Brief Announcement: Time, Fences and the Ordering of Events in TSO

Raïssa Nataf
Technion, Haifa, Israel

Yoram Moses
Technion, Haifa, Israel

Abstract

Total Store Order (TSO) is one of the most popular relaxed memory model in multiprocessor architectures, widely implemented, for example, in Intel's x86 and x64 platforms. It delays write visibility via store buffers, thereby allowing a significant improvement in efficiency. This, however, complicates reasoning about correctness, as executions may violate sequential consistency. We present a semantic framework that provides effective tools that can pinpoint when such synchronization is necessary under TSO. We define a TSO-specific *occurs-before* relation, adapting Lamport's happens-before to TSO, and prove that events at different sites can be temporally ordered only via an occurs-before chain. Analyzing how fences and `RMW`s create these chains lets us identify when they are unavoidable. We present in this BA how these results impact linearizable implementations of registers, capturing information flow and causality in TSO. The full version of this work provides details as well as results regarding the need for synchronization in linearizable implementations of additional objects.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Concurrent algorithms

Keywords and phrases TSO, linearizability, happens before, fences, synchronization actions

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.62

Related Version *Full Version*: `https://arxiv.org/abs/2508.11415` [15]

1 Introduction and Related Work

Modern multiprocessors rely on relaxed memory models to improve performance through techniques such as store buffering and out-of-order execution. Among these, *Total Store Order (TSO)* – used in Intel's x86 and x64 architectures – is one of the most widely deployed. TSO increases efficiency by allowing writes to be held temporarily in per-process store buffers, deferring their visibility to other processors. However, this optimization breaks *sequential consistency* (SC) [11], making reasoning about correctness more difficult: a program correct under SC may fail under TSO due to delayed write visibility. To ensure correct behavior under TSO, programmers can use synchronization primitives such as *memory fences* (`F`) and *read-modify-write* (`RMW`) operations. These enforce memory ordering by flushing buffers or performing atomic accesses. While effective, they limit concurrency and reduce performance [4, 9, 17]. Determining *when such synchronization is truly necessary* is therefore a central question in the study of weak memory models such as TSO.

Lamport's *happens-before* relation [10] is central in asynchronous computing, underlying vector clocks, race detection, causal memory [12, 7, 1], and more. Happens-before was originally defined in asynchronous message-passing models, and it captures all of the information about timing that processes can obtain in such settings. In a recent paper, we proved a theorem called Delaying the Future (DtF) that provides a close formal connection between happens-before and the ability to reorder actions and events in asynchronous message-passing systems [14]. Roughly speaking, DtF implies that if there is no message chain between an

operation A and an operation B in a given run r, then there is a run r' that is indistinguishable to all processes from r in which A takes place after B does in real-time. This is of importance, for example, when considering linearizable implementations of concurrent objects, since the real-time order of event invocations and completions plays a central role in the definition of Linearizability [8]. The happens-before relation has been considered in different models, including TSO [3]. In this work, we introduce a new definition for the analogue of happens-before in TSO, which we call the *occurs-before* relation. One of the differences is that our definition stems from an operational model of TSO, whereas previous definitions such as in [3] are formulated using declarative models. As a result, our approach employs a lower level of abstraction, making it easier to derive concrete implementation constraints. We demonstrate the analogy between the classical happens-before relation of message chains to our occurs-before relation by proving a version of the DtF theorem for TSO, which is obtained by replacing "happens before" by "occurs before." This makes it possible to prove necessary conditions on the use of synchronization in linearizable TSO implementations, much in the spirit and style that [14] does for linearizable register implementations in asynchronous message passing.

Related Work. Prior work has tackled the need for synchronization actions in weak memory models from both practical and theoretical perspectives. Automated tools attempt to minimize fences while preserving correctness [2, 17], and impossibility results show that synchronization is sometimes unavoidable.

Attiya et al. [4] and Castañeda et al. [6] are most directly related to our TSO results. Both works establish conditions on when linearizable implementations require the use of synchronization operations. In [4], covering arguments are used to show that precise communication patterns involving reads and writes are unavoidable in implementing classic and widely used specifications. Their results apply to objects whose methods are strongly non-commutative – such as sets, queues, or stacks – but not to registers. The more recent [6] proves a mergeability theorem for TSO and related weak memory models traces and applies it to obtain results about objects with one-sided non commutative methods such as registers. They show that *linearizable* TSO implementations of objects, including registers, must use fences or `RMW`s in some executions. In [14], a DtF theorem is proved for asynchronous message-passing systems. It is then used to prove the need to construct message chains between operations in linearizable register implementations in that model.

The current paper can be viewed as performing an analogous analysis for TSO to that of [14]. Our analysis highlights the role of synchronization primitives, extending lower bounds on the use of synchronization primitives in TSO. Practically, such lower bounds identify the synchronization mechanisms implementations must employ and indicate when further attempts to remove them would be futile.

2 Model and Preliminary Definitions

This section outlines the main features of our model; the reader should consult the full version [15] for additional details. The model is based on the operational definition of TSO given in [6]. (Similar, though slightly different, operational models for TSO have appeared, e.g., in [16, 5].) It consists of a set $\Pi = \{1, \ldots, n\}$ of n processes and a finite set Var of variables. A basic TSO state is a pair $\sigma = \langle \mathsf{m}, \mathsf{buf} \rangle$, where $\mathsf{m} \in \mathsf{Var} \to \mathsf{Val}$ describes the main memory and $\mathsf{buf} \in \Pi \to (\mathsf{Var} \times \mathsf{Val})^*$ assigns a queue called a *store buffer* to every process. Processes can perform actions from the set $\{\mathtt{R}[x], \mathtt{W}[x, v], \mathtt{F}, \mathtt{RMW}[x, v_{\mathsf{exp}}, v_{\mathsf{new}}], \bot\}$. These correspond,

respectively, to reading the value of x (from local buffer if it contains an unexecuted write to x and from the physical variable x in memory otherwise), writing the value v in x (v is appended to the local buffer), performing a *fence* (which occurs only when the local buffer is empty), performing a read-modify-write action $\texttt{RMW}[x, v_{\mathsf{exp}}, v_{\mathsf{new}}]$ (which occurs only if the buffer is empty and the value of x in the memory is v_{exp}. Its effect is to write v_{new} directly to the memory). To account for propagating values from the store buffer to variables in memory, we associate with each $i \in \Pi$ a "local dispatcher" component d_i with a single action `prop`, whose impact is to propagate a write command from the buffer queue to the memory. We denote $\mathrm{Ag} \triangleq \Pi \cup \{d_i\}_{i\in\Pi}$ and call its elements *agents*. Reads and writes are always enabled, while fences, `RMW` and `prop` actions can take place only if specific preconditions hold.

We think of an action taking place as being an *event*. It is convenient to consider two types of read events: $\texttt{RfB}(x, v)$ and $\texttt{RfM}(x, v)$. The former is when the value of x is read from the store buffer, and the latter is when the value of x is read from the variable $x \in \mathsf{m}$ in memory. In both instances, the value returned by the read action is $v \in \mathsf{Val}$. Using $\beta|_x$ to denote the restriction of a store buffer β to pairs of the form $\langle x, \_\rangle$ involving writes to $x \in \mathsf{m}$, the preconditions and effects of the event of a single action being performed in a TSO state $\sigma = \langle \mathsf{m}, \mathsf{buf}\rangle$ are defined as follows:

WRITE
$$\frac{e = i : \texttt{W}(x, v) \qquad \mathsf{buf}' = \mathsf{buf}[i \mapsto \mathsf{buf}(i) \cdot \langle x, v\rangle]}{\langle \mathsf{m}, \mathsf{buf}\rangle \xrightarrow{e} \langle \mathsf{m}, \mathsf{buf}'\rangle}$$

READ-FROM-BUFFER
$$\frac{e = i : \texttt{RfB}(x, v) \qquad \mathsf{buf}(i)|_x = \_ \cdot \langle\langle x, v\rangle\rangle}{\langle \mathsf{m}, \mathsf{buf}\rangle \xrightarrow{e} \langle \mathsf{m}, \mathsf{buf}\rangle}$$

READ-FROM-MEMORY
$$\frac{e = i : \texttt{RfM}(x, v) \qquad \mathsf{buf}(i)|_x = \varepsilon \qquad \mathsf{m}(x) = v}{\langle \mathsf{m}, \mathsf{buf}\rangle \xrightarrow{e} \langle \mathsf{m}, \mathsf{buf}\rangle}$$

RMW
$$\frac{e = i : \texttt{RMW}(x, v_{\mathsf{exp}}, v_{\mathsf{new}}) \qquad \mathsf{buf}(i) = \varepsilon \qquad \mathsf{m}(x) = v_{\mathsf{exp}}}{\langle \mathsf{m}, \mathsf{buf}\rangle \xrightarrow{e} \langle \mathsf{m}[x \mapsto v_{\mathsf{new}}], \mathsf{buf}\rangle}$$

FENCE
$$\frac{e = i : \texttt{F} \qquad \mathsf{buf}(i) = \varepsilon}{\langle \mathsf{m}, \mathsf{buf}\rangle \xrightarrow{e} \langle \mathsf{m}, \mathsf{buf}\rangle}$$

PROPAGATE
$$\frac{e = d_i : \texttt{prop}(x, v) \qquad \mathsf{buf}(i) = \langle\langle x, v\rangle\rangle \cdot \beta \qquad \mathsf{m}' = \mathsf{m}[x \mapsto v] \qquad b' = b[i \mapsto \beta]}{\langle \mathsf{m}, \mathsf{buf}\rangle \xrightarrow{e} \langle \mathsf{m}', \mathsf{buf}'\rangle}$$

A TSO action is considered a *memory access* of the variable x if it is either a $\texttt{RMW}[x, \cdot]$, a `prop` action resulting in $\texttt{prop}(x, \cdot)$, or a read action $\texttt{R}[x]$ resulting in $\texttt{RfM}(x, \cdot)$.

Runs and Protocols. A protocol $P = (P_1, \ldots, P_n)$ maps each process' local state to a nonempty set of enabled actions. Local states record the sequence of actions observed so far. A global state is $(\sigma, \ell_1, \ldots, \ell_n)$ with σ being the TSO state and ℓ_i the local state of process i. A *run* is an infinite sequence of global states, starting with empty buffers and initial local states; each step applies a *joint action* of process, dispatcher, and environment actions, with no conflicting memory accesses to the same variable. A run is of P if every process action in it is allowed by P. We make two liveness assumptions: (i) every enabled `prop` eventually occurs, ensuring fences terminate, and (ii) any process with an enabled action eventually moves.

Nodes and Tags. A node $\theta = \langle b, t\rangle$ denotes agent b at time t; $\theta.\alpha$ is b's action at time t (possibly a null action). The k^{th} write $\texttt{W}(x, v)$ by i is taken to have a *tag* $\texttt{W}.tag = \langle i, k\rangle$; the matching `prop`, `RfB`, and `RfM` on that value inherit the same tag.

3 The Occurs-before Relation in TSO

In asynchronous message-passing systems, the only way that a protocol can ensure that an action takes place later than a specific event at another process, is by forcing the action to be delayed until a message chain from the second process has been constructed. This

is Lamport's *happens-before* relation ([10]). We now define an analogous relation for TSO, where scheduling and propagation are asynchronous and processes lack direct knowledge of event timing. Complicating matters is the fact that distinct processes can read different values of a variable x *at the same time*. Our *occurs-before* relation, also builds chains across process timelines,[1] and is defined as follows:

▶ **Definition 1** (TSO occurs before). *Let $t < t'$, let $i, j \in \Pi$ and let $b, c \in \mathrm{Ag}$. We define the binary* occurs before *relation $\overset{ob}{\rightsquigarrow}_r$ between nodes in r as follows:*

1. $\langle b,t\rangle \overset{ob}{\rightsquigarrow}_r \langle b,t'\rangle$ *for every agent b.*
2. $\langle i,t\rangle \overset{ob}{\rightsquigarrow}_r \langle d_i,t'\rangle$ *if* $\langle i,t\rangle.\alpha \in \{\mathtt{W},\mathtt{RfB}\}$, $\langle d_i,t'\rangle.\alpha = \mathtt{prop}$ *and* $\langle i,t\rangle.\alpha.tag = \langle d_i,t'\rangle.\alpha.tag$.
3. $\langle b,t\rangle \overset{ob}{\rightsquigarrow}_r \langle c,t'\rangle$ *if $\langle b,t\rangle.\alpha$ and $\langle c,t'\rangle.\alpha$ are both memory access of the same variable* ***unless***
   a. $\langle b,t\rangle.\alpha$ *and* $\langle c,t'\rangle.\alpha$ *are both* $\mathtt{RfM}$ *actions, or*
   b. $b = d_i$, $c = i$, $\langle d_i,t\rangle.\alpha = \mathtt{prop}$ *and* $\langle i,t'\rangle.\alpha = \mathtt{RfM}$.
4. $\langle d_i,t\rangle \overset{ob}{\rightsquigarrow}_r \langle i,t'\rangle$ *if* $\langle d_i,t\rangle.\alpha = \mathtt{prop}$ *and* $\langle i,t'\rangle.\alpha \in \{\mathtt{F},\mathtt{RMW}\}$.
5. $\theta \overset{ob}{\rightsquigarrow}_r \theta'$ *if* $\theta \overset{ob}{\rightsquigarrow}_r \hat{\theta}$ *and* $\hat{\theta} \overset{ob}{\rightsquigarrow}_r \theta'$ *for some node $\hat{\theta}$.*

What makes $\overset{ob}{\rightsquigarrow}_r$ interesting and useful is the fact that, in a precise sense, it covers all the information that may be available to the processes regarding the ordering of events. We will show that, roughly speaking, if an event e in r is not related by $\overset{ob}{\rightsquigarrow}_r$ to another event e' in r, then there is a run r' that is indistinguishable from r in which e' takes place strictly before e does. We now turn to prove a more general result which will imply this fact.

4 Delaying the Future in TSO and its implications

Our goal is to use the occurs-before relation to prove lower bounds and necessary conditions on protocol structure. First, we show that processes cannot guarantee a specific event order without an occurs-before chain. If a process has the same local state at two execution points where it moves, it will take the same action at both points; events that did not modify its state cannot affect its behavior. This motivates the definitions of $\mathtt{Past}_r(S)$, the set of nodes in the causal past of S and of runs *local equivalence*.

▶ **Definition 2** (The past).[2] *For a set of nodes S in a run r, we define* $\mathtt{Past}_r(S) \triangleq \{\theta : \theta \overset{ob}{\rightsquigarrow}_r \theta' \text{ for some } \theta' \in S\}$ *and* $\mathtt{Past}^+_r(S) \triangleq \mathtt{Past}_r(S) \cup S$.

▶ **Definition 3** (Local Equivalence). *Two runs r and r' are called* locally equivalent, *denoted by $r \approx r'$, if for every process j, a local state ℓ_j of j appears in r iff ℓ_j appears in r'.*

We can now show:

▶ **Theorem 4** (Delaying the future in TSO). *Let r be a TSO run of a protocol P, let S be a set of nodes of r, and let $\Delta \geq 0$. Then there exists a TSO run $r' \approx r$ of the same protocol P such that:*

(a) *Every agent $b \in \mathrm{Ag}$ performs exactly the same actions in the same order in both runs; moreover, an action that b performs at time t in r is performed in r' at time t if $\langle b,t\rangle \in \mathtt{Past}_r(S)$, and it is performed at time $t + \Delta$ otherwise.*

[1] We do not call this relation "happens-before" because the latter has become synonymous with message chains over the last five decades, and the new relation in TSO has a different flavor.
[2] See Figure 1 for an illustration.

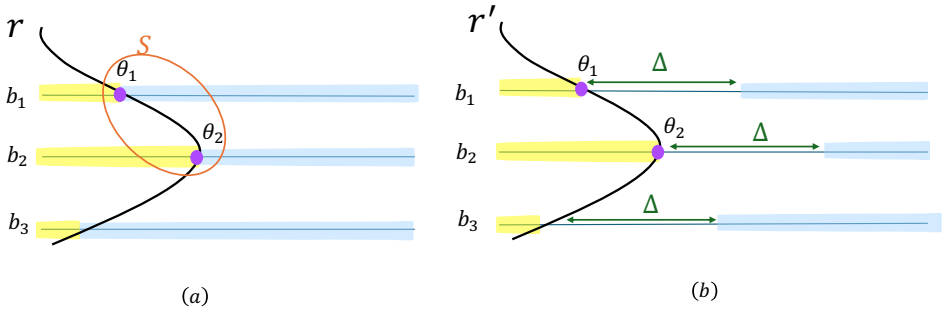

Figure 1 $S = \{\theta_1, \theta_2\}$ and $\texttt{Past}_r(S)$ is composed of all the nodes in the yellow part. $\texttt{Past}_r^+(S)$ contains in addition the nodes on the black curve, delimiting the nodes in past of S from the other nodes. (b) represents the run r' guaranteed to exist by Theorem 4.

(b) *Every process j's local states are shifted accordingly:*

$$r_j(t) = \begin{cases} {r'}_j(t) & \text{for all } \langle j,t\rangle \in \texttt{Past}_r^+(S) \\ {r'}_j(t+\Delta) & \text{otherwise.} \end{cases}$$

Intuitively, Theorem 4 implies that everything outside the $\texttt{Past}_r$ of a node (or set of nodes) can be moved forward in time by an arbitrary amount, while preserving the timing of events in this past. This implies that without an $\overset{ob}{\rightsquigarrow}$ connection, events and operations preceding a given operation can be shifted to no longer precede it, without changing its outcome. If changing the real-time order of two operations can affect the correctness of a given execution, as it typically does when linearizability is required, then implementations must ensure that operations whose order should not be changed are related by $\overset{ob}{\rightsquigarrow}$. In some cases, e.g., when a process runs solo, it is possible to show that establishing $\overset{ob}{\rightsquigarrow}$ requires a process to perform explicit synchronization operations. This is the subject of the next section.

5 Implementing Linearizable Operations in TSO

Roughly speaking, an implementation $\mathcal{I}$ of an object is a protocol satisfying the object's specification. It is said to be *linearizable* if in every execution of $\mathcal{I}$, operations appear to occur instantaneously in a way that is consistent with the original execution and that satisfy the sequential specification of the object. See [8] for a formal definition.

Registers. We consider implementations of a multi-writer multi-reader register, where every process may perform (linearizable) Read and Write operations, in the TSO memory model. For obstruction-free register implementations, techniques used in the proof of Theorem 4 and in [13] can be used to show:

▶ **Corollary 5.** *Let $n \geq 2$ and let $\mathcal{I}$ be an obstruction-free linearizable implementation of a register in TSO for n processes. Then for every $m > 0$ there must be a run r_m of $\mathcal{I}$ in which exactly m* Write *operations are performed, and each of the* Write*s performs a fence* F *or an* RMW*.*

We remark that Castañeda *et al.* prove in [6] that *spec-available* implementations of linearizable registers in TSO must contain a run in which at least one Read or Write operation performs an F or RMW action. Since every obstruction-free implementation is in particular

spec-available, the same follows for obstruction-free implementations. For obstruction-free implementations, Corollary 5 provides sharper bounds. In [15] Theorem 4 (DtF) is applied in the style of Theorem 6 of [14] to show that linearizable register implementations must create $\overset{ob}{\rightsquigarrow}$ relations between Write operations, to ensure linearizability. In addition, DtF is applied to the analysis of linearizable implementations of snapshot objects.

The Need for Synchronization to Establish Occurs Before. In a precise sense, Corollary 5 makes use of the fact that a process performing a linearizable Write must guarantee that the value it is writing is recorded in memory. We can show that the only way for a process to know that the value it wrote can be read by others is either by way of a read that provides it with feedback from other processes or by synchronization.

▶ **Definition 6** (feedback loop). *We say that there is a* feedback loop *between nodes* $\langle i, t_1\rangle$ *and* $\langle i, t_2\rangle$ *in run* r *(and write* $\langle i, t_1\rangle \circlearrowleft_r \langle i, t_2\rangle$*) if there is a node* $\langle b, t\rangle$ *with* $b \notin \{i, d_i\}$ *such that* $\langle i, t_1\rangle \overset{ob}{\rightsquigarrow}_r \langle b, t\rangle \overset{ob}{\rightsquigarrow}_r \langle i, t_2\rangle$*. We say that an operation* X *contains a feedback loop in* r *if* X.$s \circlearrowleft_r$ X.e.

▶ **Theorem 7.** *Let* X *be an operation in* r *containing a write with a tag* $\kappa = \langle i, k\rangle$*. If*

1. X *does not contain a feedback loop, and*
2. i *does not perform a* `F` *or* `RMW` *action during* X.

Then there is a run $r' \approx r$ *in which* κ *is not propagated to memory before* X.e.

This is a powerful result. In TSO, a process shares information only by writing to memory and making the write visible. If completing an operation requires others to observe it, the process must either obtain feedback from another process or use a `F` or `RMW` operation. In obstruction-free protocols, feedback can not be guaranteed, so fences or `RMW` become necessary to ensure visibility before operations can be completed.

6 Discussion

While we focus in this BA on registers, the full version [15] also covers snapshots, and extending the approach to the study of linearizable implementations of other shared objects appears promising. The register results of [6] exploit the fact that Read operations are one-sided non-commutative with respect to Write operations, yielding (roughly) that there exist runs where at least one of two adjacent write–read operations must contain a `F` or `RMW`. Our framework strengthens this: we prove that there exist runs where in fact Write s must contain a `F` or `RMW`. In typical register implementations, Writes complete with an acknowledgment independently of preceding Writes, meaning they are not one-sided non-commutative. Nevertheless, our tools show that there are runs where Writes must contain `F` or `RMW`. In addition, the DtF theorem and its uses, both in the TSO setting and in [14], raise the question of whether similar results hold in other memory models. Finally, we observe that although our $\overset{ob}{\rightsquigarrow}$ relation in TSO is analogous to happens-before in message passing, it differs in that $\overset{ob}{\rightsquigarrow}$ does not necessarily convey information, even under full information, whereas happens-before does. Nevertheless, like happens-before, the $\overset{ob}{\rightsquigarrow}$ relation remains a necessary condition for ordering in many cases. This fundamental difference is both interesting and worth deeper investigation.

References

1 Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995. doi:10.1007/BF01784241.

2 Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. *ACM Trans. Program. Lang. Syst.*, 39(2), May 2017. doi:10.1145/2994593.

3 Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 258–272. Springer, 2010. doi:10.1007/978-3-642-14295-6_25.

4 Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *38th POPL*, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1926385.1926442.

5 Gérard Boudol and Gustavo Petri. Relaxed memory models: an operational approach. In *36th POPL*, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1480881.1480930.

6 Armando Castañeda, Gregory V. Chockler, Brijesh Dongol, and Ori Lahav. What cannot be implemented on weak memory? In *DISC 2024*, LIPIcs, 2024. doi:10.4230/LIPICS.DISC.2024.11.

7 Cormac Flanagan and Stephen N Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 121–133. ACM, 2009. doi:10.1145/1542476.1542490.

8 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, july 1990. doi:10.1145/78969.78972.

9 Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. *SIGPLAN Not.*, 52(6):618–632, June 2017. doi:10.1145/3140587.3062352.

10 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, july 1978. doi:10.1145/359545.359563.

11 Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers c-28*, 9:690–691, 1979. doi:10.1109/TC.1979.1675439.

12 Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

13 Yoram Moses. Relating knowledge and coordinated action: The knowledge of preconditions principle. In R. Ramanujam, editor, *15th TARK*, pages 231–245, 2015. doi:10.4204/EPTCS.215.17.

14 Raïssa Nataf and Yoram Moses. Communication Requirements for Linearizable Registers. In *DISC 2024*, Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2024. doi:10.4230/LIPIcs.DISC.2024.33.

15 Raïssa Nataf and Yoram Moses. Time, fences and the ordering of events in TSO, 2025. arXiv:2508.11415.

16 Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag. doi:10.1007/978-3-642-03359-9_27.

17 Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying fence elimination optimisations. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 146–162, Berlin, Heidelberg, 2011. Springer-Verlag. doi:10.1007/978-3-642-23702-7_14.