TECHNICAL REPORT #2011-04

On Implementation of Read/Write Shared Variables

by

Sibsankar Haldar and K. Vidyasankar

Department of Computer Science
Memorial University of Newfoundland
St. John's, NL, Canada A1B 3X5

October 2011

On Implementation of Read/Write Shared Variables [*]

Sibsankar Haldar[†]
Motorola Mobility, Inc
809 11th Avenue
Sunnyvale, CA 94089
United States of America

K. Vidyasankar
Department of Computer Science
Memorial University of Newfoundland
St. John's, Newfoundland
Canada A1B 3X5

October 2, 2011

Abstract

A shared variable is an abstraction of persistent interprocessor communications. Processors execute read and write operations, often concurrently, on shared variables to exchange information among themselves. The behavior of operation executions is required to be "consistent" for effective interprocessor communications. Consequently, a consistency specification of a shared variable describes some guarantees on the behavior of the operation executions. Shared variables of different consistencies have been defined in the literature, and are categorized in hierarchies.

In this paper, we deal with implementations of higher-level shared variables from lower-level ones. We synthesize a set of axioms based on space-time points (and coincidences), that must be satisfied by the lowest level variables. These axioms help us to analyze and argue about plausible implementations. We then discuss an optimization of the write operation execution to improve performance, and its effect on the consistency guarantees of an implementation. We show that (i) this optimization is possible for sequentially consistent variables but not for linearizable variables, and hence (ii) linearizable variables cannot be wait-freely implemented from sequentially consistent variables alone.

Key Words: Atomicity, axiom, causality, concurrency, consistency, defining condition, illegality, implementation, impossibility proof, linear time reference, linear extension, linearizability, nonatomic operation execution, optimization, partial order, physics, relativity, shared variable, sequential consistency, space-time coincidence, strong consistency, system execution, wait-freedom.

1 Introduction

We consider a computing system that consists of a finite collection of $n > 1$ asynchronous and autonomous sequential processors[1], $P_1, \cdots, P_n$. They communicate among themselves by reading and modifying shared variables. (They have no other means of communications

[*]This research is supported in part by the Natural Sciences and Engineering Research Council of Canada Individual Research Grant OGP0003182.

[†]Associated to Basic Foundation, Vivekananda Nagar Main Road, Madhyamgram, Kolkata 700130, India.

[1]A processor, here, corresponds to a single thread of execution flow in the computing system.

among themselves.) They are driven by their local *control programs.* Each processor fetches one instruction from its control program, and executes the instruction to the completion before it fetches the next instruction.[2] Executions of instructions may involve accessing shared variables.

A *shared variable* is an abstraction of persistent interprocessor communications. Persistence here means that the same information can be retrieved from the variable multiple times (without any upper bound) until the information is changed explicitly. Shared variables are logical units of reference by the processors. Each shared variable has a unique name and a *type.* The type defines a finite domain of values that the variable can assume, the interface operations, and the consistency semantics of the operations.

An instance of an execution of any operation on a shared variable will be referred to as an *operation execution*. By an *execution* of an operation on a shared variable, we mean a sequence of two *events*: (1) the executing processor issues the execution request with appropriate arguments to the variable's interface and then (2) the processor gets a completion report from the interface.[3] The former is called an *invocation* or *request* event, and the latter a *response* or *reply* event. By laws of Physics, the former event precedes (in real time) the latter event at the processor, and the time interval, in any linear time reference, between these two events is unknown but non-zero, positive (however small), and finite.[4] (Operation executions are not instantaneous, that is, not atomic.) The actual execution of the request is carried out by some agent(s) on behalf of the requesting processor. (The agent might be the processor itself in a new supervising role.)

For the purpose of this study, we assume that shared variables support only two interface operations: *read* and *write.* A processor, in a write operation execution (*Write*, for short) of a shared variable, sends a value from the value domain of the variable to the interface, and receives an acknowledgment back from the interface. In a read operation execution (*Read*, for short), it sends a read request to the interface, and receives a value back from the interface. The corresponding values are said to be 'associated' with the respective operation executions. A Write (respectively, Read) is considered to be complete when the processor receives the acknowledgment (respectively, value) from the interface. We call such shared variables *Read/Write* variables; unless stated otherwise, all variables of our investigation are Read/Write variables.

In the present study, we consider a computing system containing only a single shared variable. Every processor executes its permitted operations on the variable sequentially, as directed by its control program. However, operation executions of different processors may overlap, and may affect the behavior of one another. The speeds at which the processors execute their operations are unknown. That is, the time gap between two operation executions of a processor is unknown but non-zero, positive, and finite.[5]

When all operation executions of a shared variable execution are sequential, that is without overlap in real time, analyzing their behavior is straightforward: if all Reads return

[2]We do not address the issues of "out of order" executions of operations by a single processor.

[3]The issuing processor is assumed to be blocked until it gets the completion report from the interface. That is, the execution history of each processor is sequential, starting with a request event and alternating with a report event.

[4]For the sake of convenience, we are assuming finite time interval to indicate that each operation execution terminates.

[5]For the sake of convenience, we are assuming here processors are non terminating, and they keep on executing Read/Write operations.

values from their respective immediately preceding Writes in the sequence, we say they are *legal Reads* and the shared variable execution is also *legal*. Defining what are legal Reads is at the core of shared variable specifications, especially in the presence of concurrent operation executions. Even though operations on a shared variable may be executed concurrently by different processors, the behavior of the Reads must be predictable, i.e., must satisfy some consistency guarantees for effective interprocessor communications. Two factors are predominantly used to determine the level of consistency guarantees a shared variable provides. One is the values associated with the operation executions. We can observe these values, i.e., what values Writes write and what values Reads report. Whether or not the values reported by Reads satisfy our expectation is decided by the other factor: the history of operation executions, that is, the order relation among the operation executions. For a given execution of a shared variable, we cannot change the values associated with the Reads and Writes, but we can define the order relation variously [10]. Differently defined relations can order the same two operation executions differently. For instance, on one extreme, the order relation can be defined empty; the execution is then considered to be legal if each Read returns a(ny) value from the domain of the variable. Of course, a shared variable whose Reads return any values from its domain will be useless for application development and would not satisfy our expectations. On the other extreme, the order relation can be the global time order of all operation executions and the values correspond to those in a serial order of all the operation executions extending their global time order such that each Read returns the value written by the most recent Write preceding the Read in the serial order; the serial order is a *legal linearization* of the given execution. A wide range of consistencies between these two extremes can be associated with executions of shared variables, by defining different order relations and weakening the definition of read legality. This is also observed independently by Higham et al. [13]; they say "the notion of validity is independent of the computation". The computation is what actually happens in reality, but the validity is how we perceive the computation.

Many shared memory consistencies have been reported in the literature. They include linearizability [11, 24], sequential consistency [15], casual consistency [3], processor consistency [8], PRAM [21], cache consistency [8], partial/total store order [13], etc. These specifications, of course, transcend to shared variable specifications as a special case where the shared memory holds only one shared variable. In [10], we particularly study specifications of shared variables and characterize them based on five types of read-illegalities, and present a hierarchy of shared variables. This paper takes [10] one step further: we study plausible implementations of shared variables.

Implementing higher-level shared variables from lower-level ones is of fundamental importance for any hierarchical system building. Conceptually, the 1-writer 1-reader safe/regular bits, defined in [18], are the most fundamental building blocks of system development. (We overlook in this paper the debates about whether such bits can be fabricated in modern hardware.) For the sake of convenience, these bits are also referred to as the *primitive* variables in this paper. They are more like 2-processor unidirectional pipes. Only these bits are provided at the lowest level (called *primitive level*) for interprocessor communications within a single chip. They are used to construct higher-level shared variables, which are used to construct further higher-level variables, and so on.

Note that an execution of an implementation of a higher-level shared variable induces an execution on each individual primitive variable. Given an order relation for operation executions of an execution of any shared variable at any (primitive or higher) level, it is relatively easy to analyze the behavior of the operation executions and determine the consistencies of the execution. It is relatively hard to correlate order relations between two levels of an execution. In this paper, we bring out some fundamental issues relating to the consistency guarantees a shared variable implementation may ensure. We substantiate these issues with ideas from computer hardware and advanced Physics, and provide some philosophical discussions.

We note that an execution of a shared variable implementation may be observed at least at two different levels: the interface and the primitive levels. (The interface is what users of the shared variable see; the primitive level is hidden from them.) The execution observed at the primitive level will be referred as the 'real' execution in this paper, and the one observed at the interface level is called the 'abstract' execution. Thus, an abstract execution is implemented by a real one, and we need to determine the consistencies of the abstract executions to study implementations. We envisage two issues: (i) the consistency guarantees the real executions provide at the primitive level; and (ii) the correlation of the real executions to abstract ones. We discuss these issues in this paper.

In the primitive level, we have the operation executions and their associated values, but how do we define an order relation among them? We take a cue from certain principles of special relativity theory of Physics and postulate that we need time references (for both processors and primitive variables) to define order relation in the primitive level. There is no single global time reference that can be used by all parties involved in a distributed computation. So, we assume the existence of many *private* linear time references in the system. Furthermore, we postulate that, not just the concept of time but the space-time points and coincidences are the most fundamental concept in analyzing implementations at the primitive level. We propose axioms that capture this concept.

One goal of any practical implementation is achieving high system performance, that is, speeding up the executions of Reads and Writes. For this purpose, the operation executions need to be optimized somehow. However, such optimizations may affect the consistency guarantees of the implementation. We discuss one simple optimization and how it affects the consistency guarantees; in particular, we show that this optimization can be used to guarantee sequential consistency but not linearizability. Using this property, we prove that linearizable variables cannot be wait-freely implemented from sequentially consistent variables alone.

The rest of this paper is organized as follows. Section 2 presents basic preliminaries that are required for presenting our work. This section is a review from literature. Section 3 discusses the issues involved in implementations of shared variables. The issues are (i) formulating system executions at the primitive level, (ii) correlating system executions at two levels where orderings between operation executions are defined differently, and (iii) effects of operation execution optimizations. We also present two impossibility results in this section. Section 4 concludes the paper.

2 Preliminaries

An abstract mathematical structure $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ is called a *system execution* if $\mathcal{S}$ is a finite or countably infinite set, and the two relations $\longrightarrow$ and $\dashrightarrow$ on $\mathcal{S}$ satisfy the following Axioms A1–A5 [16, 18]. (Here, A, B, C, D are elements of $\mathcal{S}$.)

(A1) The relation $\longrightarrow$ is an irreflexive partial order.

(A2) If $A \longrightarrow B$, then $A \dashrightarrow B$ and $B \not\dashrightarrow A$.

(A3) If $A \longrightarrow B \dashrightarrow C$ or $A \dashrightarrow B \longrightarrow C$, then $A \dashrightarrow C$.

(A4) If $A \longrightarrow B \dashrightarrow C \longrightarrow D$, then $A \longrightarrow D$.

(A5) For any A, the set of all B such that $A \not\longrightarrow B$ is finite.

Anger [4] later augmented the axiomatic system of Lamport by adding the following two axioms[6] to get models of system executions:

(A6) $A \dashrightarrow A$.

(A7) If $A \dashrightarrow B \longrightarrow C \dashrightarrow D$, then $A \dashrightarrow D$.

Note that all the symbols used in the above seven axioms are uninterpreted, and they can stand for any thing we could imagine. For example, $\langle \mathcal{N}, <, \leq \rangle$, where $\mathcal{N}$ is the set of natural numbers, *is a* system execution as it satisfies all the seven axioms. For another example, a real execution of a shared variable can be modeled as a system execution $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$, where $\mathcal{S}$ constitutes all operation executions on the variable and the partial order $\longrightarrow$ is derived using a defining condition and $\dashrightarrow$ is derived using the above mentioned seven axioms. (The derived relation $\longrightarrow$ is referred to as a *defining relation* in [10]. As mentioned previously, for a given execution of a shared variable, we can define many $\longrightarrow$ relations and hence have many system executions for the same shared variable execution to study its various consistencies.) Whatever be the way, when we transform a shared variable execution into a system execution, it becomes a little easier to analyze the behavior of the operation executions and determining consistencies of the shared variable execution.

The relation $\longrightarrow$ is normally referred to as "precedes" or "happens before" or "(strong) causal connection", and $\dashrightarrow$ as "may affect" or "weak connection". They are also referred to, simply, as precedence relations. For two operation executions O_1 and O_2, if $O_1 \longrightarrow O_2$ we also say the value or effect of O_1 is available or *traceable* to O_2; it does not matter whether O_2 does not trace or ignores O_1. In a system execution, two operation executions are said to be *concurrent* if they are not related by the $\longrightarrow$ relation. Note that the above axioms permit that concurrent operation executions may or may not be related by $\dashrightarrow$. For example, $\langle \{A, B\}, \phi, \{A \dashrightarrow A, B \dashrightarrow B\} \rangle$ is a valid system execution as it satisfies all the seven axioms; A and B are neither strongly nor weakly related to one another. An abstract *system* is defined to be a set of system executions [18].

The above seven axioms take care of the syntax of system executions, that is, how operation executions (in a system execution) are relatively ordered with each other. They however do not correlate operation executions from semantics point of view. The syntax is one factor in consistency, and helps us in setting up a structure on operation executions. The $\longrightarrow$ and $\dashrightarrow$ relations and the seven axioms exhibit cause-effect relationships on operation executions, i.e., the execution flow in system executions. They are truly independent of the behavior exhibited by operation executions. The other factor is the values associated

[6]Axiom A7 is originally proposed by Abraham and Ben-David in [1] (as reported in [6]). Lamport did not assume Axioms A6–A7 to prove his register implementations, but used A7 in [19].

with operation executions to relate Reads with Writes. Thus, for consistency specifications, we need to associate some meaning to the elements in each system execution. Therefore, we need to augment the above mentioned seven axioms by imposing further restrictions on the structure $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$, and/or by including new axioms for operation executions. For example, to strengthen the structure, we can add the global-time axiom: for all A, B in $\mathcal{S}$, if $A \not\longrightarrow B$, then $B \dashrightarrow A$; and by Axiom A2, $\mathcal{S}$ has $A \dashrightarrow B$ or $B \dashrightarrow A$ or both. For Read/Write variables, we partition elements in $\mathcal{S}$ into two disjoint sets, *Read set* and *Write set*. We implicitly assume the existence of such partitions, and we would not mention them in the rest of the paper. That is, we can identify each element in $\mathcal{S}$ as Read or Write (not both). Lamport defines six additional axioms in [18] to assign semantics to operation executions toward specifying 1-writer multireader safe, regular, and atomic variables. For the sake of completeness, we state the six axioms below, even though some are informal. For formal discussions, readers are advised to read his paper [18]. Again, note that he states these axioms for a single 1-writer shared variable.

(B1) All Writes are totally ordered by $\longrightarrow$, and the very first Write causally precedes all other operation executions.

(B2) For any Read R and Write W to the same variable, either $R \dashrightarrow W$, or $W \dashrightarrow R$, or both.

(B3) A Read always returns a value from the domain of the variable.

(B4) (safe) A Read not concurrent with any Write returns the most recently written value.

(B5) (regular) A Read returns the most recently written value, or the value of one of the concurrent Writes.

(B6) (atomic) Each Read is effectively not concurrent with any Write and returns the most recently written value preceding it.

These axioms are not sufficient for multiwriter shared variables. We need to augment or rewrite these six axioms to accommodate multiwriter variables in his framework. In [10], we presented one possible rewrite. There, to characterize multiwriter multireader shared variables, we identify five orthogonal criteria, all in terms of the Writes that the Reads read values from. They are called read-illegalities. For completeness we present a review of [10] here.

We note that, to have a legal serial extension of the $\longrightarrow$ relation, a given system execution $\langle \mathcal{S}, \longrightarrow, \dashrightarrow, \pi \rangle$[7] cannot have the following situation for any Read $R(W_1)$:

$$O(W_1) \longrightarrow O'(W_2) \longrightarrow R(W_1),$$

where $W_1 \neq W_2$ are Writes, and each of O and O' may indicate either a Read or a Write.[8] If there is such a situation, we say $R(W_1)$ is *illegal with respect to* $\longrightarrow$. We say $R(W_1)$ is:

- *ww-illegal* if O indicates a Write and O' a Write,

[7] We assume the existence of a 'reading mapping' function π from the Read set to the Write set associating each Read R to the Write $\pi(R)$ whose value R returns; for convenience and succinctness, we denote this by notation $R(W)$, where W is $\pi(R)$. Note that we must have $\pi(R) \dashrightarrow R$. In the sequel, we sometimes denote system executions by $\langle \mathcal{S}, \longrightarrow, \dashrightarrow, \pi \rangle$, whose consistency semantics we want to determine.

[8] In this notation, $W(w)$ is the same as w, when w is a Write.

- *wr-illegal* if O indicates a Write and O' a Read,
- *rw-illegal* if O indicates a Read and O' a Write, and
- *rr-illegal* if O indicates a Read and O' a Read.

These four illegalities involving only the $\longrightarrow$ relation form four criteria. The fifth one is the *new-old inversion* or *new-old-illegality* that involves two different Writes and two different Reads:

- *no-illegal*[9]: for two different Writes W_1 and W_2 and two different Reads R and R', we have $W(W_1) \longrightarrow W(W_2)$ and $R(W_2) \longrightarrow R'(W_1)$.

We [10] classify all system executions $\langle \mathcal{S}, \longrightarrow, \dashrightarrow, \pi \rangle$ into different categories $\mathcal{C}$ satisfying different read constraints. Let ww denote that ww-illegality is allowed, and $\overline{ww}$ denote that ww-illegality is not allowed; we use similar notations for the other illegalities. The class $\mathcal{C}(ww, wr, rw, rr, no)$ will consist of those system executions where any read-illegalities could occur. The class $\mathcal{C}(\overline{ww}, \overline{wr}, \overline{rw}, \overline{rr}, \overline{no})$ will consist of those system executions which have none of these read-illegalities. These are called *causal* executions in [10].

Given a system execution $\langle \mathcal{S}, \longrightarrow, \dashrightarrow, \pi \rangle$, we [10] define three relations on the Writes, induced by Reads as follows. Here, $W_1 \neq W_2$ are Writes and $R \neq R'$ are Reads.

- $W(W_1) \longrightarrow_{rr} W(W_2)$ if $R(W_1) \longrightarrow R'(W_2)$,
- $W(W_1) \longrightarrow_{rw} W(W_2)$ if $R(W_1) \longrightarrow W(W_2)$, and
- $W(W_1) \longrightarrow_{wr} W(W_2)$ if $W(W_1) \longrightarrow R(W_2)$.

We [10] define an augmented relation $\longrightarrow_e$ on $\mathcal{S}$ as $(\longrightarrow \cup \longrightarrow_{rr} \cup \longrightarrow_{rw} \cup \longrightarrow_{wr})^*$, and call it the *exclusion closure* of $\longrightarrow$. A system execution $\langle \mathcal{S}, \longrightarrow, \dashrightarrow, \pi \rangle$ is called an *atomic* execution if $\longrightarrow_e$ is a partial order and there are no read-illegalities with respect to the augmented relation $\longrightarrow_e$. That is, $\langle \mathcal{S}, \longrightarrow, \dashrightarrow, \pi \rangle$ is an atomic execution if $\langle \mathcal{S}, \longrightarrow_e, \dashrightarrow_e \rangle$ is a causal execution.[10] (One may note that 'atomicity' is a generic term here and atomicity of an execution depends on how we define the order relations. Atomic executions have stronger consistencies than non atomic ones for the same order relation.) If the execution satisfies a defining condition C, it is called C-atomic. Linearizability and sequentially consistency are both atomic, but their defining conditions are different. Both are, in theory, strong consistencies.

In summary, for a given real execution of a shared variable, we first define a syntax, aka, structure $\langle \mathcal{S}, \longrightarrow, \dashrightarrow, \pi \rangle$ out of the given execution by using a defining condition and Axioms A1–A7, and then use further axioms to assign semantics to the structure and classify the given execution of the shared variable. The foremost challenge is deriving system executions from real executions of implementations (of shared variables), which is the topic of the next section.

[9]Read as en-o illegal.

[10]Here $\dashrightarrow_e$ is equal to $\longrightarrow_e \cup \dashrightarrow$.

3 System implementation

As mentioned previously, shared variables can be organized into a hierarchy based on their consistencies. Higher-level shared variables can be implemented from lower-level ones. Any axiomatic system that helps us to specifying consistencies of shared variables must also have mechanisms to help us in determining the types of these implemented shared variables. Many consistency specifications in the literature can be stated in terms of the absence of various read-illegalities in system executions (see [10]). Thus, the foremost task is to derive higher-level system executions from lower-level ones. Only then we can use an axiomatic system to determine the type of the implemented shared variable by analyzing what read-illegalities are present or absent in the derived higher-level system executions.

Practical systems are implemented in multiple levels. Thus, we may need to analyze executions of an implementation at various levels of abstractions. As stated earlier, at the most fundamental level, a computing system is nothing but an interconnection of 1-writer 1-reader bits (primitive variables). Each primitive variable is written by a preassigned agent, and read by a preassigned different agent. An execution of an implementation (of a higher-level variable) induces, at the primitive level, one execution of each primitive variable. Each primitive variable by itself is a system and has its own interfacing read and write operations. The induced execution of each primitive variable contains all Reads and Writes on the variable. And, these induced executions (of primitive variables) are independent of one another, because they are not aware of one another. Thereby, each induced execution can be analyzed independently. It satisfies its own consistency semantics, whatever it might be.

In the sequel, by a *real execution* of an implementation we mean the collection of the induced executions, one for each primitive variable observed at the primitive level. To determine the higher-level consistency ensured by the implementation, we have two tasks at hand for each real execution of the implementation: (i) from the given real execution we need to form abstract individual primitive system executions and (ii) from these abstract primitive system executions, we must be able to derive abstract higher-level system executions along with the two precedence relations and the reading-mapping function. Only then we can use an axiomatic system to determine the strength of the implementation, that is, the type of the constructed higher-level shared variable. Several interesting issues arise in the derivation process. We discuss them in the rest of this section. Section 3.1 is a review from the literature.

3.1 Implementation relation

Let I be an implementation of a higher-level shared variable V that employs a set v of primitive variables, and let E be a real execution of I. Let $\mathcal{S}$ be the set consisting of all higher-level operation executions (Reads and Writes, observed at the interface of V) in E. Let $\mathcal{P}$ be the set of all primitive operation executions observed at the primitive level for E. Given two system executions $\langle \mathcal{P}, \longrightarrow_{\mathcal{P}}, \dashrightarrow_{\mathcal{P}} \rangle$ and $\langle \mathcal{S}, \longrightarrow_{\mathcal{S}}, \dashrightarrow_{\mathcal{S}} \rangle$, when can we say that the former one implements the latter? We give the definition of implementation by Lamport [18] below in terms of higher-level view and induced precedence relation.

A set $\mathcal{S}$ is a *higher-level view* [18] of a set $\mathcal{P}$ if (1) each element of $\mathcal{S}$ is a finite, non empty set of elements of $\mathcal{P}$, and (2) each element of $\mathcal{P}$ belongs to a finite, non-zero number of elements of $\mathcal{S}$.

Let O and O' be two operation executions in $\mathcal{S}$. As mentioned previously, there are many ways one can define the two precedence relations between O and O'. Lamport [18] defines two induced relations $\overset{*}{\longrightarrow}_{\mathcal{P}}$ and $\overset{*}{\dashrightarrow}_{\mathcal{P}}$ between O and O' as follows:

$$O \overset{*}{\longrightarrow}_{\mathcal{P}} O' \equiv \forall o \in O, \forall o' \in O' : o \longrightarrow_{\mathcal{P}} o', \text{ and}$$
$$O \overset{*}{\dashrightarrow}_{\mathcal{P}} O' \equiv \exists o \in O : \exists o' \in O' : o \dashrightarrow_{\mathcal{P}} o' \text{ or } o = o'.$$

Then, $\langle \mathcal{S}, \overset{*}{\longrightarrow}_{\mathcal{P}}, \overset{*}{\dashrightarrow}_{\mathcal{P}} \rangle$ represents a higher-level system execution of I, induced by $\mathcal{P}$.

Lamport [18] says a (primitive) system execution $\langle \mathcal{P}, \longrightarrow_{\mathcal{P}}, \dashrightarrow_{\mathcal{P}} \rangle$ *implements* a (higher-level) system execution $\langle \mathcal{S}, \longrightarrow_{\mathcal{S}}, \dashrightarrow_{\mathcal{S}} \rangle$ if (1) $\mathcal{S}$ is a higher-level view of $\mathcal{P}$ and (2) $\overset{*}{\longrightarrow}_{\mathcal{P}} \subseteq \longrightarrow_{\mathcal{S}}$. (We say an *implementation relationship* exists between the two system executions.) We note that the closer $\overset{*}{\longrightarrow}_{\mathcal{P}}$ comes to $\longrightarrow_{\mathcal{S}}$ (that is, the fewer elements $(\longrightarrow_{\mathcal{S}} - \overset{*}{\longrightarrow}_{\mathcal{P}})$ has), the better is the relationship between $\mathcal{P}$ and $\mathcal{S}$ and the implementation comes closer to our expectation. When $\longrightarrow_{\mathcal{S}}$ is equal to $\overset{*}{\longrightarrow}_{\mathcal{P}}$, it is a perfect or exact implementation; otherwise, it is a restrictive or inexact implementation. When $\overset{*}{\longrightarrow}_{\mathcal{P}} \subset \longrightarrow_{\mathcal{S}}$, Lamport [18] says $\langle \mathcal{S}, \longrightarrow_{\mathcal{S}}, \dashrightarrow_{\mathcal{S}} \rangle$ is a *pretend* system execution relative to $\langle \mathcal{P}, \longrightarrow_{\mathcal{P}}, \dashrightarrow_{\mathcal{P}} \rangle$.

In reality, as said before, a system may be constructed in many levels, instead of just two levels. We can use the above definition of implementation recursively for any two consecutive levels and see if the lowest-level system implements the highest-level one.

3.2 Primitive system executions

The above definition of implementation (from [18]) helps us in defining a higher-level system execution from a lower-level one via induced relations and pretension. For this, we need to begin with a known system execution at the primitive level of system building, and gradually derive higher-level system executions until we reach the topmost level. The most intriguing question we face in reality is this: we definitely can observe what primitive operation executions and their associated values are, but how do we define the two relations $\longrightarrow$ and $\dashrightarrow$ on primitive operation executions? What defining conditions do the precedence relations satisfy at the primitive level, for a single primitive variable? Can we define the precedence relations arbitrarily? The above mentioned Axioms A1–A7 do help us in deriving primitive system executions, but they are not really sufficient. We ought to assume some ordering on primitive operation executions, and the assumptions must reflect our expectation. The most obvious and natural choice before us seems to be the physical time for this purpose. For asynchronous computing systems, it is tacitly assumed that there is no concept of global time reference(s) in the system. Then, what should be a basis for ordering primitive operation executions? We try to answer it by reviewing the literature.

Lamport in [18] postulated Axioms B1–B6 (see Section 2) to define 1-writer safe, regular, and atomic variables. We are particularly concerned with the Axiom B2 that correlates operation executions of any two processors (one reader and the other writer): For any Read R and Write W to the same variable (at any level in shared variable hierarchy), $R \dashrightarrow W$, or $W \dashrightarrow R$, or both. What is the rationale behind this axiom? As mentioned previously, this axiom holds for any system execution that satisfies the global-time axiom, but may not hold in general. (Axiom B2 is not equivalent to the global-time axiom. For example, it does not correlate operation executions of two reader processors.) It also means that Lamport assumed the existence of some sort of weak interactions between any Read and any Write of every shared variable (at any level of system building).

Lamport justifies Axiom B2 in [17] using four dimensional space-time model from Physics. In the rest of this sub-subsection, we take a cue from Physics and claim that the global-time axiom is needed at the primitive level to derive system executions from real executions of a primitive variable, and there does not seem to exist any other alternatives before us. That is, Axiom B2 must hold at the primitive level.

We believe that to define the two precedence relations on primitive operation executions we need external time reference frames for different observers of the real execution. Many consistency specifications in the literature we know of are defined in terms of execution histories. An execution history is a (finite or countably infinite) sequence of invocation and response events. From the linear execution history, partial orders are defined on operation executions, and analyzed based on the axioms of some axiomatic system. There are exceptions though. For example, Higham et al. [13] assume a computation to be a set of sequences of operation executions, one sequence for each processor. They do not however assume the same (i.e., sequences of operation executions) for each shared variable. Our concern here is the assumption of linear execution histories, and not the axiomatic systems being used for the analysis. (We can use any axiomatic system of our preference only after setting up order relations among operation executions and defining their reading-mapping function.) Although a global time reference is not mentioned explicitly, every sequence connotes a discrete global time reference, in which events are "moments of time". This is perhaps first observed by Lamport in [14]. He mentions that the concept of time is vital to our sense of thinking, understanding, and reasoning. At the fundamental depth (deep in the shared variable implementation), we may need to reason about interactions of primitive operation executions based only on time. We strongly believe that without the concept of some time references at the primitive level, it is not possible to derive the two precedence relations on primitive operation executions.[11] However, there is no single global time reference that can be used by all parties involved in a distributed computation. So, we assume the existence of many *private*, and not global, linear time references in the systems. Each time reference continuously increases with the physical time (as defined by physicists, based on the constancy of the velocity of light). For our purpose, a time reference never runs backward and it increases monotonically; time references, however, can vary their rates of increase arbitrarily and need not be running synchronously in lock-steps. The time references (actually, space-time points) are used only to derive causality relationship among operation executions.

We assume that every processor has its own private time reference which is not accessible to other processors. For any (higher-level) operation executions O and O' of processor P_i, we say that O precedes O' in the *processor order*[12] denoted $O \longrightarrow_i O'$, if O completes before O' starts in the private time reference of P_i. It is a total order. All local observers residing at P_i will see this order. Here is our first axiom.

(T1) Each processor P_i has its own private time reference T_i which is not used by others.

[11] As we will see shortly time references are not sufficient, we need space-time points and coincidences as defined by Physicists.

[12] Program order and processor order are the same in our study as the processors do not execute program instructions out of order.

> Each operation execution of P_i spans a non-zero finite interval in T_i and the intervals do not overlap with each other in the time reference. The time reference is used only to determine individual processor order $\longrightarrow_i$. From this we can derive $\dashrightarrow_i$ using Axioms A1–A7, and define $\langle \mathcal{S}_i, \longrightarrow_i, \dashrightarrow_i \rangle$ as a system execution, where $\mathcal{S}_i$ is the set of all (higher-level) operation executions of processor P_i.[13] That is, for every processor P_i, $\langle \mathcal{S}_i, \longrightarrow_i, \dashrightarrow_i \rangle$ is a system execution, and it does satisfy the global-time axiom.

We also assume the existence of a separate (independent) time reference T_x for each primitive shared variable x, for the following philosophical reason. Every operation execution on a primitive variable x will span a non-zero finite time interval in T_x; for the sake of convenience, a single space-time point is considered a non-zero interval. Two operation executions o and o' on x are said to overlap iff they have intersecting time intervals (or time moments) in the time reference T_x.[14] This is called the principle of the invariance of "coincidence of time and space" in Physics: if one (inertial) observer sees two events occur at the same space and at the same time, then all (inertial) observers in our universe will see the same coincidence of the two events but they may see at different times (of their local time references).[15] That is, all observers in the universe will agree that there is a space-time coincidence on shared variable x, but they may disagree on the time and they may see it at different (their own) local times. (Said differently, although there is no agreement about time among observers, there is definitely an agreement on a coincidence. If there is no coincidence of two operation executions of x, then the operation executions are not concurrent.) Such coincidence events can have an immediate (and direct) impact on each other, and hence we assume that they do affect one another. Said differently, two concurrent operation executions can affect one another if and only if they have some common space-time points, aka, coincidences. Thereby, if the two primitive operation executions happen in two different spaces at the same time (whoever's time that is), they cannot directly affect one another. This is another law of nature. If they occur at the same space but at different times, then also there are no space-time coincidences but the latter one can sense or trace the effect of the former one because the space (aka, primitive variable) is assumed to be a persistent interprocessor communication medium (unless some later space-time coincidences on the same space have annulled the effect). (Note that this statement is not valid for non-persistent, aka, transient communications. Traceability of information is what makes persistent variables useful in computer programs.) A primitive shared variable represents a (persistent) space component here. Thus, the other component of space-time, namely, time reference is vital to analyze executions of the primitive variable. The time reference is independent of those of processors, and is solely used to determine different

[13]For this case, to obtain $\dashrightarrow_i$ from $\longrightarrow_i$, first apply Axioms A6 and A2 and then Axioms A3 and A7. As $\longrightarrow_i$ is linear, $\dashrightarrow_i$ is in fact redundant.

[14]Although they often say an operation execution O_1 of processor P_1 and an O_2 of P_2 overlap, they actually mean O_1 and O_2 overlap on some common variable x, but they never say whose time is used for space-time coincidences. The time is actually of x, and not of P_1 nor of P_2. For example, when we say two processors are in the same critical section simultaneously, we mean simultaneously by the time reference of the critical section and not those of the processors. As the space-time events produced by the processors coincides, every observer in the universe will be able to see this coincidence, and say the processors are in the critical section simultaneously. Without the assumption of the time reference of the critical section, how would we formulate the mutual exclusion property?

[15]Actually, in Physics, different observers may see the same coincidence in different space points. The variables (i.e., objects in our study) are relatively stationary with respect to one another, that is, they do not change their positions in the space with respect to time. Thus, different observers will see the same coincidence in the same variable, but may see it at different (their own local) times.

orderings among operation executions of the primitive shared variable. We represent this fact by *weak object order* $o \dashrightarrow_x o'$ and $o' \dashrightarrow_x o$, where o and o' have common space-time points, aka, coincidences; that is, they may affect the behavior of each other irrespective of the kind of operation executions o and o' are. Similarly, we define $o \longrightarrow_x o'$, called *strong object order*, if all space-time points of o precede those of o' (i.e., o is completed before o' starts and there are no common space-time coincidences) in the time reference T_x. Now, we can complete the content of the two precedence relations $\longrightarrow_x$ and $\dashrightarrow_x$ with the help of the aforementioned Axioms A1–A7 to encompass all primitive operation executions of x into a single "abstract system execution" of x. All local observers residing at x will see the same relations $\longrightarrow_x$ and $\dashrightarrow_x$. Then, $\langle \mathcal{P}_x, \longrightarrow_x, \dashrightarrow_x \rangle$ becomes a system execution, where $\mathcal{P}_x$ is the set of all operation executions on x, that does satisfy the global-time axiom. We now state the axioms for primitive shared variables.

(T2) Each primitive shared variable x has its own private time reference T_x which is not used by others. Each operation execution on x spans a non-zero finite interval in T_x. The time reference is used solely to determine the object orders $\longrightarrow_x$ and $\dashrightarrow_x$, and primitive system executions $\langle \mathcal{P}_x, \longrightarrow_x, \dashrightarrow_x \rangle$.

(T3) Operation executions of each primitive variable x comply with the time reference T_x. Let o and o' (whatever be their sources) be two operation executions on x. Then, $o \dashrightarrow_x o'$, or $o' \dashrightarrow_x o$, or both. That is, primitive system executions satisfy the global-time axiom.

Axiom T3 implies that Lamport's Axiom B2 holds for all primitive variables solely due to the Physics principle of space-time coincidence and the definition of $\dashrightarrow$; we believe there are no other fundamental reasons behind it.[16] Primitive-object order relation is fundamental in our study and we now know how to construct individual system executions of a primitive variable from its real executions.

A system with only one primitive variable is uninteresting. As mentioned previously, in practical systems, a higher-level variable is implemented from many primitive variables. Things become more complicated to handle when we have many independent time references in the same system. Each primitive variable has its own space-time characteristics, i.e., induced object order. We would need to integrate individual object orders (of all primitive variables) for the integrity of the entire primitive execution (similar to logical clocks defined in [14, 22, 23]) by setting up order relationships between operation executions of different primitive variables. Note that, for an execution of an implementation, we do know which primitive operation executions are parts of which higher-level operation executions. Additional ordering on primitive operation executions on different primitive variables is determined based on the order of their executions by the higher-level operation executions. For example, if a processor P_i first executes o on x and then o' on y, then $o \longrightarrow_i o'$. We can now define $\langle \mathcal{P}, \longrightarrow_{\mathcal{P}}, \dashrightarrow_{\mathcal{P}} \rangle$ using $\longrightarrow_x$, $\dashrightarrow_x$, and $\longrightarrow_i$, where $\mathcal{P}$ contains all primitive operation executions and satisfies Axioms A1–A7.

The above defined $\langle \mathcal{P}, \longrightarrow_{\mathcal{P}}, \dashrightarrow_{\mathcal{P}} \rangle$ is good enough for us. If we assume the global-time axiom for the integrated system, we can have more structures in the primitive system

[16]Axiom T3 may appear to be weaker than Axiom B2 because here, in theory, both o and o' can be Reads, but it is not because, all primitive variables being 1-reader, two Reads can never be concurrent on the same primitive variable. Thus, T3 and B2 are the same at the primitive level. That is, though nowhere stated explicitly, B2 in disguise is *the* global-time axiom at the primitive level!

execution. The following three axioms are assumed to correlate the private time references of all processors and all primitive variables.

(T4) There is one and only one global linear time reference T_g accessible to all global observers. It is not accessible to any processors or shared variables.

(T5) As per Physics, nothing in our universe moves faster than light.[17] So, no (inertial) observer will see an operation finishing its execution before it is even initiated for execution.[18] More precisely, two events occurring in some linear order in any private time reference will be seen by all global observers in the same linear order though the time gap between the events may vary from one observer to another. This is also known as the principle of causality in Physics. Thereby, a global observer should be able to see the same causal orderings of operation executions (and space-time coincidences) of every private time reference. Therefore, we assume that each private time reference has an order preserving embedding (similarity mapping) in the global time reference T_g. These are bi-continuous functions from private time references to the global time reference.[19]

(T6) The similarity mappings in T5 must be mutually consistent with reality and our expectation. Let SM_i be similarity mappings for processor P_i, and SM_x be for primitive variable x. Each operation execution O_i of P_i, by Axiom T1, spans a time interval in T_i. If we apply the SM_i on this time interval, we get a time interval in T_g. Without loss of generality, we use the same notation SM_i to represent this mapping of O_i to the time interval in T_g. Thereby, $SM_i(O_i)$ is a set of points (time moments) in T_g. Similarly, for operation execution o_x on x, $SM_x(o_x)$ is a set of points in T_g.

1. Suppose O_i is a higher-level operation execution of P_i, and $o_x \in O_i$. Then, it must be the case $SM_x(o_x) \subseteq SM_i(O_i)$, that is, all mapped points of o_x are also mapped points of O_i.
2. For two primitive operation executions o_x and o_y on two different primitive variables x and y, respectively, if processor P_i executes both of them, first o_x to completion and next o_y (that is, $o_x \longrightarrow_i o_y$), then $SM_x(o_x) \prec SM_y(o_y)$, that is, all mapped points of o_x precede all mapped points of o_y. (Note that no local observer will be able to see or derive this relation across two primitive shared variables.)

These time axioms are very intuitive. For an execution of an implementation, by the above time axioms, we can define $\longrightarrow_{\mathcal{P}}$ and $\dashrightarrow_{\mathcal{P}}$ on primitive operation executions across all primitive variables as follows. They definitely include the object orders $\longrightarrow_x$ and $\dashrightarrow_x$, respectively, and may have some more orderings among operation executions, as defined below. For any two operation executions o_x and o_y, on two different primitive variables x and

[17]Even if this turns out not to be the case, we would not debate about the fastest element in the universe in this paper.

[18]This is akin to the assumption that a message can only be received after it has been sent [14].

[19]Similarity mapping is also called homeomorphism by mathematicians. Bi-continuous functions are those continuous functions whose inverses are also continuous. For our case, the causal ordering of two events in one private time reference will preserve the same ordering in the global time reference by the similarity mapping. And hence, whatever causal orderings local observers can see will also be seen by all global observers.

y, respectively, we define $o_x \longrightarrow_{\mathcal{P}} o_y$ if $SM_x(o_x) \prec SM_y(o_y)$, and $o_x \dashrightarrow_{\mathcal{P}} o_y$ if $o_x \longrightarrow_{\mathcal{P}} o_y$ or $SM_x(o_x)$ intersects $SM_y(o_y)$. We now use Axioms A1–A7 to make $\langle \mathcal{P}, \longrightarrow_{\mathcal{P}}, \dashrightarrow_{\mathcal{P}} \rangle$ a system execution, where $\mathcal{P}$ is the set of all primitive operation executions (of all primitive variables). (It is interesting to note that $\longrightarrow_{\mathcal{P}}$ on the set of primitive operation executions may not be equal to the global-time relation $\longrightarrow_{gt}$ on the same set of operation executions.)

We note that different sets of mutually consistent similarity mappings (of Axiom T5) could define different primitive system executions for the same real execution of the implementation. But, they all preserve individual object orders $\longrightarrow_x$ and $\dashrightarrow_x$ and processor orders $\longrightarrow_i$. These derived precedence relations $\longrightarrow_{\mathcal{P}}$ differ from each other only in what is called the "relativity of simultaneity", again from Physics: if one observer A says that two events in two different spaces happen at the same time (by A's own time reference), another observer B may disagree based on B's own time reference. From these primitive system executions, we may derive higher-level system executions using the concept of higher-level view and induced precedence relations of Lamport [18]. We say an implementation implements a higher-level system, if for each real execution of the implementation, there exists a (at least one) set of mutually consistent similarity mappings such that the derived primitive system execution from the similarity mappings implements a system execution of the higher-level system. Otherwise, the implementation is not correct.

All the previous time axioms are oriented toward ordering of operation executions for defining primitive system executions. That is, they are used to define the syntax of primitive operation executions. Unfortunately, as mentioned previously, ordering is not everything in studying system executions. It is one factor for specifying consistency. The other factor is the values associated with operation executions. Given a real execution, we know what values are associated with operation executions of both the higher-level variable and lower-level primitive variables. We must have some minimum consistency guarantee at the primitive level to make primitive variables worthwhile. Without this minimum consistency, primitive variables are not useful to us. Note that though the primitive-object orders are induced by higher-level operation executions, the semantics of primitive operations are ensured by the primitive variables themselves. The primitive variables must provide a consistency guarantee that does not violate our intuition. We expect that they would provide a bare minimum guarantee to retain the latest value in the absence of new Writes (aka, persistent interprocessor communications at the primitive level), for otherwise the variables cannot be used for effective system building. Each Read always gets the most recently written value or one of overlapping Writes.[20] For the purpose of interprocessor communications, each primitive variable must have at least two states, and hence all primitive variables are assumed to be boolean. As mentioned previously that primitive system executions cannot satisfy any defining condition other than T3 (or B2), so we postulate that all primitive shared variables are of the *same* type, i.e., multiple different classes of primitive variables do not exist. We now state the axiom of semantics of primitive variables.

(T7) Executions of each primitive variable x are in category $\mathcal{C}(\longrightarrow_x)(\overline{ww})$. That is, primitive executions are regular (Cf. Axioms B1–B5) relative to their respective time references.

That is, each primitive shared variable is 1-writer 1-reader regular, and it represents a

[20]A primitive variable cannot toss a coin nor play a dice to return values for Reads. Moreover, it cannot remember its past history, that is, the previously written values. In addition, Reads do not alter the information content of the variable.

persistent medium (for two boolean values, namely true and false or 1 and 0) between a writer and a reader.

At the primitive level, we can have only 1-writer 1-reader boolean regular variables that satisfy only the above mentioned time axioms (aka, the defining conditions at the primitive level). At the higher levels, we can implement multiwriter multireader multivalued variables that satisfy different defining conditions. We discuss some issues involved in those implementations in the next subsection.

3.3 Optimization issues

One goal of any practical implementation of higher-level shared variable is achieving high performance for higher-level operation executions. We here measure the performance by the amount of time processors wait or spend for higher-level operation executions to complete. The lesser the waiting time, the better is the performance. Thereby, to improve the performance of higher-level operation executions, we need to cut short the waiting time of processors (between the invocation event and the corresponding response event), and there are no other alternatives. That is, if possible, an operation invocation may be delivered its response by the shared variable interface even when all its primitive operation executions are not complete; the remaining part of the higher-level operation execution will be carried out asynchronously by some agents on behalf of the invoking processor.

The example of an execution of a shared variable, shown in Figure 1 in the global-time setup, will be used to study the impact of optimizations on consistencies. Every higher-level operation execution is associated with two lines in the figure. The lower (solid) lines represent the executions at the primitive level. If we consider an induced higher-level view of this execution, we have only $W_1 \xrightarrow{*} R$; we do not have $W_2 \xrightarrow{*} R$; the induced system execution does not violate any read-illegality with respect to this $\xrightarrow{*}$ relation. Then, (by Proposition 3.3 of [10]) the execution in the figure is atomic with respect to this defining relation $\xrightarrow{*}$. Here, as each higher-level operation execution encompasses all its primitive operation executions, we can have, by the time axioms, a set of mutually consistent similarity mappings where we have $\longrightarrow_{gt}$ equal to $\xrightarrow{*}$ for this induced system execution. So, the execution is linearizable.

Now, the upper (dashed) lines in the Figure 1 show the case where higher-level Writes are optimized: Writes W_1 and W_2 make early returns without completing all primitive operation executions. We now have $W_1 \longrightarrow_{gt} W_2 \longrightarrow_{gt} R$ at the interface level and the system execution is not linearizable because it violates ww-legality with respect to $\longrightarrow_{gt}$. We specifically note that higher-level operation executions do not encompass all of their primitive operation executions any longer, and this causes violations of linearizability. We also note that, due to optimizations, Axiom T6.(1) may not be valid any more. The higher-level system execution (based on some defining conditions) may have more order relationships than the induced $\xrightarrow{*}$ relation and thereby, violate some consistency conditions. It is interesting to note that both the (optimized and non-optimized) executions in the figure are sequentially consistent.

For the optimized part of Figure 1, can we say that higher-level operation executions are higher-level views of primitive operation executions? (1) Suppose no. Then, higher-level operation executions collectively may not encompass all of primitive operation executions, and, we cannot use Lamport's definition of system implementation any more. (2) Suppose yes. Then, the implementation becomes more and more pretending (i.e., inexact). In addition, for two causally related higher-level operation executions, $O_1 \longrightarrow_{gt} O_2$, not all lower-level operation executions of O_1 precede those of O_2. In fact, if they access the same primitive variable x, the primitive operation executions on x from O_2 may precede those on x from O_1. Thus, as mentioned previously, Axiom T6.(1) may not be valid any more. Thus, in either alternative, we have a dilemma about the definition of system implementation.

Some of the questions that arise at this point are the following. How do we relate the primitive system execution to the higher-level system execution when the Writes are optimized? How do we define the implementation relationships between the primitive and higher levels? What is the relationship between the defining conditions in both levels? How do we implement an optimized higher-level system execution? We do not know answers to these questions as of now.

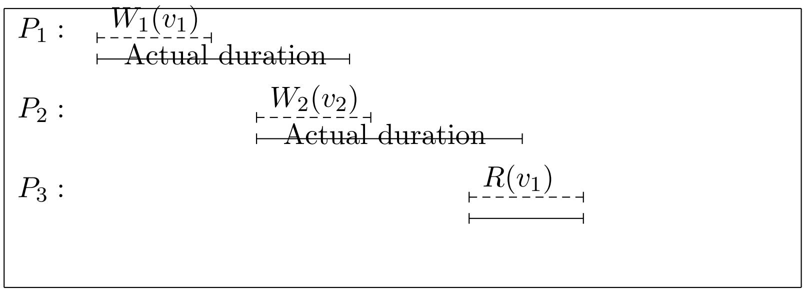

Figure 1: A typical high performance execution of a shared variable V.

3.4 Shared variable implementation

As mentioned previously, each processor executes its operations sequentially in the order specified by its control program. It does not start executing the next operation until it receives a response to the previous request. It is shown by various researchers that linearizable Read/Write variables can be constructed from the most fundamental 1-writer 1-reader safe bits in a wait-free manner [2, 9, 18, 20, 26]. Such constructions implement a READ function and a WRITE procedure for read and write operations, respectively. (These routines, in fact, constitute shared variable interface.) Li, Tromp, and Vitányi [20] present constructions of multiwriter multireader linearizable variables from 1-writer 1-reader linearizable variables and also from 1-writer multireader linearizable variables. They also present constructions of 1-writer multireader linearizable variables from 1-writer 1-reader linearizable variables. Abraham [2] presents a construction of multiwriter multireader linearizable variables from 1-writer multireader linearizable variables. Lamport [18] shows how to construct 1-writer 1-reader linearizable and 1-writer multireader regular variables from 1-writer 1-reader safe bits. Haldar and Vidyasankar [9] present a construction of 1-writer multireader linearizable variable from 1-writer multireader regular variables. Thus, we can implement wait-free Read/Write linearizable shared variables from safe bits.

Let us look at one implementation-specific characteristic of all these constructions. In these constructions, when a higher-level operation execution returns, there is no pending lower-level operation execution to be carried out asynchronously, and hence, no higher-level operation execution makes early return. The question is: Can we construct Read/Write variables with optimized operation executions? We discuss this issue in this subsection. We point out which types of variables can (and cannot) have Write optimizations in the next sub-subsection. In the following sub-subsection we use this result to show that sequentially consistent variables alone cannot wait-freely implement a linearizable variable.

3.4.1 Write optimization

In computer hardware domain, we are familiar with two types of basic building blocks: flip-flop and latch. They are the same hardware, but operate on different logic (Cf. [27]). A flip-flop is edge (the rising edge of a driving clock) triggered, whereas a latch is level triggered. The latch takes the entire clock pulse to change its state; the flip-flop finishes early. The same concept may be applied to operation executions at various levels of system building.

For each operation execution on a higher-level shared variable V, the executing processor first issues an invocation request (with appropriate arguments) to the variable interface. The request is appropriately translated into read/write requests on primitive variables used in the physical representation of V by the interface (or its agents) which sends a reply back to the requesting processor. This entire end-to-end time span (in any time reference) of the operation execution may be imagined as a single pulse. Let us call it an *operation pulse*. An operation pulse is considered complete when no agents are involved any more in the operation execution. (Note that the original requesting processor considers the operation execution to be over right when it receives back a notification reply. This is irrespective of whether the operation pulse is complete or not. The processor does not know this and can start a new operation execution immediately.)

By an *edge triggered* Write of a shared variable V, we mean that, on receiving the write invocation request, the interface of V sends a reply back to the issuing processor before completing 'all' required low-level read/write actions on the primitive variables. That is, the interface does not wait until the entire operation pulse is complete; the operation pulse continues in the background until all the involved primitive operation executions are complete. Some agents will carry out these pending primitive operation executions on behalf of the original issuing processor, but we do not know when the agents will finish the pending work. (For example, for a write operation execution, as done in some shared variable implementations, the interface may write the value into a local cache and then return the acknowledgment to the writer. See [13]. The value is propagated to other caches asynchronously by agents of the cache.[21] The operative word is asynchronous propagation, and not a time bound propagation.) Note that a Read of V occurring after a Write in the global time order may not be able to trace the value of the Write. (In Figure 1, for the optimized part, the Read R could not trace the value of Write W_2 even though $W_2 \longrightarrow_{gt} R$ at the higher level.) By a *level triggered* Write of V, we mean that the interface sends a reply back to the issuing processor only after all representations of V are appropriately updated. That is, the entire operation pulse is complete, and no more actions remain pending after the Write is complete. All Reads occurring after the Write in the global time order will be

[21] You may note that a Read from another processor may not be able to trace the value in a bounded amount of time if its request to the value-holding cache is not answered in a bounded time. Even if a time bound is assured, whose time reference is used to measure the bound? The reading processor may speed up its clock and declare a timeout.

able to trace the value of the Write. We note that, for efficiency, we may early complete the Write, but at the least, the value of the Write must be traceable to all new Reads R such that $W \longrightarrow_{gt} R$ and R can do so in a bounded amount of time. The operative word is bounded time traceability. This is different than edge triggered case where W may not be traceable to R in a bounded time. That is the difference between level and edge triggers. In the edge trigger case, R may not see any space-time point of W, but in the level trigger case it will see some space-time points.

A Read of a shared variable V can also be edge or level triggered. No sooner than the interface (of V) can decide on a consistent value for the Read, it can return the value to the Read. Thus, the Read is edge triggered and the interface may not need to consult all processors before completing the Read. However, after completing a Read, it does not make much sense to continue the Read's effect asynchronously as the Read does not change the value of V. So, we assume Reads are always level triggered. Consequently, the shared variable interface has the liberty to decide as to when acknowledgments are returned to the Writes. In summary, a level triggered Write informs all future Reads about the new value written by it, but an edge triggered Write may not do so. Thereby, at the interface of V, edge triggered Writes appear to be faster than level triggered ones.

It is interesting to note that all Reads and Writes on each primitive variable are always level triggered. This is because by assumption primitive variables are 1-writer 1-reader, and the writer processor cannot early-return from a Write of the variable and then come back and finish the Write. (If it does so, the Write will be treated as multiple Writes on the primitive variable.) No other agent can finish the Write also since otherwise the variable becomes 2-writer. The question is: do all consistency specifications require to have only level triggered Writes for higher-level shared variables? The answer is perhaps no. The challenge is to effectively construct edge triggered Writes at higher-levels using these level-triggered primitive variables. But, we may not be able to do write optimizations for all consistencies. This is partly addressed in the following sub-subsection.

3.4.2 Some impossibility results

Attiya and Welch [5] present a quantitative analysis to show that linearizability is a costly consistency condition compared to sequential consistency. What is the fundamental reason behind this? The answer is given in the following theorem.

Theorem 1 *Linearizability does not permit edge triggered Writes.*

Proof: For a defining relation ρ, ρ-atomic executions of a shared variable are linearizable if ρ includes the global time order of operation executions [10]. It means that there are no read-illegalities in ρ_e, the exclusion closure of ρ. To avoid all read-illegalities, any implementation of a linearizable variable V must have at the least the following property: Each Write W of V, which precedes a Read R of V in the global time order $\longrightarrow_{gt}$ (i.e., $W \longrightarrow_{gt} R$), must leave some information that is "traceable" by the future Read R, and R must be able to decide, in a bounded time, on the latest value of V from this traceable information. You may note that "W is not traceable to R" implies that no space-time point for W precedes or coincides that of R. As primitive variables are our only communication medium and persistent storage, we need to map the (higher level) $\longrightarrow_{gt}$ ordering into some (lower level) primitive-object ordering(s) $\longrightarrow_x$ for some primitive variable(s) x. Consequently, by the principle of space-time coincidence and storage persistence behavior, W or some agents of W and R must either (1) overlap at the same space (at least one primitive variable) and at the same time (the primitive variable's time) on each other to have a space-time coincidence for information transfer, or (2) W or its agent precedes R on that space in the time reference of that space. This implies that the shared variable interface of V (acting on behalf of the Write W), before replying to W, must induce space-time point(s) by writing some primitive variables x (in the representation of V) that would be read by the Read R. That is, there is at least one primitive variable x such that $W(x) \dashrightarrow_x R(x)$ by the time reference T_x of x. As x is a regular variable, to avoid flickering behavior, we must have $W(x) \longrightarrow_x R(x)$ before W is complete. This is true for all primitive variables x whose values R uses to construct a value of V. Thus, the interface of V cannot reply to W before it completes writing some primitive variables x that will be read by R to construct the value it would return. By doing this, the interface would translate the global time order $\longrightarrow_{gt}$ between W and R into strong causal orders in some time references T_x (Cf. Time Axioms T5 and T6).

Side bar: This is the fundamental problem to be addressed in any implementation of linearizable shared variables, that is, the implementation must translate global time ordering into strong primitive-object orders. Thereby, the higher-level Write-Read ordering in $\longrightarrow_{gt}$ gets translated into (lower level) strong primitive-object ordering in $\longrightarrow_x$ for some primitive variables x.

This is true for a Read coming from any processor that is allowed to read V. Thereby, the Write must be complete for all those processors, and Reads from the processors occurring after the Write is complete must be able to trace the value of the Write. As linearizability is a nonblocking property,[22] in the absence of new Writes, at least one Read must be able to complete its execution in a bounded time (of the reading processor). Thus, Writes cannot be edge triggered (that is, complete for some processors and incomplete for others), and they must be level triggered; otherwise some Read will be blocked forever. $\Box$

Theorem 1 is perhaps a good indication why linearizability is a local property: a system as a whole is linearizable when every individual variable is linearizable [11]. The phenomenon described in the sidebar in the proof of the above theorem may have been observed by Garg and Raynal [7]. They conclude that the use of the global time ordering in the original definition of linearizability in [11] is not necessary when operation executions are unary (involving only one object), and lower-level object orders are sufficient. Note that under level triggered operation executions, each higher-level operation execution encompasses all its lower-level operation executions, and hence object orders will include the global time order. In the following claim we show that for sequential consistency, in contrast, all Writes need not be level triggered.

Claim 1 *Sequential consistency does permit edge triggered Writes.*

Proof: Sequential consistency does not give importance to the global time ordering of higher-level operation executions of different processors as far as determining the defining condition is concerned at the higher level. It ensures atomicity for the defining relation $(\bigcup_i \longrightarrow_i \cup \longrightarrow_{rf})^*$ on the higher-level operation executions, where $\longrightarrow_{rf}$, the read-from relation

[22]A *nonblocking property* implies a pending execution of a totally defined operation is never required to wait for another pending operation execution to complete. Some particular implementation may block an operation execution, but it is not an inherent property of linearizability.

specifies which Reads read from which Writes. Consider a simple execution scenario of a sequentially consistent variable V in which there are a Write W from a processor P_1 and a Read R from a different processor P_2 such that $W \longrightarrow_{gt} R$. For this execution scenario, sequential consistency permits the Read R to return the initial value of V. Then, $\langle R, W \rangle$ is a valid total order for the given execution to be sequentially consistent. Thus, the value of W need not be traceable to R. Thereby, for all the common primitive shared variables x that W writes and R reads, we can have $R(x) \longrightarrow_x W(x)$ at the primitive level. So, there is no need for $W(x)$ to be completed before the interface of V acknowledges the completion of W. These Writes on x can be scheduled later by the interface of V. Thus, higher-level Writes W need not be level triggered (because W need not be traceable to Reads of V from other processors when W is complete), and we can make them edge triggered.

Side bar: The value of W will be asynchronously transferred to all primitive variables so that other processors eventually see the value of W. The point to note here is that the semantics of sequential consistency does not enforce a time bound for the agents to complete the asynchronous transfers even though some specific implementations may do.

□

It has been shown in the literature that sequential consistency is weaker than linearizability: for example sequential consistency is not a local property, but linearizability is [11]. Here we have given an alternative proof from a different perspective. We conjecture that any lower-level object that permits edge triggered Writes may not be used alone to implement wait-free linearizable variables. We substantiate this conjecture by proving below that sequentially consistent variables alone cannot wait-freely implement a linearizable variable.

Theorem 2 *We cannot (deterministically) construct a wait-free 1-writer 1-reader linearizable variable from a finite number of 1-writer 1-reader sequentially consistent variables.*

Proof: Suppose, on the contrary to the statement of the theorem, that a 1-writer 1-reader linearizable variable V is wait-freely constructed from a finite collection of 1-writer 1-reader sequentially consistent variables $x_1, x_2, \cdots, x_m$ and $y_1, y_2, \cdots, y_n$. The former are written by the writer processor P_w and read by the reader processor P_r, and the latter are written by P_r and read by P_w. Let the initial value of V be v represented by $x_i = v_i$, for $i = 1, 2, \cdots, m$ and $y_j = u_j$, for $j = 1, 2, \cdots, n$. Suppose the first Write W of V by P_w writes v' in it. By Theorem 1, it is a level-triggered Write; and, it has to make its value traceable to Reads of V by P_r. The Write W may read variables y_j and write variables x_i a bounded number of times (by the nonblocking property of the linearizability of V). Suppose, when W is complete, the values of variables x_i are v'_i representing $V = v'$. The values of y_j remain the same. The reader P_r starts the first Read R of V after the Write W is complete. That is, $W \longrightarrow_{gt} R$. (Global observers will be able to see this, but P_r may not.) The Read R may read variables x_i and write variables y_j, a bounded number of times to determine the value of V (again by the nonblocking property of linearizability). As lower-level variables are sequentially consistent, the Writes on each sequentially consistent variable from one processor may not affect the Reads of the variables from the other processor. That is, for any (lower-level) Write $W_{P_w}(x_i)$ and (lower-level) Read $R_{P_r}(x_i)$, we can have $R_{P_r}(x_i) \longrightarrow_{x_i} W_{P_w}(x_i)$ at the interface of x_i. Similarly, for the y_j variables, we can have $R_{P_w}(y_j) \longrightarrow_{y_j} W_{P_r}(y_j)$ at the interface of y_j. Then, R would read v_i (the initial value) from x_i. As R reads the initial values from sequentially consistent variables x_i, it cannot trace the value of W and it will return the initial value v from V. This violates the linearizability of V, because by virtue of $W \longrightarrow_{gt} R$, R should have returned the value of W. The theorem follows. $\square$

4 Concluding remarks

In practice, we should be able to determine consistencies ensured by shared variable implementations. To classify executions of these implementations, we first need to define abstract higher-level system executions from the lower-level ones. We must have some means or tools to do this. Lamport [18] defines an implementation relation for this purpose.

At the primitive level, the shared variables are 1-writer 1-reader regular bits. To apply Lamport's implementation relation, we need to define system executions for the primitive variables. For this, we need to define $\longrightarrow$ and $\dashrightarrow$ precedence relations on primitive operation executions. Lamport assumes the Axiom B2 for this purpose. We wanted to investigate the rationale behind this axiom.

For asynchronous distributed systems, it is tacitly assumed that processors do not have access to any form of time references. But, we do not explicitly say so for the shared variables. We conjecture that it is not possible to define overlapping operation executions on a primitive shared variable without the assumption of a linear time reference for that variable. The concept of time is vital to our sense of thinking, understanding, and reasoning. We implicitly assume, in our subconscious mind, the existence of such a time reference for each primitive shared variable in the system. We take a cue from the special relativity theory of Physics, and claim that a time reference (in fact, space-time coincidence) is needed to study the behavior of operation executions on a primitive variable. These time references provide us a mental framework to define the "cause-effect" relationship on operation executions of two different processors. Sequential consistency is expensive to implement as it must promote these implicit shared variable time references to a unique time reference for all operation executions. Linearizability, in addition to those of sequential consistency, needs to respect the order of operation executions separated in the global time. These time references are akin to Lamport's logical clock [14]; we can think in terms of each communication channel having a virtual clock that ensures that a message sent is never delivered in the temporal past. We may note that Axiom B2 is same as the global-time axiom at the primitive level.

Several studies have been done for implementing stronger consistencies from weaker ones: causal consistency to sequential consistency [25], sequential consistency to linearizability [5], regularity to atomicity [9, 18], etc. There are a lot of things that are not clear to us from these implementations. By Theorem 2, we cannot wait-freely construct a 1-writer 1-reader linearizable variable from a finite number of 1-writer 1-reader sequentially consistent variables. This implies that we cannot wait-freely construct linearizable memory from sequentially consistent memory without introducing other (special synchronization) primitives to access shared variables. What are the weakest required primitives for such purposes? The proposed constructions (Cf. [5]) in the literature use message passing system for interprocessor communications. The message passing system definitely plays a non trivial role and ensures these primitives even though we do not know what they are. We need to investigate these constructions to find out the weakest set of primitives required to wait-freely implement linearizable variables from sequentially consistent variables. The next question is where does 1-writer 1-reader sequentially consistent variables reside in Herlihy's hierarchy [12]?

During establishing our axioms we assumed non-zero finite time intervals for operation executions. This is purely for convenience in understanding the axioms and results presented in this paper. We may not need such stringent assumptions. At one extreme, an operation execution can be a single time moment, causing a single space-time coincidence, instead of spanning a time interval. At the other extreme, a time interval can be infinite too, taking care of failed operation executions.

References

[1] Abraham, U., and Ben-David, S. 1987. Informal and formal correctness proofs for programs (for the critical section problem). Preprint.

[2] Abraham, U. 1995. On interprocess communication and the implementation of multi-writer atomic registers. *Theoretical Computer Science 149*, 2, 257–298.

[3] Ahamad, M., Neiger, G., Burns, J., Kholi, P., Hutto, P. 1995. Causal memory: definitions, implementation, and programming. *Distributed Computing, 9*, 37–49.

[4] Anger, F. 1989. On Lamport's interprocess communication model. *ACM Transactions on Programming Languages and Systems, 11*, 3, 404–417.

[5] Attiya, H., and Welch, J.L. 1994. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems, 12*, 2, 91–122.

[6] Ben-David, S. 1988. The global time assumption and semantics for concurrent systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing.* ACM, New York, pp. 223–231.

[7] Garg, V.K., and Raynal, M. 1999. Normality: A consistency condition for concurrent objects. *Parallel Processing Letters, 9*, 1, 123–134.

[8] Goodman, J. Cache Consistency and Sequential Consistency. Technical Report No. 61, SCI Committee, March 1989.

[9] Haldar, S., and Vidyasankar, K. 1995. Constructing 1-writer multireader multivalued atomic variables from regular variables. *J. ACM 42*, 1, 186–203.

[10] Haldar, S., and Vidyasankar, K. 2007. On specification of Read/Write shared variables. *Journal of the ACM 54*, 6.

[11] Herlihy, M., and Wing, J. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12*, 463–492.

[12] Herlihy, M. 1991. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst. 13(1)*, 124–149.

[13] Higham, L., Jackson, L., and Kawash, J. 2007. Specifying memory consistency of write buffer multiprocessors. *ACM Trans. Comput. Syst. 25*, 1, 1–42.

[14] Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7, 558–565.

[15] Lamport, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transaction on Computer, C-28*, 9, 690–691.

[16] Lamport, L. 1979. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems, 1*, 1, 84–97.

[17] Lamport, L. 1986. The mutual exclusion problem: part I — a theory of interprocess communication. *J. ACM 33*, 2, 313–326.

[18] Lamport, L. 1985. On interprocess communication — Part I: Basic formalism, Part II: Algorithms. *DEC SRC Report 8.* (Also in *Distributed Computing, 1*, 1986, 77–101.)

[19] Lamport, L. 1997. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transaction on Computer, C-46*, 7, 779–782.

[20] Li, M., Tromp, J., and Vitányi, P.M.B. 1996. How to share concurrent wait-free variables. *J. ACM 43*, 723-746.

[21] Lipton, R.J., and Sandberg, J.S. 1988. PRAM: a scalable shared memory. Tech Rep 180-88, Department of Computer Science, Princeton University, Sept 1988.

[22] Mattern, F. 1989. Virtual Time and Global States of Distributed Systems. In : Cosnard M. et al. (eds): *Proceedings of the Workshop on Parallel and Distributed Algorithms*, North-Holland / Elsevier, pp. 215-226. (Reprinted in: Z. Yang, T.A. Marsland (eds.), "Global States and Time in Distributed Systems", IEEE, 1994, 123-133.)

[23] Mattern, F. 1992. On the Relativistic Structure of Logical Time in Distributed Systems. In *Datation et Controle des Executions Reparties, Bigre 78*, (ISSN 0221-525), pp. 3-20.

[24] Misra, J. 1986. Axiom for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems, 8*, 1, 142–153.

[25] Raynal, M., and Schiper, A. 1995. From causal consistency to sequential consistency in shared memory systems. In *Proceedings of Foundations of Computer Science and Theoretical Computer Science.* Lect Notes Comput Sci, vol 1026, Springer, Berlin Heidelberg New York 1987, pp. 180–194.

[26] Singh, A.K., Anderson, J.H., and Gouda, M.G. 1994. The elusive atomic register. *J. ACM 41*, 311-339.

[27] Tannenbaum, A. S. 1990. *Structured computer organization.* Prentice-Hall Inc.