The Complexity of Weak Consistency

Gaoang Liu, Xiuying Liu

The Complexity of Weak Consistency

Gaoang Liu[1,2] and Xiuying Liu[2,3]

[1] State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Science
[2] University of Chinese Academy of Sciences
`Gaoang@ios.ac.cn`
[3] Wispirit Technology Ltd
`Liuxiuying@66nao.com`

Abstract. Weak consistency is a memory model that is frequently considered for shared memory systems. Its most distinguishable feature lies in a category of operations in two types: data operations and synchronization operations. For highly parallel shared memory systems, this model offers greater performance potential than strong models such as sequential consistency by permitting unconstrained optimization on updates propagation before synchronization is invoked. It captures the intuition that delaying updates produced by data operations before triggering a synchronization operation does not typically affect the program correctness.
To formalize the connection between concrete executions and the corresponding specification, we propose in this work a new approach to define weak consistency. This formalization, defined in terms of distributed histories abstracted from concrete executions, provides an additional perception of the concept and facilitates automatic analysis of system behaviors. We then investigate the problems on verifying whether implementations have correctly implemented weak consistency. Specifically, we consider two problems: (1) the testing problem that checks whether *one* single execution is weakly consistent, a critical problem for designing efficient testing and bug hunting algorithms, and (2) the model checking problem that determines whether *all* executions of an implementation are weakly consistent. We show that the testing problem is NP-complete, even for finite processes and short programs. The model checking problem is proven to be undecidable.

1 Introduction

Many modern computer architectures and multi-core processors support shared memory in hardware, a design that facilitates fast access and provides user-friendly programming perspective to memory. A shared memory system permits concurrent accesses from multiple processes to a single shared address space. To avoid undesirable behaviors, memory consistency must be properly maintained. Informally, a memory consistency specifies the guarantees that a system makes on the value of read operations from the shared memory. Strong models such as atomic consistency (also known as linearizability) [1] and sequential consistency (SC) [2] are intuitively composed but restrictive in performance as they would severely restrict the set of possible optimization, such as pipelining write accesses and caching write operations. Also, implementing these strong consistency criteria in message-passing system is very expensive. For better performance, in the past several decades many weaker consistency models, for example weak consistency (WC) [3], causal consistency (CC) [4] and PRAM consistency [5], have been proposed, explored and revised as various attempts.

Weak consistency was first proposed by Dubois et al. as *weak ordering* [3]. The motivation for designing this model is to tackle with the logical problems of memory accesses buffering stemmed from multiprocessor systems, especially in the cache-based systems or systems with a distributed global memory. Adve and Hill generalized weak consistency to the order strictly necessary for programmers with "SC for DRF" [6], which was re-specified and served as the cornerstone for Java memory model [7] and C++ [8]. A weak consistent system distinguishes two types of variables: shared program variables that are visible to all processes; and synchronizing variables for concurrent executions synchronization. The latter can be recognized by instructions such as TEST_AND_SET (TAS), COMPARE_AND_SWAP (CAS), or special LOAD and STORE instructions. In the Java memory model, for example, to allow synchronization races but not data races, variables having potential to race must be tagged as synchronization, using keywords such as **volatile** or **atomic**. Programmers can also create synchronization locks implicitly with Java's monitor-like **synchronized** methods.

Based on the type of variable it accesses, an operation in a weakly consistent system is either a data operation and a synchronization operation. The latter works in a way similar to fence instructions, whose initiation suggests all previous references to the shared variables have completed and all future references have to wait. The idea of exploiting fence operations to achieve synchronized concurrency is also widely adopted by many commercial models. For example, the Power model [9] implemented by IBM PowerPC is similar in spirit to weak consistency by utilizing varieties of fences to synchronize concurrently executing processes. PowerPC uses an instruction called *Hwsync* to keep all writes in a consistent total order, it can also achieve sequentially consistent behaviors if Hwsync is used together with address dependencies and message passing [10]. In addition to PowerPC, another commercial example implementing a model that relies heavily on proper synchronization is ARM multiprocessor [11]. Similarly, ARM has multiple flavors of fences, including one data memory barrier that can order all memory accesses.

Generally, developing distributed implementations to satisfy a relaxed consistency model is very challenging because of the complicated issues related to communication. At different stages of development, the developers need to exploit different testing and verification techniques to validate the system they have built so far. This highlights the need for more research on the feasibility and efficiency of algorithms that are utilized during the development. We thus investigate in this paper two fundamental problems: (1) the testing problem that asks whether one given execution of an implementation is weakly consistent, and (2) the model checking problem (or verifying problem) that asks whether all the executions of an implementation are weakly consistent. The behaviors of implementations we consider in this study are restricted to the setting of `read/write` memory (RWM) abstraction, which is at the base of many distributed data structures used in practice.

We start by presenting in this work a new approach to formalize weak consistency in terms of distributed histories. This formalization associates a concrete execution of an implementation with its abstract description — a sequence that conforms to the specification for RWM, in our case. A specification is simply a characterization of admissible behaviors in the sequential setting. For a consistency model, a good characterization should, first of all, precisely captures the subtle behaviors of programs running under the model, and also be formal and convenient for developers to automatically analyze and verify their codes. Previous work [3, 6] on weak consistency elaborated on the concept with great details, but the definitions presented in both work are either too informal or excessively convoluted to be directly applied to automatic verification. This makes our new formalization a necessity, with which we show the complexity of the testing problem is NP-complete. The result is achieved by reducing the serialization problem, a NP-complete problem for database transactions to a restricted version of the testing problem, TWC-read problem. We also investigate the problems with limited accesses per process and limited processes in an instance. Both problems are proven to be NP-complete, even when the access number is limited to two and the process number to three. For the TWC-read problem restricted to two processes, we prove there exists a polynomial algorithm. We then prove that the model checking problem is undecidable by a reduction from the Post Corresponding Problem (PCP). The idea is to relate each PCP instance to a WC implementation, such that the instance has a valid witness exactly when the implementation is weakly consistent.

The remainder of this paper is organized as follows. Section 2 presents the notion of distributed history and the specification for `read/write` memory. Section 3 briefly reviews the intuition behind WC and then shows how to formalize this concept. We present in section 4 and section 5 the complexity results for the testing problem. The decidability result of the model checking problem is proved in Section 6. Section 7 discusses the related work. Finally, section 8 concludes the paper.

2 Preliminaries

2.1 Sets, Relations and Labeled Posts.

Given a set $\mathbb{E}$ and a relation $R \subseteq \mathbb{E} \times \mathbb{E}$, we denote by $e_1 \prec_R e_2$ the fact that $(e_1, e_2) \in R$. We write $(R)^+$ to denote the *transitive closure* of R. A relation is a *strict partial order* if it is transitive and irreflexive.

A *poset* is a pair $(\mathbb{E}, \prec)$ where $\prec$ is a strict partial order. Given a set Σ, a *Σ labeled poset ρ* is a tuple $(\mathbb{E}, \prec, \iota)$ where $(\mathbb{E}, \prec)$ is a poset and $\iota : \mathbb{E} \to \Sigma$ is the labeling function.

We introduce a relation on labeled posets, denoted $\rhd$. Let $\sigma = (\mathbb{E}, \prec, \iota), \sigma' = (\mathbb{E}, \prec', \iota')$ be two posets labeled by the same set Σ. We denote by $\sigma \rhd \sigma'$ the fact that σ imposes less constraints on operations in $\mathbb{E}$. Formally, $\sigma \rhd \sigma'$ if $\prec \subseteq \prec'$ and $\iota = \iota'$, i.e., for all operation $e \in \mathbb{E}, \iota(e) = \iota'(e)$.

2.2 Histories and Specifications

A distributed system is a composition of a set of processes/participants invoking methods on shared objects (registers, queues, etc.). An object implements a programming interface (API) defined by a set of *methods* $\mathbb{M}$ with input and output from a data domain $\mathbb{D}$. The behaviors of a system can be characterized by a set of executions. Each execution is a labeled poset consist of a collection of events, representing operation invocations by different participants, and a relation on events, describing abstractly how the system processes the corresponding operations. The characterization of distributed executions are what we call *distributed histories.* Formally,

Definition 1. *A distributed history is a poset $\mathcal{H} = \{\mathbb{E}, \prec_{\text{PO}}, \iota\}$ labeled by $\mathbb{M} \times \mathbb{D} \times \mathbb{D}$, where $\mathbb{E}$ is a countable set of events; $\prec_{\text{PO}} \subset (\mathbb{E} \times \mathbb{E})$ is a strict partial order called* program order*; and $\iota : \mathbb{E} \to (\mathbb{M} \times \mathbb{D} \times \mathbb{D})$ is a labeling function.*

The labeling function ι maps each event to its corresponding operation that invoke a method from $\mathbb{M}$. To model events that return no value, we use a value $\bot \in \mathbb{D}$, which is often omitted for better readability.

Example 1. For the `read/write` memory, $\mathbb{M} = \{r, w\}$ is the set of operations reading and writing variables. Let the variables set be $\mathbb{V}$, the domain $\mathbb{D}$ is therefore $(\mathbb{V} \times (\mathbb{N} \uplus \{\bot\})) \uplus \mathbb{V} \uplus \mathbb{N} \uplus \{\bot\}$. A labeling function ι maps each write event e_w to a tuple $\langle w, (v, n), \bot \rangle$ (abbreviated as $w(v, n)$) and each read e_r to $\langle r, (v, \bot), n \rangle$ $(r(v)/n)$.

The consistency of shared objects is a criterion that links the distributed executions to a particular specification, which characterizes the admissible behaviors for a program that uses the objects. For an object defined by a method set $\mathbb{M}$ and data domain $\mathbb{D}$, the specification S can be specified by a set of sequences labeled by $\mathbb{M} \times \mathbb{D} \times \mathbb{D}$.

Generally, specifications can be defined with posets of events (as in a way used in [12] instead of sequences of events that are total ordered. The former provides a more general and precise way to model conflict resolution policies. In this work, however, we focus on the `read/write` memory, as illustrated in Example 1. A sequential specification is enough to meet our needs for the modeling purpose. We define the specification $S_{\mathrm{r/w}}$ as a set of sequences where each value read matches the most recent write. Formally, it is the smallest set of sequences by inductively adopting the following rules:

- $\epsilon \in S_{\mathrm{r/w}}$,
- $\sigma \cdot w(v, d) \in S_{\mathrm{r/w}}$ if $\sigma \in S_{\mathrm{r/w}}$,
- $\sigma \cdot r(v, 0) \in S_{\mathrm{r/w}}$ if $\sigma \in S_{\mathrm{r/w}}$ and σ contains no write on v,
- $\sigma \cdot r(v, d) \in S_{\mathrm{r/w}}$ if $\sigma \in S_{\mathrm{r/w}}$ and the last write on v in σ is $w(v, d)$,

where $v \in \mathbb{V}, d \in \mathbb{D}$ and ϵ is the empty sequence.

3 Weak Consistency

3.1 Weak Consistency: Informal Description

Many consistency models have been proposed to specify shared memory. Strong consistency models such as SC [2] and linearizability [1] are intuitively composed but restrictive in performance as they would severely restrict possible optimization such as pipelining write accesses. Besides, implementing strong consistency criteria in message-passing systems are strikingly expensive. For example, the duration of reads or writes in SC systems has to be linear with the latency of the network [5]. To increase efficiency, researchers explored a variety of weak consistency criteria [3, 13, 4, 5].

Among these consistency models, WC is distinguishable from many of its counterparts by its category of memory accesses. Based on the type of accesses, it imposes different ordering constraints. For example, in the case of data operations, buffered requests (to memory) may be allowed to pass each other in the buffer. This is referred to as *jockeying* [3], which is often permitted between requests for different memory locations. On the other hand, the synchronization operations are requested to be *strongly ordered* [3], and the jockeying with data operations is forbidden. The purpose of synchronization operations is, as the name suggests, to sync updates between different processes by propagating local updates outward and bring in all updates from other processes. Before a subsequent synchronization is initiated, the propagation of local updates can be arbitrarily postponed. Informally, weak consistency ensures that reordering memory operations to shared data between synchronization operations does not typically affect program correctness.

Dubois et al. presented in [3] the first description of WC by enforcing on storage accesses the following constraints :

Definition 2. *In a multiprocessor system, storage accesses are weakly ordered if (1) accesses to synchronizing variables are strongly ordered, (2) no access to a synchronizing variable is allowed to be performed until all previous writes have completed everywhere, and (3) no data access (read or write operation) is allowed to be performed until all previous accesses to synchronizing variables have been performed everywhere.*

The idea of supporting jockeying to add efficiency has also been exploited by many other models, such as the “write to read” relaxation, which corresponds to the Total Store Order (TSO) model. Apart from jockeying, another important feature of WC is that it enforces consistency on a group of operations, rather than on individual reads or writes. For memory accesses to different locations, absent of any barriers or dependencies, the behavior can be unconstrained. This is best illustrated by the following example.

Example 2. History (a) in Figure 3.1 is weakly consistent while (b) is not. Because of jockeying, it is possible for P_3 in (a) to read the initial value of x even when a previous read on y returns the latest value. In (b), however, the sync operation forces out all previous updates to P_2, making the read $r(x, 0)$ impossible.

$$\begin{array}{llll} P_1: & w(x,1) & w(y,1) & \\ P_2: & r(y,1) & \text{SYNC} & r(x,1) \\ P_3: & r(y,1) & r(x,0) & \\ & & \text{(a)} & \end{array} \qquad \begin{array}{llll} P_1: & w(x,1) & \text{SYNC} & w(y,1) \\ P_2: & r(y,1) & r(x,0) & \\ & & \text{(b)} & \end{array}$$

Fig. 3.1. The visualization of WC, where SYNC represents a synchronization operation on a variable other than x and y. Initially, $x = y = 0$.

3.2 Weak Consistency: A Formal Definition

We now present an approach to characterize concrete executions of concurrent systems employing a weakly consistent model. This characterization links each execution to a set of sequences, among which there is at least one can provide a reasonable explanation as to why the execution is weakly consistent.

Given a history $\mathcal{H}$ with an event set $\mathbb{E}$, we denote by $\mathbb{E}_S$ the set of synchronization events and $\mathbb{E}_D$ the set of data events. Two events from different processes can be ordered only if there exists an intervening synchronization between them. To capture this property, we introduce *happen-before* relations for events in any history. Two types of relation are considered: program order $\prec_{\text{PO}}$, and synchronization order $\prec_{\text{SO}}$. Formally, let e_u and e_v be any two operations occurring in $\mathcal{H}$. Then:

- $e_u \prec_{\text{PO}} e_v$ iff e_u occurs before e_v in the same process.
- $e_u \prec_{\text{SO}} e_v$ iff $e_u, e_v \in \mathbb{E}_S \wedge var(e_u) = var(e_v)$ and e_u is performed before e_v,

where $var(e) \in \mathbb{V}$ is the variable accessed by e. The concept of *performed* is borrowed from [6], where a read is said to be *performed* at a point in time when no subsequent write, from the same or another process, can affect the value returned. Similarly, a write is said to be *performed* when all subsequent reads return the written value until another write to the same memory location is performed. For a synchronization operation, it is performed also means all preceding updates of data operations are propagated outward.

Definition 3 (Weak Order). *A weak order $\prec_{\text{WO}}$ of a history $\mathcal{H}$ is the irreflexive transitive closure of program order and synchronization order, that is $\prec_{\text{WO}} = (\prec_{\text{PO}} \cup \prec_{\text{SO}})^+$. The set of events happening before e w.r.t. the $\prec_{\text{WO}}$ is denoted by $\lfloor e \rfloor = \{e' \in \mathbb{E} : e' \prec_{\text{WO}} e\}$.*

Intuitively, $\lfloor e \rfloor$ is the *weak past* of e, i.e., the set of operations whose effects are visible to e. To associate any weakly consistent history to the sequential specification, we need to define a way to explain how events on a history are generated. For example, it should explain which write is responsible for a read that is accessing the same variable. This is achieved by defining a function, called *observation function* below, that links each event to an event set, which, if considered together with $\prec_{\text{WO}}$, will be sufficient to explain why the history is admissible by the specification $S_{\text{r/w}}$.

Definition 4. *Given a history $\mathcal{H}$, a function $\tau : \mathbb{E} \to 2^{\mathbb{E}}$ is an observation function on $\mathbb{E}$, if $\forall e \in \mathbb{E}$, $\tau(e) = \{e\} \cup \lfloor e \rfloor \cup C_e$ for some set $C_e \subseteq \mathbb{E}$ whose elements are concurrent events of e. The set $\tau(e)$ is called an observation set of e.*

The observation set $\tau(e)$ is acctually a snapshot of updates observed by e and, hence, should not deviate from the constraints imposed by WC. Apart from the constraints required by $\prec_{\mathrm{WO}}$, it should also respect what we call *update inheritance* — updates observed by one event are inherited to its successors. Specifically, if an event e observes updates happened before a synchronization, then all events observed e should too have observed those updates. Formally, for two events $e_u, e_v \in \mathbb{E}$, if

(a) $e_u \in \tau(e_v)$ and at least one of e_u, e_v belongs to $\mathbb{E}_S$, or
(b) $e_u \in \mathbb{E}_S \wedge \exists e_w.(e_w \in \tau(e_v) \wedge e_u \in \tau(e_w))$, (*)

then $\tau(e_u) \subseteq \tau(e_v)$. We call such a function a *observation closed function* (OCF).

Definition 5. *A history $\mathcal{H}$ is weakly consistent with respect to $S_{\mathrm{r/w}}$ if there exists an* OCF *τ such that for any $e \in \mathbb{E}$, there exists a sequence $\sigma_e \in S_{\mathrm{r/w}}$ such that $(\tau(e), \prec_{\mathrm{WO}}, \iota) \rhd \sigma_e$.*

Example 3. To illustrate, the history in Figure 3.1(b) has no OCF τ for its event set. For otherwise we have $e_{w(x,1)} \in \tau(e_{\mathrm{SYNC}})$ and $\tau(e_{\mathrm{SYNC}}) \subseteq \tau(e_{r(x,1)})$, and by the condition (*) the observation set $\tau(e_{r(x,0)})$ will contain $e_{w(x,1)}$, implying the update on x is observable to $e_{r(x,0)}$, whose return value should thus be 1 instead of 0.

4 The Testing Problem of Weak Consistency

We first investigate the testing problem of weak consistency (TWC), which is relevant for instance in the context of testing a given distributed object. We prove that this problem is NP-complete. This is achieved by a reduction from the serializability problem for database histories. The membership in NP can be easily proved, following from the fact that, for any given instance (history) $\mathcal{H}$, one can guess an observation function τ, and a sequence $\sigma_e \in S_{\mathrm{r/w}}$ for each event e, and then check in polynomial time whether τ is an OCF and the relation $\rhd$ in Definition 5 holds.

To prove the NP-hardness, we first define a restricted version of the TWC problem and reduce the restricted problem to the serializability problem. We consider the case in which for each read, it is known precisely which write was responsible for the value read. We call this the *TWC-read* problem. The function mapping each read to the responsible write is called a read-mapping.

The serializability problem for database transactions is one exploring the existence of a schedule that is equivalent to one that executes the transactions serially in some order. One major type of the problem is view serializability [14], in which we are given a history, a total order on a set of reads and writes, where each read or write is associated with a particular transaction. The problem asks if there is a total order on the transactions that preserves the reads-from mapping of the original history. The view serializability problem is NP-complete [14]. TWC-read is a problem more general than view serializability by permitting solutions in which the accesses for a processor (transaction) are in order but may not be consecutive. To illustrate, the instance in Figure 4.1 is a "yes" instance for TWC-read problem, but a "no" instance for view serializability, because both $P = P_1P_2$ and $P = P_2P_1$ have reads-from violations.

Lemma 1. *The TWC-read problem is* NP*-complete.*

$$\begin{aligned} P_1 :&\ \ w(x,0), w(y,1), r(x,1) \\ P_2 :&\ \ r(y,1), w(x,1) \end{aligned}$$

Fig. 4.1. A "yes" instance of the TWC-read problem

Proof. Given $\mathcal{H}$, an instance of a view serializability problem, we construct an instance of TWC-read as follows. Let v be a variable not accessed by any operation in $\mathcal{H}$, and $\#_\alpha$ a synchronization operation on a variable α. Let P_i be the sequence of operations in $\mathcal{H}$ for transaction i ($i \in \{1, .., n\}$), where each write in a transaction is assigned a unique value to write, and each read is assigned the value of the closest previous write to the same address in $\mathcal{H}$.

For all transactions i, let $P'_i = w(v,i)\#_\alpha P_i \#_\alpha r(v,i)$. Our TWC-read instance is $\mathcal{H}' = \|_{i \in \{1,..,n\}} P'_i$. The intervening $\#_\alpha$ guarantees the update to v is observed by other transactions before P_i initiates and updates from other transactions were brought in before reading v. This construction ensures that for each P'_i once the write $w(v,i)$ starts, the remainder (i.e. $\#_\alpha P_i \#_\alpha r(v,i)$) must be scheduled consecutively. If two transactions P'_i, P'_j interleaves, then at least one of them, say P'_i, will return for its last read a value other than i, and thus violates the read-from relation. It follows that $\mathcal{H}'$ is in TWC-read if and only if $\mathcal{H}$ is view serializable.

The above result on TWC-read implies that the general TWC problem is at least NP-hard, since the problem is also in P, we have the following theory.

Theorem 1. *Checking whether a distributed history $\mathcal{H}$ is weakly consistent with respect to $S_{\mathrm{r/w}}$ is* NP*-complete.*

5 Restricted TWC Problems

The previous section shows that the TWC problem is generally NP-complete. In this section, we investigate two restricted versions of TWC that consider only instances with limited number of accesses or processes. Such problems are interesting because many multiprocessors have only a small number of processors, e.g., 8, 16 or 32; and when it comes to testing, the size of instances are usually small.

We show that these restricted problems remain NP-complete, even when the number of accesses per process is limited to two and the number of processes to three. These results imply the testing problem of weak consistency is intrinsically hard. For a very rare case of this problem, in which only two processes are involved and a read-mapping is provided, we prove there exists a polynomial algorithm.

5.1 Restricting the number of accesses

We now investigate the restricted problem in which each process has at most two data operations and each data variable is written to at most twice. We show this problem is NP-complete.

The result is generated from a reduction from 3SAT. Let $\mathcal{F}$ denote a 3SAT instance. For a literal l_i in a clause, we use the notation $\mathcal{B}(l_i)$ to represent T (`True`) if l_i is a positive literal (i.e., a variable), and F (`False`) otherwise. We need to simulate the logical connectives (i.e., an OR and an AND), as well as an assignment of variables that remains in effect until the formula is evaluated. We observe that (1) a read must wait for its responsible write to occur, (2) the second access at a process must wait for the first. Then the assignment to v_i can be simulated as follows (each column represents a sequence):

$$\begin{array}{llll} w(v_i,T) & r(x,1) & w(v_i,F) & r(x,1) \\ & \#_\alpha & & \#_\alpha \\ & r(v_i,T) & & r(v_i,F), \end{array}$$

where x is initially 0 and $\#_\alpha$ represents a synchronization operation on variable α. A single write $w(x,1)$ (shown below) occurs only after the satisfiability of $\mathcal{F}$ has been simulated. Then one and only one write to v_i occurs before $w(x,1)$, ensuring that the initial assignment to each v_i must remain in effect until the satisfiability of $\mathcal{F}$ has been simulated.

An OR is simulated by separating the literals of a clause into three reads, whose executions determine the truth value of that clause. For each clause, $C_i = l_p \vee l_q \vee l_r$, we have four sequences:

$$\begin{array}{llll} r(l_p,\mathcal{B}(l_p)) & r(l_q,\mathcal{B}(l_q)) & r(l_r,\mathcal{B}(l_r)) & r(d_i,T) \\ w(d_i,T) & w(d_i,T) & w(c_i,T) & w(c_i,T) \end{array}$$

By the two observations above, this ensures that C_i is not set to T unless clause i is satisfied by the guessed truth assignment.

The AND of the clauses can be easily simulated by a sequence $r(c_1,T), r(c_2,T), ..., r(c_m,T)$, $w(x,1)$. But this sequence involves $m+1 > 2$ data operations. In order to have only two accesses per process, we separate this sequence into $m+1$ sequence: a single write $w(x,1)$ and m sequences ending with a read $r(x,0)$:

$$\begin{array}{lllll} w(x,1) & r(c_1,T) & r(c_2,T) & ... & r(c_m,T) \\ & \#_\alpha & \#_\alpha & ... & \#_\alpha \\ & r(x,0) & r(x,0) & ... & r(x,0) \end{array}$$

This ensures that x is not set to 1 unless all clauses have been satisfied by the guessed assignment.

Lemma 2. *Let $\mathcal{F}$ be an instance of 3SAT, and $\mathcal{W}$ the instance of the TWC problem constructed as described above. Then $\mathcal{W}$ is weakly consistent if and only if $\mathcal{F}$ is satisfiable.*

Proof (sketch). Assume there is a satisfying assignment for $\mathcal{F}$. We can construct a corresponding schedule for $\mathcal{W}$ such that it can be sequentially ordered by that schedule. That is $\mathcal{W}$ is sequentially consistent and, hence, weakly consistent. Conversely, if $\mathcal{W}$ is weakly consistent, the first value written to each variable v_i forms the satisfying assignment. (A full proof is deferred to Appendix A.1).

By this result, the following theory is straightforward.

Theorem 2. *The TWC problem, restricted to instance in which each sequence contains at most two data operations and each data variable occurs in at most two write operations, is* NP*-complete.*

5.2 Restricting the number of processes

We have shown that the TWC problem is NP-complete with $O(n)$ processes (n is the number of data variables). In this problem, the number of processes per instance grows proportionally with the number of variables, this raises another question: what is the complexity for TWC problems with a fixed small number of processes. This question is answered by the theory below, showing that the problem remains NP-complete even when the number of processes per instance is limited to three.

To simulate an instance of 3SAT, the proof of Lemma 2 constructs a disjoint set of processes for each variable. This strategy can not be transferred here as the number of processes is limited.

We consider to analyze the problem with a reduction from one-in-three 3SAT problem, a variant of 3SAT where the input instance is the same, but the question is to determine whether there exists a satisfying assignment so that exactly one literal in each clause is set to true. This problem is known to be NP-complete. A monotone version of this problem, positive one-in-three 3SAT, where each clause contains only positive literals, remains NP-complete. Our strategy is, for a given positive one-in-three 3SAT instance $\mathcal{F}$, we construct an instance $\mathcal{W}$ of the TWC problem, using only three processes, such that $\mathcal{F}$ has a satisfying assignment exactly when $\mathcal{W}$ is weakly consistent. The TWC instance $\mathcal{W}$ for each $\mathcal{F}$ is depicted in Figure 5.1.

The synchronization parts (as illustrated by # in the figure) separate the construction into $m+1$ stages. The key idea behind this construction is to use the value at the end of the (INIT) stage as the assignment. The history is weakly consistent if every stage is weakly consistent. For each clause, each of the three processes is satisfied by a particular assignment. The subtle part is the writes highlighted in blue. Once a way of satisfying a clause is settled, the writes free up the other two processes by negating variables, and then return all variables to their initial setting (for the next stage). Conversely, for any assignment of $\mathcal{F}$ that does not satisfy this clause, there is no way to prove the weak consistency of this stage.

	P_1	P_2	P_3
(INIT)	$w(v_1, F)$	$w(v_1, T)$	
	...	...	
	$w(v_n, F)$	$w(v_n, T)$	
(#)			$r(u, 1)$
	$w(u, 1)$	$w(u, 2)$	$r(u, 2)$
	$\#_\alpha$	$\#_\alpha$	$\#_\alpha$
	$r(u, 3)$	$r(u, 3)$	$w(u, 3)$
(C_1)	$r(v_{x_1}, T)$	$r(v_{y_1}, T)$	$r(v_{z_1}, T)$
	$r(v_{y_1}, F)$	$r(v_{z_1}, F)$	$r(v_{x_1}, F)$
	$r(v_{z_1}, F)$	$r(v_{x_1}, F)$	$r(v_{y_1}, F)$
	$w(v_{x_1}, F)$	$w(v_{y_1}, F)$	$w(v_{z_1}, F)$
	$w(v_{y_1}, T)$	$w(v_{z_1}, T)$	$w(v_{x_1}, T)$
	...	...	...
(#)	...	...	...
(C_m)	...	...	...

Fig. 5.1. Transfroming an instance of positive one-in-three 3SAT to an instance of TWC. The 3SAT instance contains n variables $v_1, ..., v_n$ and m clauses: $C_1, ..., C_m$, where $C_i = (v_{x_i} \vee v_{y_i} \vee v_{z_i})$ for some $x_i, y_i, z_i \in [1..n]$.

Theorem 3. *The TWC problem restricted to three processes is* NP*-complete.*

In the above reduction, each clause of the 3SAT instance requires at least three processes for the simulation procedure. This is the simplest reduction we are aware of, leaving open the TWC problem restricted to two processes. Nevertheless, we prove below there is a polynomial algorithm for TWC-read problem restricted to two processes.

5.3 TWC-read problem with two processes is in P

For an instance $\mathcal{H}$ of two-processes TWC-read problem, every read in $\mathcal{H}$ knows precisely its responsible write, which means if two write $w(v, d_1), w(v, d_2)$ are accessing the same location v, then d_1 and d_2 must differ. To sovle the two processes TWC-read problem, we begin by constructing an OCF, τ, capturing all events been observed by each access. The instance $\mathcal{H}$ is in TWC-read exactly when there exists a witness OCF that respects certain constraints, and involves none of the anomalous forms (as defined in the proof for Lemma 3). Constructing this OCF and checking whether certain conditions are met can be done in polynomial time. That is, there is a polynomial algorithm to solve this problem in two steps: (1) constructing for an instance a witness OCF, (2) checking whether this OCF meets required conditions. The proof for the following lemma is deferred to Appendix A.3.

Lemma 3. *The TWC-read problem restricted to two processes is in* P.

6 Undecidability of Verifying Weak Consistency

We consider in this section the model checking problem of weak consistency, i.e., the problem of deciding whether a given implementation has correctly implemented weak consistency. For systems maintaining memory coherence, weak consistency implies sequential consistency if every data operation is synchronized by the same sync variable [15]. This implies the model checking problem of WC because the same problem of SC is generally undecidable [16]. However, here we offer another proof, which may provide some additional insight into the reason why verifying weak consistency is undecidable. For the proof, we assume the implementations are regular and specification is restricted to $S_{\mathrm{r/w}}$ with a fixed number of variables whose domain sizes are fixed, such that this specification corresponds to a particular regular language. We then reduce PCP to the model checking problem.

Definition 6. *Let Σ be an alphabet with at least two letters. An instance of PCP is given by two sequences $U = \{u_1, ..., u_n\}$ and $V = \{v_1, ..., v_n\}$ of words over Σ. The problem is to determine whether there is a sequence $(i_1, ..., i_p)$ with $i_j \in \{1, ..., n\}$ and $p > 1$ such that $u_{i_1} \cdots u_{i_p} = v_{i_1} \cdots v_{i_p}$.*

Theorem 4. *[17] The Post Correspondence Problem is undecidable.*

A pair of words $\langle u, v\rangle \in \langle \Sigma^* \times \Sigma^*\rangle$ is a *witness* of a PCP instance $\mathcal{P}$ if they can be decomposed into $u = u_{i_1} \cdot u_{i_2} \cdots u_{i_p}$ and $v = v_{i_1} \cdot v_{i_2} \cdots v_{i_p}$ such that $u_i = U[i]$ and $v_i = V[i]$. If there is also $u = v$, we call such a pair a *valid witness*, which corresponds to a positive answer to the PCP problem. Our goal is to build an implementation $\mathcal{I}$ that is *not* weakly consistent with respect to the `read/write memory` if and only if the instance $\mathcal{P}$ has a valid witness. That is the implementation $\mathcal{I}$ produces, for each pair of words $\langle u, v\rangle$, an execution $\mathcal{H}_{uv}$ that is not weakly consistent if and only if $\langle u, v\rangle$ forms a valid witness. The construction of each history $\mathcal{H}_{uv}$ relies on ten processes and seven variables (six data variable and one synchronization variable). To conserve space, details of proof are given in Appendix A.4.

Theorem 5. *Given an implementation $\mathcal{I}$ as a regular language, checking whether all executions of $\mathcal{I}$ are weakly consistent with respect to $S_{\mathrm{r/w}}$ is undecidable.*

7 Related Work

For the testing problem of relaxed memory consistency models, Wei et al. [18] proved that complexity of testing PRAM consistency is NP-complete. Bouajjani et al. [19] studied the complexity of verifying causal consistency for one history. It was proved that the problem is NP-complete for all the three variations of CC (causal consistency, causal convergence and causal memory). A recent work by Furbach et al. [20] showed that the testing problem for any criterion weaker than sequential consistency and stronger than slow consistency is NP-complete. This range covers many relaxed memory consistency models, including (a weaker variation of) CC, TSO, PSO and PRAM consistency, but it does not cover WC. It is easy to construct executions that conform to WC but violate slow consistency.

The model checking problem for linearizability, quasi-linearizability and SC have been extensively studied. It was shown that verifying linearizability is EXPSPACE-complete when the number of processes is bounded and undecidable otherwise [21]. Alur et al. [16] proved that checking sequential consistency is in general undecidable. The same conclusion holds for systems with only four objects. Wang et al. [22] studied the model checking problem of quasi-linearizability and proved that it is undecidable. For consistency criteria weaker than sequential consistency, eventual consistency has been shown to be decidable [12], and causal consistency was proved to be undecidable [19] in general.

The method we used to formalize weak consistency in terms of distributed histories is inspired by the work of [23], which extends the definition of CC to all abstract data types. Their work uses transition systems to specify sequentially abstract data types, which is modeled as transducers, a model that is very similar to Mealy machines [24]. Their approach for CC, however, does not easily transfer here. For WC, we have to consider the different roles played by data and synchronization operations, while causal consistency does not distinguish memory access categories.

8 Conclusion

This paper explores the complexity of deciding whether an execution of a shared memory system is weakly consistent. We prove that the TWC problem is NP-complete, even for systems with only three processes or programs in which each process is permitted to have only two memory (data) accesses. We show the TWC-read problem is also NP-complete, which implies tagging each read with the identity of the responsible write does not reduce the complexity. However, for TWC-read, if we restrict the process number to two, then a polynomial algorithm exists.

A new approach is proposed to formalize weak consistency. This new formalization, unlike those from previous work [6, 3], is given in terms of distributed histories abstracted from concrete executions, making possible a direct application of this formalization into automatic verification. We have also explored the model checking problem of weak consistency. Generally, deciding whether an implementation has correctly implemented the read/write memory is undecidable, even when the implementation and specification are both regular. These results on TWC and the model checking problem suggest that reasoning about weak consistency is intrinsically hard.

Although the read-mapping does not help with reducing the complexity, it would be interesting to investigate the TWC and model checking problem for implementations under certain constraints, such as data independence, a property ensuring the system behaviors are independent to particular data values stored at particular memory locations. These investigations remain future work. Moreover, as release consistency [25] is a consistency model that refines weak consistency by slitting synchronization into two types, the results presented in this work may well be applicable (with minor adaptations) to release consistency.

References

[1] Herlihy, M.P., Wing, J.M.: Linearizability - a correctness condition for concurrent objects. Acm Transactions on Programming Languages and Systems **12**(3) (1990) 463–492

[2] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. Ieee Transactions on Computers **28**(9) (1979) 690–691

[3] Dubois, M., Scheurich, C., Briggs, F.: Memory access buffering in multiprocessors. In: ACM SIGARCH Computer Architecture News. Volume 14., IEEE Computer Society Press (1986) 434–442

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

[5] Lipton, R.J., Sandberg, J.S.: PRAM: A scalable shared memory. Princeton University, Department of Computer Science (1988)

[6] Adve, S.V., Hill, M.D.: Weak ordering - a new definition. In: ACM SIGARCH Computer Architecture News. Volume 18., ACM (1990) 2–14

[7] Manson, J., Pugh, W., Adve, S.V.: The Java memory model. Volume 40. ACM (2005)

[8] Boehm, H.J., Adve, S.V.: Foundations of the c++ concurrency memory model. In: ACM SIGPLAN Notices. Volume 43., ACM (2008) 68–78

[9] IBM: Power ISA$^{\text{TM}}$ Version 2.06 Revision B. (2010)

[10] Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding power multiprocessors. ACM SIGPLAN Notices **46**(6) (2011) 175–186

[11] Alglave, J., Fox, A., Ishtiaq, S., Myreen, M.O., Sarkar, S., Sewell, P., Nardelli, F.Z.: The semantics of power and arm multiprocessor machine code. In: Proceedings of the 4th workshop on Declarative aspects of multicore programming, ACM (2009) 13–24

[12] Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. In: ACM SIGPLAN Notices. Volume 49., ACM (2014) 285–296

[13] Goodman, J.R.: Cache consistency and sequential consistency. University of Wisconsin-Madison, Computer Sciences Department (1991)

[14] Papadimitriou, C.: The theory of database concurrency control. (1986)

[15] Gharachorloo, K.: Memory consistency models for shared-memory multiprocessors. PhD thesis, Stanford University (1995)

[16] Alur, R., McMillan, K., Peled, D.: Model-checking of correctness conditions for concurrent objects. 11th Annual Ieee Symposium on Logic in Computer Science, Proceedings (1996) 219–228

[17] Post, E.L.: A variant of a recursively unsolvable problem. Bulletin of the American Mathematical Society **52**(4) (1946) 264–268

[18] Wei, H., De Biasi, M., Huang, Y., Cao, J., Lu, J.: Verifying pram consistency over read/write traces of data replicas. arXiv preprint arXiv:1302.5161 (2013)

[19] Bouajjani, A., Enea, C., Guerraoui, R., Hamza, J.: On verifying causal consistency. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, ACM (2017) 626–638

[20] Furbach, F., Meyer, R., Schneider, K., Senftleben, M.: Memory-model-aware testing: A unified complexity analysis. ACM Transactions on Embedded Computing Systems (TECS) **14**(4) (2015) 63

[21] Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Verifying concurrent programs against sequential specifications. Programming Languages and Systems **7792** (2013) 290–309

[22] Wang, C., Lv, Y., Liu, G., Wu, P.: Quasi-linearizability is undecidable. In: Asian Symposium on Programming Languages and Systems, Springer (2015) 369–386

[23] Perrin, M., Mostefaoui, A., Jard, C.: Causal consistency: beyond memory. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM (2016) 26

[24] Mealy, G.H.: A method for synthesizing sequential circuits. Bell System Technical Journal **34**(5) (1955) 1045–1079

[25] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory consistency and event ordering in scalable shared-memory multiprocessors. Volume 18. ACM (1990)

A Appendix

A.1 TWC problem with restricted accesses

Lemma A.1. *Let $\mathcal{F}$ be an instance of 3SAT, and $\mathcal{W}$ the instance of the TWC problem constructed as described above. Then $\mathcal{W}$ is weakly consistent if and only if $\mathcal{F}$ is satisfiable.*

Proof (sketch). We show that if $\mathcal{F}$ is satisfiable, we can construct a schedule by which $\mathcal{W}$ is sequentially consistent, which implies $\mathcal{W}$ is weakly consistent. Conversely, if $\mathcal{W}$ is weakly consistent, the first value written to each variable v_i is the satisfying assignment.

Assume, without loss of generality, $\mathcal{F}$ is satisfiable with an assignment $\mathcal{A}$ in which every variable v_i is set to be true. We construct a schedule S as follows: Firstly, the writes $w(v_i, T)$ are executed, which are followed by the reads of positive literals $r(l_i, \mathcal{B}(l_i))$ in each clause C_j, while the reads of negative literals and the following writes on d_i and c_i are postponed to the end. This ensures at least one $w(c_i, T)$ occurs before each pair of $r(c_i, T)$ and $r(x, 0)$. The write $w(x, 1)$ is then executed, which is followed by the sequences with $r(v_i, T)$ (the sequence order does not matter). When every $r(v_i, T)$ is executed, the sequences with $r(v_i, F)$ start to execute. It is not hard to prove that $\mathcal{W}$ can be sequentially ordered by the schedule S described above, which implies $\mathcal{W}$ is weakly consistent.

Now assume $\mathcal{W}$ is weakly consistent. The synchronization $\#_\alpha$ before $r(x, 0)$ ensures $w(x, 1)$ can not occur until all reads $r(c_i, T)$ are done. Since $w(x, 1)$ is the only source for $r(x, 1)$, then one and only one write to v_i occurs before $w(x, 1)$ and before each $r(c_i, T)$. Then the first value written to each v_i constitutes the satisfying assignment. Take the clause $C_i = l_p \vee l_q \vee l_r$ for example. Because $w(d_i, T)$ is the only source for each read $r(v_i, T)$, and in order for $w(d_i, T)$ to happen, at least one of the three reads $r(l_p, \mathcal{B}(l_p))$, $r(l_q, \mathcal{B}(l_q))$, $r(l_r, \mathcal{B}(l_r))$ will have to occur before $w(d_i, T)$ and thus before $r(d_i, T)$. Assume it is $r(l_q, \mathcal{B}(l_q))$ where $l_q = \neg v_k$, then assign v_k to be `false` (i.e., by executing $w(v_k, F)$) will make the clause C_i true. Other clauses can be analyzed along the same line.

A.2 TWC problem with restricted processes

Theorem A.1. *The TWC problem restricted to three processes is* NP*-complete.*

Proof. We show that the instance of TWC in Figure 5.1 is weakly consistent exactly when its corresponding one-in-three 3SAT instance $\mathcal{F}$ has a satisfying assignment. The sufficiency of this theorem can be easily validated. Assume $\mathcal{F}$ has a satisfying assignment, and C_1 is satisfied with $(v_{x_1}, v_{y_1}, v_{z_1}) = (T, F, F)$. During the (C$_1$) stage of the construction, process P_1 executes first, which is followed by P_2 and then P_3. This forms a sequential schedule for the operations during this stage, therefore (C$_1$) is weakly consistent. As each stage ends with restoring all variables $(v_{x_1}, v_{y_1}, v_{z_1})$ to their original values. Then the stage (C$_2$), as well as (C$_3$, ..., C$_m$), can be analyzed along the same line.

For the necessity, assume $\mathcal{W}$ is weakly consistent. An immediate inference is that every (sequence) segment on the stage C_i is weakly consistent. At the start of each stage C_i, only one variable is true, while the other two be false. This is because only in this way, there is a schedule by which all of the three processes can proceed. The variable values on each stage will be restored to their initial values when the stage ends. Thus, we construct an assignment $\mathcal{A}$ as one in which the value of the literals be the value at the end of stage (INIT). Obviously, this assignment $\mathcal{A}$ is the desired assignment for the instance $\mathcal{F}$.

A.3 TWC-read problem restricted to two processes

Lemma A.2. *The TWC-read problem restricted to two processes is in* P.

Proof. We now describe a polynomial algorithm that determines whether a given instance $\mathcal{H}$ is weakly consistent. We assume: (a) every read in $\mathcal{H}$ has a responsible write; and (b) no read in $\mathcal{H}$ returns 0 (the initial value of variables). These assumptions are only made to simplify the proof and have no influence over the validity of this lemma. The algorithm starts with constructing a cover function τ as: $\forall o \in \mathbb{O}, \tau(o) = \lfloor o \rfloor$.

Denote by f the read-mapping that maps each read to the identifier of the responsible write, we then upgrade the cover function to a witness OCF with the following procedures:

AddWrite For each access o, if a read r belongs to $\tau(o)$, then we upgrade $\tau(o)$ to be $\tau(o) \cup \{f(r)\}$.
ObsClosed For any two accesses u and v,
- if one of them is a synchronization, and $u \in \tau(v)$, then $\tau(v) = \tau(v) \cup \tau(u)$,
- if u is a synchronization, and there is another access $w \in \tau(v)$ such that $u \in \tau(w)$, then $\tau(v) = \tau(v) \cup \tau(u)$.

A typical OCF returns for each access a self-contained set, in which every read can find its corresponding write. This property is achieved with the procedure **AddWrite**. The procedure **ObsClosed** meets the constraints imposed by OCF as in Definition 4. Our algorithm repeats the above procedures until τ is fixed, which finishes in $O(n^4)$ ($O(n)$ for **AddWrite** and $O(n^3)$ for **ObsClosed** for n accesses in $\mathcal{H}$). Once an OCF is generated, the next step is to check whether the weak order is preserved, that is verifying: For any two accesses, u and v, if $u \in \tau(v)$, then v can not happen before u w.r.t. the weak order. This checking process can be done in $O(n^2)$.

A failure of passing the check suggests that no OCF is available to satisfy Definition 5, that is the instance $\mathcal{H}$ is not in TWC-read. However, even if the check is passed, it is still possible that $\mathcal{H}$ is not weakly consistent, since the OCF can not guarantee every $\tau(o)$ will be reasonably explained. The algorithm continues with checking if there is a set $\tau(o)$ involves one of the following anomalous forms:

ReadStaleUpdate A read r observes two write w_1, w_2 accessing the same location, the responsible write is w_1 but w_1 happens before w_2 w.r.t. the weak order.
LateWrite An access w is the responsible write for r, but r happens before w w.r.t. the weak order.

ReadStaleUpdate can be verified within $O(n^3)$ for each read, therefore the whole time required is $O(n^4)$. The form **LateWrite** cam be verified within $O(n^3)$. An OCF is anomalous forms free implies every $\tau(o)$ can be *unfolded* into a sequence residing in $S_{\mathrm{r/w}}$. Lemma A.3 validates this fact: Every $\tau(o)$ is anomalous forms free iff $\mathcal{H}$ is in TWC-read, a result, combined with the above analysis, supplies us the algorithm that can solve the two processes TWC-read problem in polynomial time.

Given $\mathcal{H}$, we denote by $\Upsilon : \tau \to \{0,1\}$ a function such that for any $\tau \in \tau$, $\Upsilon(\tau) = 1 \Leftrightarrow \forall o \in \mathbb{O}, \exists \sigma \in \mathcal{H}|_{\tau(o) \cap \mathbb{D}} \cap S_{\mathrm{r/w}}$.

Lemma A.3. *Let τ_b be the OCF generated from the aforementioned procedures, then there exists an OCF τ such that $\Upsilon(\tau)$ iff $\Upsilon(\tau_b)$.*

Proof. ($\Leftarrow$). This direction is trivial by letting $\tau = \tau_b$.

($\Rightarrow$). From the construction of τ_b, it is easy to observe that every OCF τ is a super-function of τ_b in the way that $\forall o \in \mathbb{O}, \tau_b(o) \subseteq \tau(o)$. We now prove $\Upsilon(\tau) \Rightarrow \Upsilon(\tau_b)$. For an access o, we use σ to denote a sequence such that $\sigma \in \mathcal{H}|_{\mathcal{T}(o)\cap\mathbb{D}} \cap S_{\mathrm{r/w}}$, we write $\sigma_b = \sigma|_{\tau_b(o)}$ as the sub sequence of σ restricted to $\tau_b(o)$. The sub-sequence σ_b respects the weak order since σ does, then $\sigma_b \in \mathcal{H}|_{\tau_b(o)\cap\mathbb{D}}$. Since every read r in σ_b has a responsible write that happens before r, then σ_b is LateWrite free. Also, as σ is ReadStaleUpdate free, then σ_b is ReadStaleUpdate free (otherwise there is at least one read in σ that returns a stale value), which means $\sigma_b \in S_{\mathrm{r/w}}$. Therefore, $\sigma_b \in \mathcal{H}|_{\tau_b(o)\cap\mathbb{D}} \cap S_{\mathrm{r/w}}$, that is $\Upsilon(\tau_b) = 1$.

A.4 The Decidability of Verifying Weak Consistency

Theorem A.2. *Given an implementation $\mathcal{I}$ as a regular language, checking whether all executions of $\mathcal{I}$ are weakly consistent with respect to $S_{\mathrm{r/w}}$ is undecidable.*

Proof. Let $\Sigma = \{a, b\}$ and $\mathcal{P}$ be an instance of PCP. The pair of sequences that defines $\mathcal{P}$ is: $U = \{u_1, ..., u_n\}, V = \{v_1, ..., v_n\}$. For a letter $v_i \in V$, we use $\widetilde{v_i}$ to denote b if $v_i = a$, and a if $v_i = b$. We use the term $\zeta(\xi(x, d))$ to denote a sequence of operations $w(x, 0) \cdot \xi(x, d) \cdot w(x_S, \#)$ for a $x \in \mathbb{X}$ and $d \neq 0$, where x_S is a synchronization variable, ξ is either a read or a write operation, and $\mathbb{X}$ is a finite set of variables. The synchronization used here ensures that at each time a process executes $\zeta(\xi(x, d))$, all local updates of the current history are propagated outward and updates occurred elsewhere are brought in, in this way all processes are synchronized.

Let $\langle u, v\rangle$ be a valid witness of $\mathcal{P}$. This history $\mathcal{H}_{uv}$ is constructed in a way as illustrated in Figure A.4. For the sake of convenience, we will use $[e]$ to refer to the name of the event that follows it in the history. We call a read operation $r(x, d)$ reads from a write $w(x, d)$, when $w(x, d)$ is the value provider. The integers m, n, k and $|v|$ used in the history have the relation: $m+n+k = 2|v|$, where $|v|$ is the length of sequence the v. We now prove the following equivalence: (1) $\mathcal{H}_{uv}$ is weakly consistent; (2) $u \neq v$.

($1 \Rightarrow 2$) Assume $\mathcal{H}_{uv}$ is weakly consistent. Assume by contradiction that $u = v$. Thus, $|u| = |v|$ and for all $i \in \{1, ..., |v|\}, u_i = v_i$. We prove that there is at least one $[v_{i_0}]$ that has to read from $[u_{i_0}]$. Then, since $u_i \neq \widetilde{v_i}$ for all $i \in \{1, .., |v|\}$, $[u_{i_0}]$ can not be the value provider for $[v_{i_0}]$.

In order to return 1 for the read operation $[r_T]$, one of the three write operations $[w_T^i](i = 1, 2, 3)$ has to happen before $[r_T]$. Assume the value provider is $[w_T^1]$, then $[h_1]$ is overwritten by $[v_0]$ since each synchronization operation works as a fence. Now there are $2|v| + 3$ read operations on object o in process p_v while only $2|v| + 2$ writes are available on the writer processes and helper processes. This implies there is at least one read operation $[v_i]$ will have to read from $[u_j]$ on p_u for some $j \in \{1, ..., |u|\}$. We now show that it can only read from $[u_i]$ but not from $[u_j]$ for $i \neq j$.

We start by proving that the read operation in $[t_i]$ can only read value from $[s_i]$. To accomplish this, we need to prove that $[t_i]$ can not read value from $[z_j]$ in process p_{aux_2}. Observe that $[w_q]$ (and all the subsequent events) has to wait until $[v_{|v|+3}]$ is globally performed, otherwise the value written on x_q will be visible to p_v and event $[r_q]$ is impossible. Assume by contradiction that there is a $[t_i]$ reading value from $[z_j]$ for some j, then $[r_q]$ (and all preceding operations on p_v) will certainly happen before $[t_i]$, $[r_p]$ and $[w_p]$. By the definition of $\zeta(r(x_{\mathcal{A}}, 1))$, each read operation involved in $[f_i]$ needs a unique value provider because all previous write operations on $x_{\mathcal{A}}$ is overwritten by a leading write in $\zeta(\cdot)$, then the write operations that are available for read operations in $[f_i]$ are insufficient. Therefore the read operation in $[t_i]$ can not read value from $[z_j]$. Notice that p_u contains $|u| + 1$ read operations on $x_{\mathcal{B}}$, and p_v contains $|u| + 1$ write operations on $x_{\mathcal{B}}$. Each read operation in $[t_i]$ exactly corresponds to $[s_i]$.

Similarly, we can prove that the read operation in $[f_i]$ can only read value from $[e_i]$. Now, assume that $[v_i]$ uses a write $[u_j]$ with $i < j$, which means its following $[f_i]$ will have to use $[e_k]$ for $j \leq k$, which is impossible based on the previously demonstrated property. Similarly, if $[v_i]$ uses a write $[u_j]$ with $i > j$, then $[t_j]$ need to read value from $[s_k]$ for $k > i$, which is also impossible.

This concludes the proof that there is at least one read operation $[v_{i_0}]$ that has to read from $[u_{i_0}]$, but not from $[u_j]$ for $i_0 \neq j$.

($2 \Rightarrow 1$) To prove that the converse is true, we need to consider three cases: $|u| = |v|$, $|u| < |v|$ and $|u| > |v|$, and prove that $\mathcal{H}_{uv}$ is weakly consistent under all cases. And in order to show that $\mathcal{H}_{uv}$ is weakly consistent, we need to guarantee that every read operation has a reasonable explanation for the value it returns.

Case 1: $|u| = |v|$ but $u_{i_0} \neq v_{i_0}$ for some $i_0 \in \{1, .., |v|\}$. Because $u_{i_0} \neq v_{i_0}$ implies $u_{i_0} = \widetilde{v_{i_0}}$, the write operation $[u_{i_0}]$ can provide value to $[v_{i_0}]$. Also, $[w_T^3]$ provides value to $[r_T]$, $[h_1]$ to $[v_{|v|+1}]$, $[h_2]$ to $[v_{|v|+2}]$, and $[c_k]$ to $[v_{|v|+3}]$. The remaining read operation $[v_i], [w_i]$ ($[v_{i_0}]$ excluded) can be explained by executing $\{[a_1], ..., [a_m], [b_1], ..., [b_n], [c_1], ..., [c_{k-1}]\}$ sequentially. Finally, the operations $[e_i]$ provides value to the read operation in $[f_i]$, and $[s_i]$ to the read operation in $[t_i]$ for $i \in \{1, ..., |v| + 1\}$.

Case 2: $|u| < |v|$. The write operation $[u_{|u|+1}]$ can provide value to $[v_{|u|+1}]$. This case is different with the above one in the way that the process p_u itself can not provide enough write operations on $x_{\mathcal{A}}$ that are needed by the read operations in $[f_i]$ on process p_v. However, this can be assisted by process p_{aux_1}. It causes no trouble for the read operation $[r_p]$ because the process p_{H_1} can wait until all operations on p_u are performed.

The value returned by each read operation can be explained in the following way:

- $[e_i]$ provides value to the read operation in $[f_i]$ for $i \in \{1, ..., |u|\}$
- $[s_i]$ provides value to $[t_i]$ for $i \in \{1, ..., |u|\}$
- $[y_i]$ provides value to $[t_{i+|u|}]$ for $i \in \{1, ..., |v| - |u|\}$
- $[a_i]$ provides value to to $[v_i]$ for $i \in \{1, ..., |v|\} \wedge \widetilde{v_i} = a$
- $[b_i]$ provides value to to $[v_i]$ for $i \in \{1, ..., |v|\} \wedge \widetilde{v_i} = b$
- $[u_{|u|+1}]$ provides value to $[v_{|u|+1}]$
- $[c_i]$ provides value to to $[w_i]$ for $i \in \{1, ..., |v|, |v| + 3\} \backslash \{|u| + 1\}$
- $[h_1]$ provides value to $[v_{|v|+1}]$
- $[h_2]$ provides value to $[v_{|v|+2}]$

Case 3: $|u| > |v|$. Similar to the previous case except that $[u_{|u|+1}]$ is used to provide values to $[v_{|v|+3}]$.

Now we have established the equivalence between (non) weak consistency of a history and the validity of its corresponding witness. The last thing is to describe how to define $\mathcal{I}$ as a regular language, such that $\mathcal{I}$ produces, for each witness $\langle u, v \rangle$, an execution whose history is $\mathcal{H}_{uv}$.

For an execution of $\mathcal{I}$, the helper processes execute their operations, which is followed by the auxiliary processes executing their first operations. The process p_v then chooses non-deterministically a pair $\langle u_i, v_i \rangle$ from the instance $\mathcal{P}$. Then p_v and p_u execute operations based on the symbols on v_i and u_i, the writer and auxiliary processes behave accordingly. For example, if $\langle u_i, v_i \rangle = \langle a, b \rangle$, then p_u do operations $[u_i, e_i, t_i]$, p_v do operations $[v_j, w_j, f_j, s_j]$, the writer process p_a does a write operation $w(0, a)$. The auxiliary process do an operation $[y_i], [z_i]$, respectively. This procedure may repeat for an arbitrary times, depending on the structures of u and v.

At the time that the above procedure terminates, p_u, p_v and the auxiliary processes will execute their rest operations, as detailed in Figure A.4. By the structure of operations $[t_i]$ and $[f_i]$, all processes synchronize after each choice of witness $\langle u_i, v_i \rangle$(actually, they synchronize after every symbol), $\mathcal{I}$ thus can be described by a regular language. $\square$

p_a :	p_b :	p_c :
(writer process a)	(writer process b)	(writer process c)
$[a_1] : w(o, a)$	$[b_1] : w(o, b)$	$[c_1] : w(o, Z)$
$[a_2] : w(o, a)$	$[b_2] : w(o, b)$	$[c_2] : w(o, Z)$
$\dots$	$\dots$	$\dots$
$[a_m] : w(o, a)$	$[b_n] : w(o, b)$	$[c_k] : w(o, Z)$

p_{H_1} :	p_{H_2} :	p_{H_3} :
(helper process 1)	(helper process 2)	(helper process 3)
$[h_1] : w(o, a)$	$[h_2] : w(o, b)$	$[h_3] : w(o, Z)$
$[w_T^1] : w(T, 1)$	$[w_T^2] : w(T, 1)$	$[w_T^3] : w(T, 1)$

p_{aux_1} :	p_u :	p_v :	p_{aux_2} :
$[w_p] : w(x_p, 1)$	$[u_1] : w(o, u_1)$	$[r_T] : \zeta(r(T, 1))$	$[w_q] : w(x_q, 1)$
$[y_1] : w(x_{\mathcal{A}}, 1)$	$[e_1] : w(x_{\mathcal{A}}, 1)$	$[v_0] : w(o, Z)$	$[z_1] : w(x_{\mathcal{B}}, 1)$
$\dots$	$[t_1] : \zeta(r(x_{\mathcal{B}}, 1))$		$\dots$
$[y_{\lvert v\rvert+1}] : w(x_{\mathcal{A}}, 1)$		$[v_1] : r(o, \widetilde{v_1})$	$[z_{\lvert u\rvert+1}] : w(x_{\mathcal{B}}, 1)$
	$[u_2] : w(o, u_2)$	$[w_1] : r(o, Z)$	
	$[e_2] : w(x_{\mathcal{A}}, 1)$	$[f_1] : \zeta(r(x_{\mathcal{A}}, 1))$	
	$[t_2] : \zeta(r(x_{\mathcal{B}}, 1))$	$[s_1] : w(x_{\mathcal{B}}, 1)$	
	$\dots$	$[v_2] : r(o, \widetilde{v_2})$	
		$[w_2] : r(o, Z)$	
	$[u_{\lvert u\rvert}] : w(o, u_{\lvert u\rvert})$	$[f_2] : \zeta(r(x_{\mathcal{A}}, 1))$	
	$[e_{\lvert u\rvert}] : w(x_{\mathcal{A}}, 1)$	$[s_2] : w(x_{\mathcal{B}}, 1)$	
	$[t_{\lvert u\rvert}] : \zeta(r(x_{\mathcal{B}}, 1))$	$\dots$	
	$[u_{\lvert u\rvert+1}] : w(o, Z)$		
	$[e_{\lvert u\rvert+1}] : w(x_{\mathcal{A}}, 1)$	$[v_{\lvert v\rvert}] : r(o, \widetilde{v_{\lvert v\rvert}})$	
	$[t_{\lvert u\rvert+1}] : \zeta(r(x_{\mathcal{B}}, 1))$	$[w_{\lvert v\rvert}] : r(o, Z)$	
		$[f_{\lvert v\rvert}] : \zeta(r(x_{\mathcal{A}}, 1))$	
	$[r_p] : r(x_p, 0)$	$[s_{\lvert v\rvert}] : w(x_{\mathcal{B}}, 1)$	
		$[v_{\lvert v\rvert+1}] : r(o, a)$	
		$[v_{\lvert v\rvert+2}] : r(o, b)$	
		$[f_{\lvert v\rvert+1}] : \zeta(r(x_{\mathcal{A}}, 1))$	
		$[s_{\lvert v\rvert+1}] : w(x_{\mathcal{B}}, 1)$	
		$[v_{\lvert v\rvert+3}] : r(o, Z)$	
		$[r_q] : r(x_q, 0)$	

Fig. A.1. The history $\mathcal{H}_{uv}$ that corresponds to each pair of witness $\langle u, v\rangle$.