathcal{A}^{④}$.

Decidability. The set of configurations for which we want to check reachability can always be defined as the upward closure $\widehat{G}$ of a finite set G of configurations. To see this, we observe that, for any buffer, e.g., a load , initial, or snapshot buffer, the set of *all* buffer states is the upward closure $\widehat{\{\epsilon\}}$ (wrt. $\preceq$). This means that $\mathcal{A}^{Ⓟ} \otimes \mathcal{A}^{③} \otimes \mathcal{A}^{④} \models \langle \gamma, \gamma_3, \gamma_4\rangle$ for some γ_3 and γ_4 is equivalent to $\mathcal{A}^{Ⓟ} \otimes \mathcal{A}^{③} \otimes \mathcal{A}^{④} \models \widehat{G}$. Here, $G \subseteq \{\gamma\} \times \Gamma^{③} \times \Gamma^{④}$, and all the buffers in the configurations of G are empty. Notice that this implies that G is finite. From this and Theorem 7.1 we get:

Lemma 7.2. *We can decide whether* $\exists \gamma_3 \gamma_4.\ \mathcal{A}^{Ⓟ} \otimes \mathcal{A}^{③} \otimes \mathcal{A}^{④} \models \langle \gamma, \gamma_3, \gamma_4\rangle$.

From Lemmata 6.1 , 7.2, we get:

Theorem 7.3. *The crash-free and full reachability problems are decidable*

8 CONCLUSION

We proved that verifying reachability in concurrent programs under the Px86 model is decidable. This is the first result on the decidability of this problem in the context of persistent memory models. Our approach for achieving this result is based on establishing a clear separation between the pending and the persistency stages. Besides decidability, our work provides an insight about the computational power of the persistent TSO model through the investigation and the definition of a new operational model that can be of independent interest for program designers.

Understanding the computational power of persistent TSO is also the first step towards developing efficient methods and tools for the verification of programs under this memory model. It opens the door to the investigation of various types of verification procedures combining our construction with techniques such as bounded analysis or DPOR strategies, following the lines of works developed, e.g., in [Abdulla et al. 2015, 2017b, 2016, 2018b; Atig et al. 2014; Kokologiannakis et al. 2019], for other memory models such as TSO, Power, or C11.

ACKNOWLEDGMENTS

This research was partially supported by Infosys (India), the Department of Science and Technology (India), the Swedish Research Council (Sweden) and the Project AdeCoDS of the french National Research Agency ANR (France).

REFERENCES

P.A. Abdulla, S. Aronis, M. Faouzi Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *TACAS (LNCS, Vol. 9035)*. Springer, 353–367.

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2017a. Stateless model checking for TSO and PSO. *Acta Inf.* 54, 8 (2017), 789–818.

Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankara Narayanan Krishna. 2019. Verification of programs under the release-acquire semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1117–1132. https://doi.org/10.1145/3314221.3314649

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, Egor Derevenetc, Carl Leonardsson, and Roland Meyer. 2020. Safety Verification under Power. In *NETYS 2020 (Lecture Notes in Computer Science)*. Springer.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2017b. Context-Bounded Analysis for POWER. In *TACAS*. 56–74.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2018a. A Load-Buffer Semantics for Total Store Ordering. *Logical Methods in Computer Science* 14, 1 (2018). https://doi.org/10.23638/LMCS-14(1:9)2018

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. 2016. Stateless Model Checking for POWER. In *CAV (LNCS, Vol. 9780)*. 134–156.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018b. Optimal stateless model checking under the release-acquire semantics. *PACMPL* 2, OOPSLA (2018), 135:1–135:29. https://doi.org/10.1145/3276505

Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. 1996. General Decidability Theorems for Infinite-State Systems. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*. IEEE Computer Society, 313–321. https://doi.org/10.1109/LICS.1996.561359

Parosh Aziz Abdulla and Bengt Jonsson. 1993. Verifying Programs with Unreliable Channels. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993*. IEEE Computer Society, 160–170. https://doi.org/10.1109/LICS.1993.287591

Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. 2013b. Software Verification for Weak Memory via Program Transformation. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 512–532. https://doi.org/10.1007/978-3-642-37036-6_28

Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013a. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 141–157. https://doi.org/10.1007/978-3-642-39799-8_9

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74.

ARM. 2018. ARM architecture reference manual ARMv8, for ARMv8-A architecture profile (DDI 0487D.a).

Joy Arulraj and Andrew Pavlo. 2017. How to Build a Non-Volatile Memory Database Management System. In *SIGMOD*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM.

Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak memory models. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 7–18. https://doi.org/10.1145/1706299.1706303

Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2012. What's Decidable about Weak Memory Models?. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 26–46. https://doi.org/10.1007/978-3-642-28869-2_2

Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. 2011. Getting Rid of Store-Buffers in TSO Analysis. In *CAV (LNCS, Vol. 6806)*. Springer, 99–115.

Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. 2014. Context-Bounded Analysis of TSO Systems. In *From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis, Grenoble, France, April 6, 2014. Proceedings*. 21–38.

M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. 2011. Mathematizing C++ concurrency. In *POPL*. ACM, 55–66.

Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Foundations and Trends in Programming Languages* 1, 1-2 (2014), 1–150.

Nachshon Cohen, David T. Aksun, and James R. Larus. 2018. Object-oriented recovery for non-volatile memory. *PACMPL* 2, OOPSLA (2018).

Brian Demsky and Patrick Lam. 2015. SATCheck: SAT-directed stateless model checking for SC and TSO. In *OOPSLA*. ACM, 20–36.

Alain Finkel and Philippe Schnoebelen. 2001. Well-structured transition systems everywhere! *Theor. Comput. Sci.* 256, 1-2 (2001), 63–92. https://doi.org/10.1016/S0304-3975(00)00102-X

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *POPL*. ACM, 608–621.

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm strong enough: reasoning about consistency choices in distributed systems. In *POPL 2016*. 371–384.

Intel. 2019a. Architectures Software Developer's Manual (Combined Volumes). Software.intel.com.

Intel (Ed.). 2019b. *Intel 64 and IA-32 Architectures Software Developer's Manual (Combined Volumes).* Intel.

Intel. 2019c. Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL 2017*. 175–189.

Artem Khyzha and Ori Lahav. 2021. Taming x86-TSO Persistency. *Proc. ACM Program. Lang.* 5, POPL, Article 47 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434328

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *PACMPL* 2 (2018), 17:1–17:32.

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Effective lock handling in stateless model checking. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 173:1–173:26. https://doi.org/10.1145/3360599

Michalis Kokologiannakis and Viktor Vafeiadis. 2020. HMC: Model Checking for Hardware Memory Models. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020 [ASPLOS 2020 was canceled because of COVID-19]*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 1157–1171. https://doi.org/10.1145/3373376.3378480

Ori Lahav and Udi Boker. 2020. Decidable verification under a causally consistent shared memory. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 211–226. https://doi.org/10.1145/3385412.3385966

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 649–662.

Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas F. Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross Failure Bug Detection in Persistent Memory Programs. In *ASPLOS*.

Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Manabi Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 411–425. https://doi.org/10.1145/3297858.3304015

Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An Axiomatic Memory Model for POWER Multiprocessors. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 495–512.

Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An operational semantics for C/C++11 concurrency. In *OOPSLA*. ACM, 111–128.

Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *ISCA*.

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.* 3, POPL (2019), 69:1–69:31. https://doi.org/10.1145/3290382

Azalea Raad and Viktor Vafeiadis. 2018. Persistence semantics for weak memory: integrating epoch persistency with the TSO memory model. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 137:1–137:27. https://doi.org/10.1145/3276507

Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency semantics of the Intel-x86 architecture. *PACMPL* 4, POPL (2020), 11:1–11:31. https://doi.org/10.1145/3371079

Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak persistency semantics from the ground up: formalising the persistency semantics of ARMv8 and transactional models. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 135:1–135:27. https://doi.org/10.1145/3360561

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

Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *USENIX ATC*, Dilma Da Silva and Bryan Ford (Eds.).