A Correctness Condition for High-Performance Multiprocessors*

HAGITattiya
ROY Friedman
Department of Computer Science
The Technion
Haifa 32000, Israel

February 21, 1994

*Partially supported by Technion V.P.R.—B. and G. Greenberg Research Fund (Ottawa).
1Part of this work was performed while this author was visiting DEC Cambridge Research Laboratory.
Abstract

*Hybrid consistency*, a consistency condition for shared memory multiprocessors, attempts to capture the guarantees provided by contemporary high-performance architectures. It combines the expressiveness of *strong* consistency conditions (e.g., sequential consistency, linearizability) and the efficiency of *weak* consistency conditions (e.g., Pipelined RAM, causal memory). Memory access operations are classified as either *strong* or *weak*. A global ordering of strong operations at different processes is guaranteed, but there is very little guarantee on the ordering of weak operations at different processes, except for what is implied by their interleaving with the strong operations. A formal and precise definition of this condition is given and an efficient algorithm for providing hybrid consistency on distributed memory machines is presented. The response time of the algorithm is proven to be within a constant multiplicative factor of the optimal time bounds.

To motivate hybrid consistency, it is shown that weakly consistent memories do not support *non-cooperative* (in particular, non-centralized) algorithms for mutual exclusion.
1 Introduction

Shared memory is an attractive paradigm for communication among computing entities because it is familiar from the uniprocessor case, it is more high level than message passing, and many of the classical solutions for synchronization problems were developed for shared memory. The fundamental problem is how to provide programmers with a useful model of logically shared data that can be accessed atomically, without sacrificing performance. The model must specify how the data can be accessed and what guarantees are provided about the results.

To enhance performance (e.g., response time), many implementations maintain multiple copies of the same logical piece of shared data (caching). Also, multiple application programs must be able to execute concurrently. More complications arise because at some level, each access to shared data has duration in time, from its start to its end; it is not instantaneous. A consistency mechanism guarantees that operations will appear to occur in some ordering that is consistent with some condition. Much research has addressed the issue of consistency for various system types and levels of abstraction. A major issue is which consistency condition should be supported: Which conditions can be implemented efficiently, which conditions can be used conveniently, and which conditions support faster programs.

Until recently, theoretical research on this subject addressed strong consistency conditions like sequential consistency and linearizability [12, 19, 20, 22, 25, 35, 37, 38, 41, 44, 46]. These conditions guarantee that operations appear to be executed atomically, in some sequential order that is consistent with the order seen at individual processes. Sequential consistency and linearizability provide a clean and easy semantics for the execution of operations; programming using these conditions is (relatively) easy.

Unfortunately, supporting either sequential consistency or linearizability has a non-negligible cost (cf. [39, 10]), i.e., these consistency conditions cannot be implemented efficiently. A way around this cost is to define conditions which provide weaker guarantees on the ordering of operations, and can be efficiently implemented. These conditions can be roughly classified into two categories. The first category includes weak conditions (e.g., [6, 7, 34, 36, 39]), that provide very little guarantee on the relative ordering of events at different processes. These conditions admit very efficient implementations; however, we will show that they are too weak to support conventional methods for concurrent programming. The second category includes hybrid conditions (e.g., [1, 18, 26, 29, 33, 44]), that distinguish between two types of operations—strong and weak. Strong operations appear to be executed atomically, in some sequential order that is consistent with the order seen at individual processes. The only guarantees provided for weak operations are those implied by their interleaving with strong operations. As we show in this paper, it is possible to implement hybrid consistency in such a way that weak operations are extremely fast, without degrading the response time for strong operations. Thus, a programmer can use the fast weak operations most of the time and resort to the slower strong operations only when global coordination is required.

These conditions are similar in flavor to the notion of serializability from database theory [13, 42]; however, serializability applies to transactions which aggregate many operations.
In this paper, we make a step towards a theoretical study of weak and hybrid shared memory consistency, in an attempt to further our understanding of the issues involved in selecting and implementing a memory consistency condition.

Our first major contribution is a formal and precise definition of hybrid consistency. Operations are classified as either strong or weak. Very informally, hybrid consistency guarantees two properties:

1. Strong operations appear to be executed in some sequential order.
2. If two operations are invoked by the same process and one of them is strong, then they appear to be executed in the order they were invoked.

In particular, the second property guarantees that a strong operation appear to be executed after any operation (weak or strong) invoked before it by the same process, and before any operation (weak or strong) invoked after it by the same process. The definition applies to any collection of objects for which a sequential specification is provided (cf. [35]). Our definition is very high-level and abstract—it describes the way operations appear to the programmer, not the way they are implemented. This results in a relatively simple definition and allows us to optimize its implementation. We believe that hybrid consistency supports common concurrent programming techniques: In Section 4, we show how Peterson’s simple mutual exclusion algorithm ([43]) can be modified to exploit hybrid consistency. We discuss the performance benefits achieved by using hybrid consistency, and illustrate how the formal definition is used when arguing about the correctness of algorithms. The correctness proof of this algorithm is quite similar to the proof of Peterson’s original algorithm that assumes sequential consistency. This may indicate that the complexity of arguing about programs assuming hybrid consistency is not much greater than the complexity of arguing about them assuming sequential consistency.

Our second major contribution is an algorithm that implements hybrid consistency on distributed memory machines [12, 18, 38, 44]; our algorithm supports read/write objects. The algorithm is completely asynchronous. Weak operations are executed instantaneously, while the response time for strong operations is linear in the network delay. The algorithm is based on an atomic broadcast mechanism for sending messages. This mechanism is used to guarantee that all strong operations are executed by all processes in the same order. All weak writes to the same location are executed in the same order by all processes. Combined with the fact that a strong operation does not start until all previous operations have been executed by all processes, and does not terminate until it was executed by all other processes, this guarantees the second property of hybrid consistency. By adapting proof techniques from [10, 39], we show that the response time of our algorithm is within a constant factor of the optimum.

Weak consistency admits very efficient implementations, e.g. [39], and one may wonder why we need hybrid consistency at all. Indeed, Lipton and Sandberg present solutions to several coordination problems that rely only on weak consistency, the most fundamental of which is a mutual exclusion algorithm [39]. However, this mutual exclusion algorithm is centralized and relies on a specific process moderating the entrance to the critical section. This is considered
bad practice, since the moderating process is required to participate in the algorithm, whether or not it wishes to enter the critical section. We show that this is inherent; hence, in our opinion, weak consistency does not provide adequate support for concurrent programming.

Specifically, our third major contribution shows that all solutions for the mutual exclusion problem based on weak consistency must be cooperative—certain processes have to participate in the algorithm, whether they wish to enter the critical section or not. This holds even for slightly stronger consistency conditions such as causal memory [6, 7] and processor consistency [34]. Centralized algorithms are cooperative since the process moderating the entry to the critical section must participate in the algorithm, even if it never wishes to enter the critical section. In contrast, all classical solutions to the mutual exclusion problem based on sequential consistency are not cooperative (cf. [24, 43, 45]).

Many memory consistency conditions have been suggested recently. To evaluate them on a common scale we need to apply a wide spectrum of criteria. In this paper, we take one such criterion, called compositionality and apply it to weak consistency and hybrid consistency. Compositionality captures the possibility of composing independent implementations of the same consistency condition to obtain an implementation of a larger set of objects. Thus, it is clear why compositionality is a desired characteristic for a memory consistency condition. We show that weak consistency is compositional, while hybrid consistency is not compositional. This implies that compositionality is not a monotone property: hybrid consistency is not compositional although both linearizability, which is stronger than hybrid consistency, and weak consistency, which is weaker than hybrid consistency, are compositional.

The rest of the paper is organized as follows: Section 2 includes our basic definitions and some notation. Section 3 contains the definition of hybrid consistency and a discussion of related definitions. Section 4 provides an example of programming with hybrid consistency. In Section 5, we present an algorithm for providing hybrid consistency, together with its correctness proof and performance analysis. Section 6 contains lower bounds on the response time of strong operations under hybrid consistency. Section 7 includes a definition of weak consistency, and a proof of the impossibility of non-cooperative solutions to the mutual exclusion problem, using weak consistency. In Section 8, we check the compositionality of weak consistency and hybrid consistency. We conclude, in Section 9, with a discussion of our results and directions for further research.

2 The System

We consider a collection of application programs running concurrently and communicating via virtual shared memory, which consists of a collection of objects. We assume a system

---

\[\text{footnote}{\text{2} \text{Our impossibility result holds for a specific variation of processor consistency. A slightly stronger interpretation for which the impossibility result does not hold was suggested in [5]. See Section 7.1 for detailed discussion of processor consistency.}}\]

\[\text{footnote}{\text{3} \text{originally defined and called locality by Herlihy and Wing ([35])}}\]
consisting of a collection of nodes $P$ connected via a communication network; each application program runs on a different node. The shared memory abstraction is implemented by a memory consistency system (mcs), which uses local memory and some protocol executed by the mcs processes (one at each node). A correctness condition is defined at the interface between the application programs (written by the user) and the mcs processes (supplied by the system). Thus, the mcs must provide the proper semantics when the values of the responses to calls are considered, throughout the network. An illustration of the system architecture is given in Figure 1.

The following events may occur at the mcs process on node $i$:

1. **Call events**: The initiation of operations by the application program according to their specification. For example, in the case of read/write objects, the call events are $\text{Read}_i(X)$ and $\text{Write}_i(X, v)$ for all objects $X$ and values $v$.

2. **Response events**: The response of the mcs to operations initiated by the application program, according to their specification. For example, in the case of read/write objects, the response events are $\text{Return}_i(X, v)$ and $\text{Ack}_i(X)$ for all objects $X$ and values $v$.

3. **Message receive events**: $\text{receive}(i, m, j)$ for all messages $m$ and mcs processes $i$ and $j$: the mcs process on node $i$ receives message $m$ from the mcs process on node $j$.

4. **Message send events**: $\text{send}(i, m, j)$ for all messages $m$ and mcs processes $p_i$ and $p_j$: the mcs process on node $i$ sends message $m$ to the mcs process on node $j$.

The call and message-receive events are interrupt events.
An *mcs process* (or simply *process*) is an automaton with a (possibly infinite) set of states, including an initial state, and a transition function. Each interrupt event causes an application of the transition function. The transition function is a function from states and interrupt events to states, sets of response events and sets of message-send events. That is, the transition function takes as input the current state and an interrupt event, and produces a new state, a set of response events for the application process, and a set of messages to be sent. A *step* of a process \( p_i \) is a tuple \( (s, i, s', R, M) \), where \( s \) and \( s' \) are states, \( i \) is an interrupt event, \( R \) is a set of response events, \( M \) is a set of message-send events, and \( s', R \) and \( M \) are the result of \( p_i \)'s transition function acting on \( s \) and \( i \). A *history* of a process \( p_i \) is a mapping \( h \) from \( \mathbb{R} \) (real time) to finite (possibly empty) sequences of steps such that

1. for each real time \( t \), there is only a finite number of times \( t' < t \) such that the corresponding sequence of steps \( h(t') \) is nonempty (thus the concatenation of all the sequences in real-time order is a sequence);
2. the old state in the first step is \( p_i \)'s initial state; and
3. the old state of each subsequent step is the new state of the previous step.

An *execution* of an mcs is a set of histories, one for each process in \( P \), in which there is a one-to-one correspondence from the messages received by \( p_i \) from \( p_j \) onto the messages sent by \( p_j \) to \( p_i \), for any processes \( p_i \) and \( p_j \). An *infinite execution* is an execution in which every history is infinite. We use the message correspondence to define the *delay* of any message in an execution to be the real time of receipt minus the real time of sending. (The network is not explicitly modeled, although the constraints on executions, defined below, imply that the network reliably delivers all messages sent.)

An execution \( \sigma \) is *admissible* if the following conditions hold:

1. For every \( i \) and \( j \), every message in \( \sigma \) from \( p_i \) to \( p_j \) has its delay in the range \([0, d]\), for a fixed nonnegative integer \( d \). (This is a restriction on the network.)
2. For every \( i \), at most one call at \( p_i \) is pending at a time in \( \sigma \). (This is a restriction on the application program.)

From now on, we assume all executions are admissible. A history \( h' \) is a *partial history* of a history \( h \) if there exists a time \( t_0 \) such that for all times \( t < t_0 \), \( h'(t) = h(t) \), \( h'(t_0) \) is a prefix of \( h(t_0) \) and for all times \( t > t_0 \), \( h'(t) \) is empty. A *partial execution* is a collection \( E \) of partial histories, one for every process, in which there is a one-to-one correspondence between the messages received by \( p_i \) from \( p_j \) in \( E \) and the messages sent by \( p_j \) to \( p_i \) in \( E \), for every two processes \( p_i \) and \( p_j \). (Note that it is possible that some messages are not received in the partial execution.) Given a partial history \( h \) of a process \( p_i \), let \( t_0 \) be the largest time for which \( h(t_0) \) is not empty; the *last state* of \( p_i \) in \( h \) is the local state of \( p_i \) at the end of the sequence of steps \( h(t_0) \). A *global state* is the collection of last states of all processes in a partial execution. Given
the global state of a partial execution $E$, a previous global state is the global state of a partial execution $E'$ such that every partial history in $E'$ is a partial history of a history in $E$. From now on, we use the word state for global states, unless specifically said otherwise.

Each pair of a call event and a subsequent matching response event forms an operation. The call event marks the start of the operation, while the response event marks the end of that operation. An operation $op$ is invoked when the application process issues the appropriate call for $op$; it terminates when the mcs process issues the appropriate response for $op$. An operation is pending if it is invoked and is not terminated. Note that since we assumed that all executions are admissible, there can be at most one pending operation per process, i.e., an application program does not invoke a new operation before the previous one has terminated. Given a particular mcs, an object $x$ implemented by it, and an operation type $op$ on $x$, we denote by $|op(x)|$ the maximum time taken from the invocation to the termination of an operation of type $op$ on $x$ in any execution. We denote by $|op|$ the maximum of $|op(x)|$ over all objects $x$ implemented by the mcs.

Every object is assumed to have a sequential specification (cf. [35]) defining a set of operations, which are ordered pairs of call and response events, and a set of operation sequences, which are the allowable sequences of operations on that object. For example, in the case of a read/write object, the ordered pair of events $\{\text{Read}_i(x), \text{Return}_i(x,v)\}$ forms an operation for any $p_i$, $x$, and $v$, as does $\{\text{Write}_i(x,v), \text{Ack}_i(x)\}$. The set of operation sequences consists of all sequences in which every read operation returns the value of the latest preceding write operation (the usual read/write semantics).

A sequence $\tau$ of operations for a collection of processes and objects is legal if, for every object $x$, the restriction of $\tau$ to operations of $x$ is in the sequential specification of $x$.

Given an execution $\sigma$, let $ops(\sigma)$ be the sequence of call and response events appearing in $\sigma$ in real-time order, breaking ties by ordering all events of the same process in the order they appear in that process and then using process ids. An execution $\sigma$ induces a partial order, $\Rightarrow$, on the operations that appear in $\sigma$; $op_1 \Rightarrow op_2$ if the response event of $op_1$ appears in $ops(\sigma)$ before the call event of $op_2$.

Given an execution $\sigma$, a sequence of operations $\tau$ is a serialization of $\sigma$ if it is a permutation of $ops(\sigma)$. A serialization $\tau$ of $\sigma$ is a linearization if it extends $\Rightarrow$; that is, if $op_1 \Rightarrow op_2$ then $op_1 \Rightarrow^* op_2$.

Let $\tau$ be an execution or a sequence of operations. Denote by $\tau \mid j$ the restriction of $\tau$ to operations invoked by $p_j$; similarly, denote by $\tau \mid x$ the restriction of $\tau$ to operations on object $x$.

For completeness, we repeat the definitions of sequential consistency and linearizability, using the terminology we have just defined:

**Definition 2.1 (Sequential consistency)** An execution $\sigma$ is sequentially consistent if there exists a legal serialization $\tau$ of $\sigma$, such that for each process $p_j$, $\sigma \mid j = \tau \mid j$. 


Definition 2.2 (Linearizability) An execution $\sigma$ is linearizable if there exists a legal linearization $\tau$ of $\sigma$, such that for each process $p_j$, $\sigma \upharpoonright j = \tau \upharpoonright j$.

It is possible to mark some operations as strong; all other operations are called weak. In the case of read/write objects this means that it is possible to use strong reads and strong writes. We denote the call events for strong operations by $\text{SWrite}(x, v)$ and $\text{SRead}(x)$ and the respective response events by $\text{SAck}(x)$ and $\text{SReturn}(x, v)$.

We now introduce some notation we use in the rest of the paper. By $op_i$ we denote an operation invoked by $p_i$ (weak or strong), and by $sop_i$ we denote a strong operation invoked by process $p_i$. We use superscripts, e.g., $op^1_i, op^2_i, \ldots$, to distinguish between operations invoked by the same process. We sometimes use a shorthand notation for read and write operations and denote by $r_i(x, v)$ a weak read operation invoked by process $p_i$ returning $v$ from $x$; we denote by $w_i(x, v)$ a weak write operation invoked by process $p_i$ writing $v$ to $x$. Similarly, $sr_i(x, v)$ is a strong read operation invoked by process $p_i$ returning $v$ from $x$; $sw_i(x, v)$ is a strong write operation invoked by process $p_i$ writing $v$ to $x$.

3 Definition of Hybrid Consistency

The following definition requires that (a) strong operations appear to occur in the same order at all processes, and (b) if two operations are invoked by the same process and one of them is strong, then they appear to occur at all processes in the order they were invoked.

Definition 3.1 (Hybrid consistency based on linearizability) An execution $\sigma$ is hybrid if there exists a linearization $\rho$ of $\sigma$ such that for each process $p_i$, there exists a legal sequence of operations $\tau_j$ with the following properties:

1. $\tau_j$ is a permutation of $\text{ops}(\sigma)$.
2. if $op^1_i \xrightarrow{\rho} op^2_i$ and at least one of $op^1_i$ and $op^2_i$ is strong, then $op^1_i \xrightarrow{\tau_j} op^2_i$, for any $i$.
3. if $op^1_i \xrightarrow{\rho} op^1_i$ and $op^1_i$ and $op^2_i$ are strong, then $op^1_i \xrightarrow{\tau_j} op^2_i$.
4. $\tau_j \upharpoonright j = \sigma \upharpoonright j$.

Definition 3.1 defines the view $\tau_j$ for each process $p_j$. This view must include all the memory operations in the execution (this is the first requirement). Any two operations invoked by the same process $p_i$, where at least one of them is strong, must be viewed in their order of invocation in $p_i$ by $p_j$ (this is the second requirement). The views of all processes must agree on the order of the strong operations (this is the third requirement). Finally, process $p_j$ must view its own operations in their order of invocation (this is the fourth requirement).
The definition requires the existence of a linearization $\rho$ of $\sigma$. However, $\rho$ is used only to force a global order on the strong operations. Therefore, we can require the existence of a linearization only of the strong operations in $\sigma$, obtaining an equivalent definition.

When restricted to strong operations, this definition is equivalent to linearizability. However, following the results of [10], we might be interested in examining hybrid models in which strong operations are only sequentially consistent. This leads to the following variant of hybrid consistency:

**Definition 3.2 (Hybrid consistency based on sequential consistency)** An execution $\sigma$ is hybrid if there exists a serialization $\rho$ of $\sigma$ such that for each process $p_i$, there exists a legal sequence of operations $\tau_i$ with the following properties:

1. $\tau_i$ is a permutation of $\text{ops}(\sigma)$.
2. if $op^1_i \xrightarrow{\tau} op^2_i$ and at least one of $op^1_i$ and $op^2_i$ is strong, then $op^1_i \xrightarrow{\tau^\prime} op^2_i$, for any $i$.
3. if $op^1_i \xrightarrow{\tau} op^2_i$ and $op^1_i$ and $op^2_i$ are strong, then $op^1_i \xrightarrow{\tau^\prime} op^2_i$.
4. $\tau_i \| j = \sigma \| j$.

The only difference between the two definitions is that the second definition requires that $\rho$ is a serialization of $\sigma$ rather than a linearization.

### 3.1 Related Definitions

We now present a detailed comparison of hybrid consistency, as defined above, with other definitions that distinguish between two kinds of operations.

Most previously known definitions of hybrid consistency are given by specifying, at some degree of formality, how the mcs handles operations, i.e., how the hardware should behave. In contrast, our definition specifies how the operations appear to the application program, i.e., how the hardware should appear to the programmer. In particular, our definition is given at the interface between the application program and the mcs, while most previous definitions of hybrid consistency are given at the interface between the mcs and the network.

Although from the architect’s point of view it is easier to work with a definition that is given at the interface between the mcs and the network, a definition at the interface between the application program and the mcs is more convenient for programming. It is more difficult to write programs or argue about their correctness when the exact behavior of the hardware has to be considered. Also, programs that are written with respect to an abstract consistency condition are more portable; they can run correctly on different implementations of the consistency condition, regardless of the optimizations that are used by the hardware. Our
work complements previous work by presenting an abstract and formal definition at a more comfortable interface, even if this disallows some hardware optimizations.

We now turn to a more detailed comparison of our definition of hybrid consistency with related definitions.

### 3.1.1 Dubois, Scheurich and Briggs

The most related condition is *weak ordering*, defined by Dubois, Scheurich and Briggs [26]. Variables are classified as either regular or synchronization, and it is required that:

1. Accesses to global synchronization variables are strongly ordered.
2. No access to a synchronizing variable is issued by a process before all its previous global data accesses have been executed by all the processes.
3. No access to global data is issued by a process before its previous access to a synchronizing variables has been executed by all the processes.

This definition is given in terms of restrictions on the implementation (i.e., at the interface between the mcs processes), rather than on the way the mcs should appear to the programmer (i.e., at the interface between the application programs and the mcs).

This definition rules out implementations which obey the intended semantics of the definition and give better performance. For example, the implementation given in [26] could be improved by pipelining weak operations with strong operations. This improvement seems to obey the intended semantics of the definition, but violates the definition itself. To alleviate this problem and be able to compare this definition with ours, the following definition is specified at the interface between the mcs and the application program, and (we hope) provides the semantics intended by the definition in [26]:

**Definition 3.3** Let $X$ be the set of synchronizing variables in the system. An execution $\sigma$ is object based hybrid if there exists a serialization $\rho$ of $\sigma$ such that for each process $p_j$, there exists a legal sequence $\tau_j$ of operations such that the following holds:

1. $\tau_j$ is a permutation of $\text{ops}(\sigma)$.
2. if $op^1_j \in (\sigma \mid X)$ or $op^2_j \in (\sigma \mid X)$ and $op^1_j \rightarrow op^2_j$, then $op^1_j \rightarrow_{\tau_j} op^2_j$.
3. if $op^1_j, op^2_j \in (\sigma \mid X)$ and $op^1_j \rightarrow op^2_j$, then $op^1_j \rightarrow_{\tau_j} op^2_j$.
4. $\tau_j \mid j = \sigma \mid j$. 

9
This definition is a special case of our definition of hybrid consistency in which strong operations are restricted to special synchronization objects. In contrast, our definition allows the use of strong and weak operations on the same object. This property is exploited in the mutual exclusion algorithm given below in Section 4. In this algorithm, the reads from synchronization objects are weak. This results in a reduced number of messages in the system.

3.1.2 Adve and Hill

A different approach splits the responsibility for memory consistency between the software and the hardware. In this approach, the hardware must exhibit some known predefined behavior only for software that obeys certain requirements. For example, the following definition of weak ordering was given by Adve and Hill [1]:

"Hardware is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all software that obeys the synchronization model."

Following this definition, they give a formal definition of a synchronization model DRF0. In this synchronization model, an application program is required to use synchronization operations in any place where a data-race is possible.

Using the same approach, a generalization of DRF0 that supports further classification of synchronization operations into releases and acquires is presented in [2, 3]. This synchronization model is called DRF1. Again, the application program is required to use synchronization operations to prevent data races.

The approach taken by Adve and Hill divides the responsibility for correctness between the mcs and the application program: The mcs supports a consistency condition very similar to hybrid consistency, and the application program is required to obey a certain synchronization model, in this case DRF0 (or DRF1). Then it is guaranteed that the programs will run as if the mcs was sequentially consistent. In this approach, the mcs provides a guarantee about the results of memory access operations only for programs that obey DRF0 (or DRF1). There is absolutely no guarantee (or at least no crisp guarantee that does not rely on understanding a specific implementation) for programs that violate DRF0 (or DRF1).

3.1.3 Release Consistency

Release consistency, defined by Gharachorloo et al. ([29]), is supported in the Stanford DASH multiprocessor. The definition given in [29] assumes three types of operations: ordinary, sync and nsync. Each type is further divided to reads and writes. The following list of sufficient conditions is then given:

1. Uniprocessor data dependencies are respected.
2. All writes to the same location are serialized in some order and are executed in that order with respect to any process.

3. Before a sync write access is allowed to execute with respect to any other process, all previous ordinary write accesses by the same process must be executed.

4. Sync and nsync accesses are kept sequentially consistent with respect to one another.

Besides being at the interface between the mcs and the network, this definition is given as a list of sufficient conditions on the hardware. These conditions are based on the notion of “an operation being executed with respect to a process”, whose meaning depends on the specific implementation which was kept in mind when the definition was given.

Formal definitions of release consistency are given in [32, 33]. These definitions are quite complex, and make reasoning about programs directly with release consistency a very difficult task. To overcome this problem, a synchronization model called PL-programs is defined; this model is similar to DRF1. It is then guaranteed that any program obeying this synchronization model will run as if the mcs was sequentially consistent. This approach, as the one taken by Adve and Hill, gives little support to the design of programs that do not obey specific synchronization models, e.g., our modification of the classical solution to mutual exclusion (presented in the next section). Note that sometimes it is easier to prove the correctness of a program with respect to a consistency condition from scratch than to prove that the program obeys some synchronization model like DRF0, DRF1 or PL.

A major conceptual difference between release consistency and hybrid consistency is the further classification of strong operations into release and acquire. We remark that it is not difficult to modify our definition to capture this further distinction (see [9]). However, recent experiments [28, 49] suggest that, at least in certain cases, the benefits of this further classification are somewhat limited when compared to hybrid consistency. In addition, release consistency does not require that each process views the weak reads of other processes. Hybrid consistency, on the other hand, requires that the view of every process includes all operations of any other process, in particular, the weak reads. We believe that this additional requirement makes the programmer’s model more comprehensive and therefore more convenient to use, although it may prohibit some otherwise possible hardware optimizations.

4 Programming with Hybrid Consistency

Hybrid consistency supports a very simple method of programming: use only strong operations and “think sequential consistency” (or linearity, as the case may be). Clearly, this method is prone to the same performance penalties as sequential consistency. A contrasting method is to ignore strong operations altogether and use only weak operations, similar to [39].

\footnote{The definition in [32] is a generalization of the definition in [33] that allows pipelining of memory operations.}
Figure 2: Mutual exclusion using hybrid consistency – code for process $p_i$, $i = 0, 1$.

Unfortunately, this method turns out to be “too weak”, in the sense that is discussed below, in Section 7.

A better programming method is to use weak operations most of the time and strong operations only in those places where global ordering is required. In this section, we give an example of a mutual exclusion algorithm based on hybrid consistency, and prove its correctness. This example demonstrates that programming with hybrid consistency can be simple and efficient. It also gives some intuition on when strong operations should be used, and how to prove correctness relying on hybrid consistency.

Formally, an algorithm for mutual exclusion consists of four disjoint sections—entry, critical, exit and remainder (cf. [45]). In the entry section, a process tries to gain access to the critical section; the exit section is executed by each process upon leaving the critical section; the remainder section is the rest of the code. A mutual exclusion algorithm should guarantee:

**mutual exclusion**: no two processes are inside the critical section at the same time, and

**deadlock freedom**: in every infinite execution, if there is a process in the entry section, then eventually there is some process in the critical section.

Figure 2 presents a solution for two processes which is fair; it is a simple modification of Peterson’s algorithm [43]. The algorithm can be extended to $n$ processes along the lines of Peterson’s algorithm. Lines 1-3 are the entry section; Line 4 is the exit section. In the code we use the notation $r(x) = v$ to denote a weak read of $x$ returning the value $v$ and $sw(x, v)$ to denote a strong write of $v$ to $x$, for any read/write object $x$ and value $v$.

**Lemma 4.1** The algorithm in Figure 2 guarantees mutual exclusion under hybrid consistency.

**Proof:** Assume, by way of contradiction, that there is a hybrid execution $\sigma$ in which $p_0$ and $p_1$ execute the critical section together. Denote by $sw_0(need[0],true)$ and $sw_0(turn,1)$ the last strong writes to $need[0]$ and $turn$ executed by $p_0$ before entering the critical section, and by
\(sw_1(\text{need}[1], \text{true})\) and \(sw_1(\text{turn}, 0)\) the last strong writes to \(\text{need}[1]\) and \(\text{turn}\) executed by \(p_1\) before entering the critical section. Remember that \(\sigma\) is a hybrid execution and consider a serialization \(\rho\) of \(\sigma\) as guaranteed by the definition of hybrid consistency. Without loss of generality, \(sw_0(\text{turn}, 1)\) appears before \(sw_1(\text{turn}, 0)\) in \(\rho\). In particular, \(sw_0(\text{need}[0], \text{true})\) is ordered before \(sw_1(\text{turn}, 0)\) in \(\rho\). Since \(p_1\) entered its critical section, it follows that \(p_1\)'s last \(r_1(\text{turn})\) before entering returned 1. However, by assumption, \(sw_0(\text{turn}, 1) \xrightarrow{\tau_0} sw_1(\text{turn}, 0)\). By the definition of \(\tau_1\) it follows that \(sw_0(\text{turn}, 1) \xrightarrow{\tau_1} sw_1(\text{turn}, 0) \xrightarrow{\tau_1} r_1(\text{turn}, 1)\). Thus, \(\tau_1\) is not legal, since \(r_1(\text{turn})\) should have returned 0.

**Lemma 4.2** The algorithm in Figure 2 is free of deadlock under hybrid consistency.

**Proof:** Assume, by way of contradiction, that there is a hybrid execution \(\sigma\) in which there is a deadlock. A deadlock occurs if \(p_0\) and \(p_1\) keep executing Line 3; that is, \(p_0\) continually reads \(r_0(\text{need}[1], \text{true})\) and \(r_0(\text{turn}, 1)\) in Line 3 and \(p_1\) continually reads \(r_1(\text{need}[0], \text{true})\) and \(r_1(\text{turn}, 0)\).

Since \(\sigma\) is a hybrid execution, there exists a serialization \(\rho\) of \(\sigma\). Without loss of generality, we may assume that

\[
sw_0(\text{turn}, 1) \xrightarrow{\rho} sw_1(\text{turn}, 0) \quad \text{and} \quad sw_1(\text{need}[1], \text{true}) \xrightarrow{\rho} sw_1(\text{turn}, 0),
\]

where \(sw_0(\text{turn}, 1)\) is the strong write by \(p_0\) that precedes the infinite loop in Line 3, and \(sw_1(\text{need}[1], \text{true})\) and \(sw_1(\text{turn}, 0)\) are defined similarly for \(p_1\). Consider the legal serialization \(\tau_0\) of \(\sigma\), as guaranteed by hybrid consistency. Since \(p_0\) keeps executing Line 3, it follows that \(p_0\) continually executes \(r_0(\text{need}[1], \text{true})\) and \(r_0(\text{turn}, 1)\). Let \(\tau_0(\text{need}[1], \text{true})\) and \(\tau_0(\text{turn}, 1)\) be some pair of these reads. Since \(\tau_0\) is legal,

\[
sw_0(\text{turn}, 1) \xrightarrow{\tau_0} sw_1(\text{turn}, 0) \xrightarrow{\tau_0} r_0(\text{need}[1], \text{true}) \quad \text{and} \quad
sw_1(\text{need}[1], \text{true}) \xrightarrow{\tau_0} sw_1(\text{turn}, 0) \xrightarrow{\tau_0} r_0(\text{need}[1], \text{true}).
\]

By the definition of \(\tau_0\), \(\tau_0(\text{need}[1], \text{true})\) \(\xrightarrow{\tau_0} r_0(\text{turn}, 1)\). Thus, \(sw_1(\text{turn}, 0) \xrightarrow{\tau_0} r_0(\text{turn}, 1)\), and \(\tau_0\) is not legal. A contradiction.

A mutual exclusion algorithm is sometimes required to guarantee:

**starvation freedom:** in every infinite execution, every process which executes the entry section is eventually granted permission to enter the critical section.

**Lemma 4.3** The algorithm in Figure 2 is starvation free under hybrid consistency.
Proof: Assume, by way of contradiction, that there exists an infinite hybrid execution $\sigma$ of the algorithm in which one process, say $p_0$, is trying to enter the critical section, but is never granted permission. Let $sw_0(turn, 1)$ be the last write of $p_0$ to $turn$. By Lemma 4.2, $p_1$ must enter the critical section an infinite number of times. Let $\rho$ be a serialization of all operations in $\sigma$, as guaranteed in the definition of hybrid consistency. Every operation in $\sigma$ must appear in $\rho$ after a finite number of operations. Thus, there exists an operation $sw_1(turn, 0)$ such that $sw_0(turn, 1) \xrightarrow{\rho} sw_1(turn, 0)$. Consider a legal serialization $\tau_1$ of $\sigma$, as guaranteed by hybrid consistency. Under the assumption that $p_1$ enters the critical section infinitely often, $\tau_1$ must include the following subsequence:

$$sw_0(turn, 1) \xrightarrow{\tau_1} sw_1(turn, 0) \xrightarrow{\tau_1} r_1(turn, 1).$$

A contradiction to the assumption that $\tau_1$ is legal. 

The algorithm uses only weak reads in the entry section; hence, when running on an efficient implementation of hybrid consistency (like the one presented in this paper), the entry section is executed faster and with less message overhead.5

Note that if the first operation inside the critical section is a weak operation, then we might not have logical mutual exclusion. That is, the operations of the entry and exit sections are ordered correctly, but weak operations issued by $p_0$ inside the critical section may be ordered in the view of $p_1$ before the last (weak read) operation of the corresponding entry section of $p_0$. Similarly, weak operations issued by $p_1$ inside the critical section may be ordered in the view of $p_0$ before the last (weak read) operation of the corresponding entry section of $p_1$. This need not happen in every implementation of hybrid consistency, but could happen in some of them. A detailed discussion about this problem and some solutions can be found in [9].

5 An Algorithm for Providing Hybrid Consistency

In this section we present an efficient algorithm for providing hybrid consistency that supports read/write objects, accessible by either strong or weak operations.6 The algorithm allows weak operations to be executed instantaneously, while keeping the response time for strong operations proportional to the network delay. Being able to execute weak operations fast is vital for the usefulness of this algorithm, since it is the major advantage hybrid consistency has over sequential consistency. By allowing each process to hold a local copy of the entire memory, we can guarantee instantaneous execution of weak operations, provided that write operations will eventually update all copies.

Note that by efficient implementations we refer to implementations that execute weak operations faster than the network delay. By the lower bounds proved in [10] and in Section 6, strong operations cannot be implemented faster than the network delay.

6 Algorithms for providing hybrid consistency for other objects, using somewhat different techniques, appear in [27].
The algorithm we describe is completely asynchronous. This means that every process may run at a different rate and that the delay of the network is unknown.

The algorithm uses an atomic broadcast mechanism to send messages. Atomic broadcast is becoming quite common as an underlying tool in the development of distributed systems [14, 16, 40, 48]. Our decision to use atomic broadcast follows this trend and is justified by the immediate benefits that this mechanism provides. An atomic broadcast mechanism guarantees that all messages are delivered at all processes in the same order. Thus, it creates a “global logical time”, which simplifies the code of the algorithm and its proof of correctness. Furthermore, using an atomic broadcast mechanism is more modular than implementing the communication protocol directly. Several atomic broadcast algorithms with various degrees of fault tolerance and efficiency have been developed, e.g., [8, 15, 17, 21]. Thus, by employing different atomic broadcast algorithms, one may balance the degree of fault tolerance, message complexity and time complexity, to fit the desired goals of the implementation.

In the algorithm, weak operations are executed on the local copy of the memory. Strong operations and weak write operations broadcast a message to all processes; an operation is executed by each process when the corresponding message is delivered at the process. Thus, the order of execution should reflect the order in which messages are delivered by the atomic broadcast mechanism.

Unfortunately, there is a problem since weak operations must return immediately and a weak read that immediately follows a weak write by the same process must return the value written by that write. To handle this problem, we modify the basic idea such that weak writes that are not invoked by the local process are executed under the following conditions: A weak write that is not concurrent with any other (weak or strong) write is always executed when its corresponding message is delivered. If there are two concurrent weak writes, than the first one is executed by every process and the second one is ignored by every process.

The detailed description of the algorithm and its pseudo-code in Section 5.1 are followed by a correctness proof in Section 5.2 and a complexity analysis in Section 5.3. Later, in Section 5.4, we discuss several efficient implementations of atomic broadcast and present one such possible implementation.

5.1 The Algorithm

We assume a system of \( n \) processes, each capable of holding a local copy of the entire memory. The processes are connected by some interconnection network. We assume an atomic broadcast mechanism that supports two primitives: \( \text{abc} \) and \( \text{send}_{\text{abc}} \). The first primitive, \( \text{abc} \), broadcasts a message to all processes atomically. A process may use the second primitive, \( \text{send}_{\text{abc}} \), to send a message to a single process, when the interleaving of this message with

---

\(^7\)Concurrency is with respect to the global logical time. In other words, two weak writes are concurrent if the same number of messages have been delivered at the invoking processes before they were invoked.
other messages sent by it, using either \texttt{abc} or \texttt{send\_abc}, should reflect their relative order of invocation.

More precisely, let $MA$ be the set of messages sent using \texttt{abc} and denote by $MA_i$ the sequence of messages in $MA$ that are delivered at process $p_i$, according to the order they are delivered at $p_i$. Then the following hold: (a) for any $i$ and $j$, $MA_i = MA_j$, and (b) for any two messages $m_1^i, m_2^i \in MA$ such that both $m_1^i$ and $m_2^i$ are sent by $p_i$ and $m_1^i$ is sent before $m_2^i$, then $m_1^i$ is ordered in $MA_i$ before $m_2^i$. Moreover, let $MP_{ij}$ be the sequence of messages sent from process $p_i$ to process $p_j$ using either \texttt{abc} or \texttt{send\_abc}, ordered according to the order they were sent, and let $m \in MP_{ij}$. Then $m$ is delivered at $p_j$ before any other message ordered after it in $MP_{ij}$ and after any other message ordered before it in $MP_{ij}$.

We use \texttt{send\_abc} instead of a regular \texttt{send} to send acknowledgements since the correctness of the algorithm depends on the fact that at each process, the delivery order of all messages reflects the order in which they were sent. We use \texttt{send\_abc} instead of \texttt{abc} because in some implementations of atomic broadcast, e.g., the one presented in Section 5.4, it results in a lower message complexity.

Every process maintains a local copy of the entire memory $mem$, an array of integers $last\_mess$, an array of ids $last\_id$, an integer counter $received$ and another object $val$, as described below.

The counter $received$ is used to count the messages delivered by the atomic broadcast. It serves as a logical time stamp and its value is added to messages. It is initialized to 0 and incremented each time a message that was sent with \texttt{abc} is delivered.

For each object, $last\_mess$ and $last\_id$ uniquely identify the last operation that updated the object. That is, whenever a (weak or strong) write updates an object, $last\_mess$ is assigned with the current value of the variable $received$ and $last\_id$ is assigned with the id of the process that invoked the write.

Finally, the variable $val$ is used to temporarily store a value that a pending strong read should return, as described later.

Weak operations are executed on the local copy of the memory and return immediately. A weak write causes an update message to be broadcast to all processes. This message contains the object to be updated, the new value for the object and a logical timestamp which is the value of $received$ when the operation is invoked. Whenever an update message is received, an ack message is sent back to the initiator of the operation, using \texttt{send\_abc}. Following this, processes that did not invoke the write, execute it on the local copy of the object if either (a) the previous operation that updated the same object was invoked by the same process (held in the variable $last\_id$), or (b) the value of $last\_mess$ is smaller than the value of $received$ that is included in the update message of the weak write. These conditions guarantee that all writes to the same location are executed by all processes in the same order. Note that this order corresponds to the order implied by the logical timestamp of the writes. In other words, it corresponds to the value of the variable $received$ at the time of their invocation. Ties among all writes with the same logical timestamp are broken by ordering the first delivered weak write
after all other weak writes with the same logical timestamp. The variable \textit{last\_id} is required to preserve the order of writes by the same process, since two consecutive weak writes by the same process may have the same logical timestamp.

Whenever a strong operation is invoked, if the operation is a strong read and the last local operation was a weak read, then a dummy message is broadcast to all processes. Every process that receives a dummy message, returns an \textit{ack} message to the initiator using \text{send\_abc}. Next, the process waits until all \textit{ack} messages of the previous operations and dummy messages return. Then, a \textbf{strong\_write} or a \textbf{strong\_read} message is broadcast to every process. The operation is executed by every process whenever the appropriate message is delivered. A strong write is executed by updating the local copy of the object. A strong read is executed at the invoking process by copying the value of the local copy of the object to \textit{val}. This value is returned when the operation returns. A strong read is executed by other processes by doing nothing. After executing a strong operation, an \textit{ack} message is sent back to the initiator of the operation. A strong operation does not return until all \textit{ack} messages have been received.

A dummy message is broadcast between the invocation of a weak read and the invocation of a strong read to guarantee that the invocation of a strong read is separated by at least a certain amount of “logical time” from the invocation of the last previous weak operation by the same process. This is due to the reordering of strong reads, which is done in the proof of correctness, with respect to the delivery events of their corresponding \textbf{strong\_read} messages. Note that there is no need to send a dummy message after a weak write since weak writes already broadcasts an update message; this already guarantees that there is enough “logical time” between the invocation of the weak write and the strong operation.

The precise code appears in Figure 3 and Figure 4.

5.2 Proof of Correctness

Given an execution $\sigma$, we explicitly show how to construct the set of sequences $T = \{\tau_j\}_{j=1}^n$, as required in the definition of hybrid consistency.

Very informally, the construction goes as follows. For each process $p_j$, we first build the sequence $\tau_j$, consisting of all (weak and strong) write operations, all local weak read operations, and all delivery events. The operations are ordered according to the order they occurred in $p_j$. The delivery events serve as markers which are useful for the rest of the proof. They are ordered according to the order they occurred, such that each delivery event is ordered after the corresponding operation. We show that $T' = \{\tau'_j\}_{j=1}^n$ obeys all the requirements in the definition of hybrid consistency, except for including all operations in $\text{ops}(\sigma)$. Following this, we insert the strong reads into $T'$, creating a new set of sequences $T'' = \{\tau''_j\}_{j=1}^n$, such that the strong reads are legal. $T''$ obeys all the requirements in the definition of hybrid consistency, except for including all operations in $\text{ops}(\sigma)$. Next, for each sequence $\tau''_j$, we insert all weak reads by other processes, creating a new set of sequences $T''' = \{\tau'''_j\}_{j=1}^n$ such that all reads in each $\tau'''_j$ are legal $T'''$ obeys all the requirements in the definition of hybrid consistency except
The state of each process $p_i$ consists of the following components:

- $mem$: copy of every object, initially equal to its initial value
- $last_mess$: array of integers with one entry for each object, all initially 0
- $last_id$: array of ids with one entry for each object, all initially 0
- $received$: integer, initially 0 (serial number of the last message delivered)
- $missing_acks$: integer, initially 0 (number of acknowledges that are expected)
- $val$: an object

Read($x$):
generate Return($x, mem[x]$)

Write($x, v$):

$\text{mem}[x] := v$
$\text{abc} < \text{update}, x, v, received >$
$missing_acks := missing_acks + n$
generate Ack($x$)

SRead($x$):
if the last previous operation by $p_i$ was a weak read then
$\text{abc} < \text{dummy} >$
$missing_acks := missing_acks + n$
endif
wait until $missing_acks = 0$ /* This line is not atomic */
$\text{abc} < \text{strong-read}, x >$
$missing_acks := missing_acks + n$

SWrite($x, v$):
wait until $missing_acks = 0$ /* This line is not atomic */
$\text{abc} < \text{strong-write}, x, v >$
$missing_acks := missing_acks + n$

Figure 3: The algorithm — code for process $p_i$. 
received < update, x, v, s > from p_j:

send_abc < ack, j >
received := received + 1
if ((s \geq last_mess[x]) or (last_id[x] = j)) then
  last_mess[x] := received
  last_id[x] := j
  if i \neq j then
    mem[x] := v
  endif
endif

received < dummy > from p_j:

send_abc < ack, j >
received := received + 1

received < strong-write, x, v > from p_j:

send_abc < ack, j >
received := received + 1
last_mess[x] := received
mem[x] := v
last_id[x] := j

received < strong-read, x > from p_j:

send_abc < ack, j >
received := received + 1
if (j = i) then val := mem[x] endif

received < ack, i > from p_j:
missing_acks := missing_acks - 1
if (missing_acks = 0) and there is a pending strong operation then
  if the strong operation is a write then generate SAck(x)
  else generate SReturn(x, val)
  endif
endif

Figure 4: The algorithm — code for handling message receive events at process p_i.
for having the delivery events. Finally, we construct the desired set of sequences $T$ by removing all delivery events from $T''$.

We now turn to the formal proof, and start with some definitions and notations. For every operation that causes a message to be broadcast, i.e., weak write, strong write or strong read, we denote by $del_j(op)$ the delivery event of the corresponding message (update, strong-write or strong-read, accordingly) in $p_j$. Given a strong read $sr_i$, we denote by $rel(sr_i)$ the last (strong-read, strong-write, update or dummy) message broadcast by $p_i$ before the strong-read message of $sr_i$. We slightly abuse the notation and denote by $del_j(rel(sr_i))$ the delivery event of $rel(sr_i)$ in $p_j$.

A read operation $r_i(x,v)$ reads from a write operation $w_k(x,v)$ if $w_k(x,v)$ is the last write to $x$ that updates $p_i$'s copy of $x$ before $r_i(x,v)$ reads $x$. (If $r_i(x,v)$ is strong, then $r_i(x,v)$ reads from $x$ when it copies the value of $x$ into val.) A read $r_i(x,v)$ is legal in a sequence of operations $\sigma$ if there exists a write $w_k(x,v)$ such that $w_k(x,v) \rightarrow r_i(x,v)$ and there does not exist another write $w_l(x,u)$, $u \neq v$, such that $w_k(x,v) \rightarrow w_l(x,u) \rightarrow r_i(x,v)$; otherwise, the read is illegal in $\sigma$.

Given a read operation $r_i(x,v)$ that reads from a write $w_k(x,v)$ and a sequence of operations $\sigma$, a write $w_l(x,u)$ is an obliterating write for $r_i(x,v)$ in $\sigma$ if $w_k(x,v) \rightarrow w_l(x,u)$.

A conditional execution interval of a weak write $w_k$ is the interval of events in $\sigma$ between the invocation of $w_k$ and the delivery of the update message of $w_k$ by the atomic broadcast at $p_k$. A weak write $w_k(x,v)$ is overwritten by another (weak or strong) write $w_j(x,u)$ if $w_j(x,u)$ is executed in $p_k$ during the conditional execution interval of $w_k(x,v)$. A potential execution interval of a weak write $w_k(x,v)$ is the largest interval inside the conditional execution interval of $w_k$, starting with the invocation of $w_k(x,v)$, in which no other write $w_j(x,u)$, $j \neq k$, is executed by $p_k$. Note that a weak write updates the local copy of the object and returns immediately. The potential execution interval is the interval after a weak write updates the local copy of the object, but may still be overwritten. For example, let $w_i(x,v)$ be a weak write that is overwritten by another write $w_k(x,u)$. If we denote the invocation event of $w_i(x,v)$ by $inv(w_i(x,v))$, we get

\[
\text{conditional execution interval} \quad \overbrace{inv(w_i(x,v)) \ldots del(w_k(x,u)) \ldots del(w_i(x,v))} \quad \text{potential execution interval}
\]

Define the notion of influence as follows. A weak write $w_k(x,v)$ influences process $p_j$, $j \neq k$, if it updates the copy of $x$ in $p_j$. A weak write $w_k(x,v)$ influences process $p_k$ if it is not overwritten. A strong write influences every process. Intuitively, a weak write influences its invoking process if it is not overwritten, and influences another process if it is executed by that process. Strong writes are never overwritten and are executed by every process.

Next, define the notion of an operation being executed by a process as follows:
A strong operation $sop_k$ is executed by $p_j$ when the corresponding strong-write or strong-read message is delivered at $p_j$.

A weak operation $op_j$ is executed by $p_j$ when the appropriate call event occurs.

A weak write $w_k$ that influences $p_j$, $k \neq j$, is executed by $p_j$ when the corresponding update message is delivered at $p_j$.

Let *abcast order* be the order by which all messages sent with *abc* are delivered; note that this order is well-defined.

We now turn to the details of the proof; for the rest of this section, fix some execution $\sigma$ of the algorithm.

For each process $p_j$, create the sequence of operations and delivery events $\tau'_j$ as follows: Order all delivery events in $p_j$ according to the order they occur in $p_j$. Next, add all weak operations by $p_j$ according to their order of invocation in $p_j$ (with respect to themselves and to the delivery events in $p_j$). Next, add all weak writes that are invoked by other processes and are executed by $p_j$ and all strong writes (including the strong writes of $p_j$) immediately before the delivery of their corresponding strong-write or update message. Finally, add every weak write $w_k$ that is not executed by $p_j$ immediately before the last write to the same location that is executed in $p_j$ before $w_k$, breaking ties arbitrarily. Note that if $w_k$ is not executed, then the code of the algorithm guarantees the existence of this write. Note also that all delivery events are ordered in the same order in all sequences $\tau'_j$. That is, for every two operations $op_k^1$ and $op_k^2$, $del_j(op_k^1) \xrightarrow{\tau'_j} del_j(op_k^2)$ if and only if $del_i(op_k^1) \xrightarrow{\tau'_i} del_i(op_k^2)$, for every $i$ and $j$.

**Lemma 5.1** There exists a linearization $\rho$ of all the operations in $\sigma$ such that for every pair of strong operations $sop_k$ and $sop_l$ in $\tau'_j$, $sop_k \xrightarrow{\rho} sop_l$ if and only if $sop_k \xrightarrow{\tau'_j} sop_l$, for every $\tau'_j$.

**Proof:** The strong operations appear in every $\tau'_j$ in an order which is consistent with the abcast order. Since a strong operation does not return before all its *ack* messages return, the real time of its delivery is always between the real time of its invocation and the real time of its termination. Thus, the order in which all strong operations appear in every $\tau'_j$ is a linearization.

**Lemma 5.2** For every sequence $\tau'_j$, $\tau'_j|j = \sigma|j$.

**Proof:** Atomic broadcast delivers all messages sent by the same process in the order they were sent. Also, strong operations do not return until all *ack* messages are delivered. Since weak operations are ordered according to their invocation and strong operations are ordered according to their delivery, it follows that $\tau'_j|j = \sigma|j$, for every sequence $\tau'_j$. 

---

Technion - Computer Science Department - Technical Report CS0767 - 1993
Note that in each \( \tau'_j \), each (local weak) read \( r_j(x,v) \) is ordered after the last write to \( x \) that is executed by \( p_j \) before the invocation of \( r_j(x,v) \), with no other write to \( x \) in between them. Thus, we have:

**Lemma 5.3** For every sequence \( \tau'_j \), all the weak reads of \( p_j \) are legal in \( \tau'_j \).

The following lemma is crucial in the rest of the proof.

**Lemma 5.4** If a write influences some process, then it influences every process.

**Proof:** Recall that all messages are delivered to every process in the abcast order. Furthermore, the variable \( \text{received} \) is incremented each time a message that was sent using \( \text{abc} \) is delivered by the atomic broadcast. Thus, for every weak write, the value of \( \text{received} \) is the same when the corresponding \( \text{update} \) message is delivered in every process. Moreover, this value is different for different weak writes.

Assume, by way of contradiction, that there exists some write that influences some process but not all processes. Let \( w_k(x,v) \) be the first such write, i.e., the one for which the value of \( \text{received} \) during the delivery event of its \( \text{update} \) message is minimum. Assume that \( w_k(x,v) \) does not influence \( p_j \). Thus, \( w_k(x,v) \) is not executed by \( p_j \). The code of the algorithm for executing weak writes imply that there exists another write \( w_l(x,u) \) for which the following holds: (a) \( w_l(x,u) \) influences \( p_j \), (b) the \( \text{update} \) message of \( w_l(x,u) \) is delivered in \( p_j \) before the \( \text{update} \) message of \( w_k(x,v) \), and (c) the \( \text{update} \) message of \( w_l(x,u) \) is delivered in \( p_k \) after \( w_k(x,v) \) is invoked. (This is the condition which is checked whenever an \( \text{update} \) message is delivered.) Since \( w_k(x,v) \) is the first write that influences only some of the processes, \( w_l(x,u) \) influences \( p_k \) and therefore \( w_k(x,v) \) is overwritten.

Atomic broadcast delivers all messages to all processes in the same order. Since \( w_k(x,v) \) is the first write that influences only some of the processes, \( w_l(x,u) \) influences all processes and its \( \text{update} \) or \( \text{strong-write} \) message is delivered before the \( \text{update} \) message of \( w_k(x,v) \) to all processes. Thus, by the assumptions about \( w_l(x,u) \), none of the processes executes \( w_k(x,v) \). Hence, \( w_k(x,v) \) does not influence any process. This is a contradiction to the assumption that \( w_k(x,v) \) influences some process.

**Lemma 5.5** For each sequence of operations \( \tau'_j \) and every pair of operations \( op^1_k \) and \( op^2_k \) in \( \tau'_j \) such that either \( op^1_k \) or \( op^2_k \) is strong, \( op^1_k \xrightarrow{\tau'_j} op^2_k \) if and only if \( op^1_k \xrightarrow{\sigma} op^2_k \).

**Proof:** By Lemma 5.2, the claim holds if \( k = j \). Thus, for the rest of the proof, we may assume that \( k \neq j \).

Assume, by way of contradiction, that for some sequence \( \tau'_j \) and two operations \( op^1_k \) and \( op^2_k \) such that either \( op^1_k \) or \( op^2_k \) is strong, \( op^1_k \xrightarrow{\tau'_j} op^2_k \) but \( op^1_k \xrightarrow{\sigma} op^1_k \). We claim that \( op^1_k \) is
a weak write that does not influence \( p_i \) and \( op^j_k \) is strong. Strong operations and weak writes that influence \( p_i \) are executed according to the abcast order. This order is consistent with the order by which operations are invoked at each process. Therefore, the only operations that may not be ordered correctly in \( \tau^j_i \) with respect to the previous and next strong operations by the same process are weak writes that do not influence \( p_i \). Moreover, since strong operations are ordered immediately before their corresponding delivery events, and since weak writes are always ordered before their corresponding delivery event, a weak write that does not influence \( p_i \) may not be ordered after a later strong operation by the same process.

Assume, without loss of generality, that \( op^j_k \) writes to \( x \). Let \( w_i \) be the first write to \( x \) that influences \( p_i \) and is ordered after \( op^j_k \) in \( \tau^j_i \). The existence of such a write is guaranteed by the assumption that \( op^j_k \) does not influence \( p_i \) and the construction of \( \tau^j_i \); \( w_i \) is the write that caused \( op^j_k \) to be ordered before \( op^j_k \). The code of the algorithm also implies that \( w_i \) is delivered in \( p_i \) before \( op^j_k \).

If \( i \neq j \), or if \( i = j \) and \( op^j_k \) is delivered after the completion of the conditional execution interval of \( w_i \) in \( p_i \), then by Lemma 5.4, \( w_i \) influences every process. Thus, \( w_i \) is delivered in every process before \( op^j_k \). In particular, \( w_i \) is delivered in \( p_k \) before \( op^j_k \). Since \( op^j_k \) is a strong operation and since \( op^j_k \) is invoked after \( op^j_k \), \( w_i \) is delivered in \( p_k \) before \( op^j_k \) starts. Therefore, there exists another write \( w_i(x, w) \) that influences \( p_i \) and is delivered in \( p_i \) between \( w_i \) and \( op^j_k \). Thus, \( w_i(x, w) \) is ordered in \( \tau^j_i \) between \( w_i \) and \( op^j_k \). A contradiction to the assumption that \( w_i \) is the first write to \( x \) that influence \( p_i \) and is ordered after \( op^j_k \) in \( \tau^j_i \).

If \( i = j \) and \( op^j_k \) is delivered inside the conditional execution interval of \( w_i \), then the ack message of \( op^j_k \) is delivered in \( p_k \) after the update message of \( w_i \). By Lemma 5.4, \( w_i \) influences \( p_k \) and is delivered in \( p_k \) before \( op^j_k \) starts. Therefore, there exists another write \( w_i(x, v) \) that influences \( p_i \) and is delivered in \( p_i \) between \( w_i \) and \( op^j_k \). Thus, \( w_i(x, w) \) is ordered in \( \tau^j_i \) between \( w_i \) and \( op^j_k \). A contradiction to the assumption that \( w_i \) is the first write to \( x \) that influence \( p_i \) and is ordered after \( op^j_k \) in \( \tau^j_i \).

Thus, we have shown that the sequences \( \{ \tau^j_i \}^n_{i=1} \) obey all the requirements in the definition of hybrid consistency, except for including all the operations of \( ops(\sigma) \). To finish the proof, we must insert the missing operations, i.e., the missing reads, to each \( \tau^j_i \) without violating the requirements of the definition of hybrid consistency.

We start with the strong reads. For each \( j \), create \( \tau^\mu_j \) by inserting the strong reads into \( \tau^j_i \) according to the abcast order of their strong-read messages. Unlike strong writes, sometimes it is not possible to insert a strong read immediately before its corresponding delivery event. This is because the strong-read message can be delivered at another process inside a potential execution interval of a weak write to the same object. If this happens, then the last value written to the object is the value written by the local weak write, and not the value returned by the strong read. Therefore, in order to maintain the legality of the sequence, the strong read has to be inserted before the weak write. However, since hybrid consistency requires that all views of all processes agree on the order of strong operations, if a strong read \( sr_j(x, v) \) is inserted in one sequence before another strong operation \( sop_k \), then \( sr_j(x, v) \) must be inserted
before sop$_k$ in every sequence $\tau''_j$. On the other hand, if sop$_k$ is a strong read that is inserted long before its corresponding strong-read message, then inserting sr$_i(x,v)$ before sop$_k$ could cause sr$_i(x,v)$ to be ordered before a previous weak operation by $p_k$. In order to prevent such chain reactions, it is sometimes required to drag a previously inserted strong read sr$_k(y,u)$ before sr$_i(x,v)$ instead of inserting sr$_i(x,v)$ before sr$_k(y,u)$. For example, if we have

$$\tau_i'' = \ldots w_j(x,v) \ldots sop_i \ldots sr_j(y,u) \ldots del_j(sr_i(x,u)) \ldots del_j(w_j(x,v))$$

and

$$\tau''_i = \ldots sop_i \ldots sr_k(y,u) \ldots del_i(sr_i(x,u))$$

then sr$_i(x,v)$ must be inserted before $w_j(x,v)$ and sop$_i$ in $\tau''_i$. Moreover, sr$_i(x,v)$ must be inserted before sop$_k$ in $\tau''_i$, too. In addition if sr$_k(y,u)$ is a strong read that is dragged by the insertion of sr$_i(x,v)$, then sr$_k(y,u)$ must be ordered before sr$_i(x,v)$ and sop$_i$ both in $\tau''_i$ and $\tau''_i$.

To insert the strong reads, we introduce some definitions. For each strong read sr$_i(x,v)$, and for each process $p_j$, if the strong-read message of sr$_i(x,v)$ is delivered inside a potential execution interval of some weak write $w_j(x,u)$, then let $I_j(w_j(x,u),sr_i(x,v))$ be the set of strong operations already ordered in $\tau''_j$ between the invocation of $w_j(x,u)$ and the delivery of the strong-read message of sr$_i(x,v)$. Let $B_j(w_j(x,u),sr_i(x,v))$ be the set of strong reads $\{sr_q\}$ such that $sr_q \in I_j(w_j(x,u),sr_i(x,v))$ and the write $sr_q$ reads from is included in the potential execution interval of $w_j(x,u)$. Let $I_j(sr_i(x,v))$ be the union of $I_j(w_j(x,u),sr_i(x,v))$ over all such $w_j(x,u)$; let $B_j(sr_i(x,v))$ be the union of $B_j(w_j(x,u),sr_i(x,v))$, over all such $w_j(x,u)$. Let $B(sr_i(x,v)) = \bigcup_{j=1}^{n} B_j(sr_i(x,v))$ and $I(sr_i(x,v)) = \bigcup_{j=1}^{n} I_j(sr_i(x,v))$, and let $C(sr_i(x,v))$ be the set of strong reads in $I(sr_i(x,v)) \setminus B(sr_i(x,v))$ and $D(sr_i(x,v)) = I(sr_i(x,v)) \setminus C(sr_i(x,v))$.

Intuitively, $I(sr_i(x,v))$ is the set of strong operations that $sr_i(x,v)$ might be inserted before, although $del_j(sr_i(x,v))$ is ordered after them in some sequence $\tau''_j$. $D(sr_i(x,v))$ is the set of strong operations that $sr_i(x,v)$ is actually inserted before, although $del_j(sr_i(x,v))$ is ordered after them in some sequence $\tau''_j$. $C(sr_i(x,v))$ is the set of strong reads that are dragged by $sr_i(x,v)$ while $B(sr_i(x,v))$ is the set of strong reads that appear in $I(sr_i(x,v))$ and are not dragged by $sr_i(x,v)$. $B(sr_i(x,v))$ is merely used to define $C(sr_i(x,v))$. For example, if we have

$$\tau''_i = \ldots w_j(x,v), sw_q, sr_k(y,u), w_j(z,w), sr_l(z,w), del_l(sr_i(x,u)), del_j(w_j(x,v)), \ldots$$

and this is the only potential execution interval of a weak write to $x$ that includes the delivery event of $sr_i(x,v)$, then $I(sr_i(x,u)) = \{sw_q, sr_k(y,u), sr_l(z,w)\}$, $B(sr_i(x,u)) = \{sr_l(z,w)\}$, $C(sr_i(x,u)) = \{sr_k(y,u)\}$ and $D(sr_i(x,u)) = \{sw_q, sr_l(z,w)\}$.

Add sr$_i(x,v)$ to each $\tau''_j$ in the last possible place such that it will be ordered before every strong operation in $D(sr_i(x,v))$, before the delivery event of its strong-read message and before any obliterating write operation. For every strong read sr$_q$ in $C(sr_i(x,v))$, if sr$_i(x,v)$ is ordered before sr$_q$ in $\tau''_j$, then reorder sr$_q$ immediately before sr$_i(x,v)$ in $\tau''_j$. In this case, we say that sr$_q$ is dragged by sr$_i(x,v)$ in $\tau''_j$. If there is more than one strong read that is dragged by sr$_i(x,v)$, then break ties according to the abact order of their corresponding strong-read messages.
**Lemma 5.6** A strong read reads from a write that influences every process.

**Proof:** Let \( sr_i \) be a strong read that reads from some write \( w_k \). If \( w_k \) is strong then by definition, \( w_k \) influences every process. So, assume that \( w_k \) is weak. If \( k = i \), then since \( sr_i \) is a strong operation, it is not executed until the conditional execution interval of \( w_k \) is completed. Therefore, \( w_k \) is not overwritten and by Lemma 5.4, \( w_k \) influences every process. If \( k \neq i \), then \( w_k \) influences \( p_i \) and by Lemma 5.4, \( w_k \) influences every process.

**Lemma 5.7** Every strong read is legal in every sequence \( \tau''_j \).

**Proof:** We prove that every strong read is ordered in every sequence \( \tau''_j \) after the write it reads from. Since by definition, every strong read appears in every sequence \( \tau''_j \) before any obliterating write operation, this will imply that the strong read is legal.

Assume, by way of contradiction, that there exists a strong read that is ordered in some sequence \( \tau''_j \) before the write it reads from. Let \( sr_i(x,v) \) be the first strong read such that following the insertion of \( sr_i(x,v) \), there exists a strong read which is ordered in some sequence \( \tau''_j \) before the write it reads from. Thus, either \( sr_i(x,v) \) is ordered in \( \tau''_j \) before the write it reads from, or there exists another strong read \( sr_s(y,w) \) which is dragged in \( \tau''_j \) before the write it reads from.

Assume that \( sr_i(x,v) \) is ordered in \( \tau''_j \) before the write it reads from, and denote this write by \( w_k(x,v) \). In particular, \( sr_i(x,v) \) is ordered before \( del_j(w_k(x,v)) \) in \( \tau''_j \). Thus, \( del_j(sr_i(x,v)) \) is ordered in some sequence \( \tau''_i \) inside a potential execution interval of some weak write \( w_l(x,u) \) that includes a strong operation \( sop_q \) which is ordered in \( \tau''_l \) before \( del_j(w_k(x,v)) \). (If there is no such \( sop_q \), then there is no reason to order \( sr_i(x,v) \) in \( \tau''_j \) before the write it reads from.) By the assumption that \( sr_i(x,v) \) reads from \( w_k(x,v) \) and by Lemma 5.6, \( w_k(x,v) \) influences every process. Therefore, \( del_j(w_k(x,v)) \) is ordered in \( \tau''_j \) before the potential execution interval of \( w_l(x,u) \). Since we assumed that \( sop_q \) is ordered in \( \tau''_l \) before \( del_j(w_k(x,v)) \), and since \( del_j(sop_q) \) is ordered after \( del_j(w_k(x,v)) \), it follows that \( sop_q \) is a strong read and the write \( sop_q \) reads from is ordered inside the potential execution interval of \( w_l(x,u) \). Denote the write \( sop_q \) reads from by \( w_m \). By Lemma 5.6, both \( w_m \) and \( w_k(x,v) \) influence every process and therefore both \( w_m \) and \( w_k(x,v) \) are ordered immediately before their delivery events. Thus, \( w_m \) is ordered after \( w_k(x,v) \) in \( \tau''_j \). By the minimality of \( sr_i(x,v) \), \( sop_q \) is ordered after \( w_m \) in \( \tau''_j \), and therefore after \( w_k(x,v) \). A contradiction.

Assume that \( sr_s(y,w) \) is dragged by \( sr_i(x,v) \) before the write it reads from, and denote this write by \( w_k(y,w) \). In particular, \( sr_s(y,w) \) is dragged by \( sr_i(x,v) \) before \( del_j(sr_s(y,w)) \) in \( \tau''_j \). By the minimality of \( sr_i(x,v) \), \( sr_s(y,w) \) is ordered after \( w_k(y,w) \) in every sequence \( \tau''_j \) before the insertion of \( sr_i(x,v) \). Thus, \( del_j(sr_i(x,v)) \) and \( del_j(sr_s(y,w)) \) are ordered inside a potential execution interval of some weak write \( w_l(x,u) \) that includes a strong operation \( sop_q \) which is ordered in \( \tau''_l \) before \( del_j(w_k(y,w)) \). Since we assumed that \( sop_q \) is ordered in \( \tau''_l \) before \( del_j(w_k(y,w)) \), and since \( del_j(sop_q) \) is ordered after \( del_j(w_k(y,w)) \), it follows that \( sop_q \) is a strong read and the write \( sop_q \) reads from is ordered inside the potential execution interval
of $w_i(x, u)$. Denote the write $sop_q$ reads from by $w_m$. By Lemma 5.6, both $w_m$ and $w_k(x, v)$ influence every process and therefore both $w_m$ and $w_k(x, v)$ are ordered immediately before their delivery events. Thus, $w_m$ is ordered after $w_k(x, v)$ in $\tau''_j$. By the minimality of $sr_i(x, v)$, $sop_q$ is ordered after $w_m$ in $\tau''_j$, and therefore after $w_k(y, w)$. A contradiction.

Lemma 5.8 Every strong read $sr_i(x, v)$ is ordered after $del_j(rel(sr_i(x, v)))$ in every sequence $\tau''_j$.

Proof: Assume, by way of contradiction, that there exists a strong read $sr_i(x, v)$ that is ordered in some sequence $\tau''_j$ before $del_j(rel(sr_i(x, v)))$. Assume, without loss of generality, that $sr_i(x, v)$ is the first strong read such that following the insertion of $sr_i(x, v)$, there exists a strong read $sr_i(y, w)$ which is ordered in some sequence $\tau''_j$ before $del_j(rel(sr_i(y, w)))$. Thus, either $sr_i(x, v)$ is ordered in $\tau''_j$ before $del_j(rel(sr_i(x, v)))$, or there exists another strong read $sr_i(y, w)$ which is dragged in $\tau''_j$ before $del_j(rel(sr_i(y, w)))$.

Assume that $sr_i(x, v)$ is ordered in $\tau''_j$ before $del_j(rel(sr_i(x, v)))$. Thus, there exists a sequence $\tau''_j$ for which $del_j(rel(sr_i(x, v)))$ is ordered in $\tau''_j$ inside a potential execution interval of some weak write $w_i(x, u)$ that includes a strong operation $sop_q$ which is ordered in $\tau''_j$ before $del_j(rel(sr_i(x, v)))$. Since $sr_i(x, v)$ does not return until all ack messages of $rel(sr_i(x, v))$ are delivered, $del_j(rel(sr_i(x, v)))$ is ordered in $\tau''_j$ before the potential execution interval of $w_i(x, u)$. Since $sop_q$ is ordered before $del_j(rel(sr_i(x, v)))$ in $\tau''_j$ and after $del_j(rel(sr_i(x, v)))$ in $\tau''_j$, then $sop_q$ is a strong read and the write $sop_q$ reads from is ordered inside the potential execution interval of $w_i(x, u)$. Denote the write $sop_q$ reads from by $w_m$. By Lemma 5.6, $w_m$ influences every process and is therefore ordered in $\tau''_j$ immediately before $del_j(w_m)$. Thus, $w_m$ is ordered after $del_j(rel(sr_i(x, v)))$ in $\tau''_j$. By Lemma 5.7, $sop_q$ is ordered after $w_m$ in $\tau''_j$, and therefore after $del_j(rel(sr_i(x, v)))$. A contradiction.

Assume that $sr_i(y, w)$ is dragged by $sr_i(x, v)$ before $del_j(rel(sr_i(y, w)))$ in $\tau''_j$. By the minimality of $sr_i(x, v)$, $sr_i(y, w)$ is ordered after $del_j(rel(sr_i(y, w)))$ in every sequence $\tau''_j$ before the insertion of $sr_i(x, v)$. Thus, there exists a sequence $\tau''_j$ for which $del_j(rel(sr_i(x, v)))$ and $del_j(rel(sr_i(y, w)))$ are ordered inside a potential execution interval of some weak write $w_i(x, u)$ that includes a strong operation $sop_q$ which is ordered in $\tau''_j$ before $del_j(rel(sr_i(y, w)))$. Since $sop_q$ is ordered before $del_j(rel(sr_i(y, w)))$ in $\tau''_j$ and after $del_i(rel(sr_i(y, w)))$ in $\tau''_j$, it follows that $sop_q$ is a strong read and the write $sop_q$ reads from is ordered inside the potential execution interval of $w_i(x, u)$. Denote the write $sop_q$ reads from by $w_m$. By Lemma 5.6, $w_m$ influences every process and therefore is ordered in $\tau''_j$ immediately before $del_j(w_m)$. Thus, $w_m$ is ordered after $del_j(rel(sr_i(x, y, w)))$ in $\tau''_j$. By Lemma 5.7, $sop_q$ is ordered after $w_m$ in $\tau''_j$, and therefore after $del_j(rel(sr_i(y, w)))$. A contradiction.

Lemma 5.9 There exists a linearization $\rho$ of all the operations in $\sigma$ such that for every pair of strong operations $sop_k$ and $sop_i$, $sop_k \xrightarrow{\tau''_j} sop_i$ if and only if $sop_k \xrightarrow{\tau''_j} sop_i$ for every $\tau''_j$. 
**Proof:** We have to show that all the strong operations are ordered in all the sequences \( \tau''_j \) in the same order, and that this order is a linearization of the strong operations in \( \text{ops}(\sigma) \).

We first show that all strong operations appear in the same order in all sequences \( \tau''_j \). Assume, by way of contradiction, that there exist two strong operations \( \text{ sop}_k \) and \( \text{ sop}_l \) and two sequences \( \tau''_j \) and \( \tau''_l \) such that \( \text{ sop}_k \xrightarrow{\tau''_j} \text{ sop}_l \) but \( \text{ sop}_k \xrightarrow{\tau''_l} \text{ sop}_k \). By Lemma 5.1 and since strong writes are not dragged by strong reads, this can only happen if at least one of \( \text{ sop}_k \) or \( \text{ sop}_l \) is a strong read.

If both \( \text{ sop}_k \) and \( \text{ sop}_l \) are strong reads, then let \( \text{ sop}_k = \text{ sr}_i(x, v) \) and \( \text{ sop}_k = \text{ sr}_k(y, w) \). Assume, without loss of generality, that \( \text{ sr}_i(x, v) \) appears before \( \text{ sr}_k(y, w) \) in the order of insertion for strong reads. Thus, the **strong-read** message of \( \text{ sr}_k(y, w) \) is delivered after the **strong-read** message of \( \text{ sr}_i(x, v) \). Since \( \text{ sr}_k(y, w) \xrightarrow{\tau''} \text{ sr}_i(x, v) \), then either \( \text{ sr}_i(x, v) \in D(\text{ sr}_k(y, w)) \) or \( \text{ sr}_k(y, w) \) is dragged by another strong read before \( \text{ sr}_i \). In either case, \( \text{ sr}_k(y, w) \) is ordered before \( \text{ sr}_i(x, v) \) in every sequence \( \tau''_j \). A contradiction.

Otherwise, without loss of generality, \( \text{ sop}_k \) is a strong read and \( \text{ sop}_l \) is a strong write. Let \( \text{ sop}_k = \text{ sr}_i(x, v) \). If the **strong-read** message of \( \text{ sr}_i(x, v) \) is delivered before the **strong-write** message of \( \text{ sop}_k \), then \( \text{ sr}_i(x, v) \) is ordered before \( \text{ sop}_k \) in every sequence \( \tau''_j \). Therefore, the **strong-read** of \( \text{ sr}_i(x, v) \) is delivered after the **strong-write** message of \( \text{ sop}_k \). Since \( \text{ sr}_i(x, v) \) is ordered before \( \text{ sop}_k \) in \( \tau''_j \), then either \( \text{ sop}_k \in D(\text{ sr}_i(x, v)) \) or \( \text{ sr}_i(x, v) \) is dragged by another strong read before \( \text{ sop}_k \). In any case, \( \text{ sr}_i(x, v) \) is ordered before \( \text{ sop}_k \) in every sequence \( \tau''_j \). A contradiction.

We now show that the order by which strong operations appear in \( \tau''_j \) preserves the order implied by \( \rho \). Assume, by way of contradiction, that the order by which all strong operations appear in every sequence \( \tau''_j \) is not a linearization. Note that strong reads are always inserted before their corresponding delivery event. By Lemma 5.1 and since strong writes are not dragged by strong reads, the total order is not a linearization only if there exist a strong read \( \text{ sr}_i(x, v) \) and another strong operation \( \text{ sop}_k \) such that \( \text{ sop}_k \) terminates before \( \text{ sr}_i(x, v) \) begins but \( \text{ sr}_i(x, v) \xrightarrow{\tau''_j} \text{ sop}_k \) for every sequence \( \tau''_j \). Since, by definition, \( \text{ sop}_k \xrightarrow{\tau''_j} \text{ del}_l(\text{ sop}_k) \) for every sequence \( \tau''_j \), this can only happen in one of two cases:

**Case 1:** The delivery event of the **strong-read** message of \( \text{ sr}_i(x, v) \) is ordered inside the potential execution interval of some \( \text{ w}_i(x, v) \), and \( \text{ sop}_k \) is also ordered inside the same potential execution interval. Then the **ack** message for \( \text{ sop}_k \) is delivered in \( \text{ p}_k \) after the **update** message of \( \text{ w}_i(x, u) \), while the **strong-read** message of \( \text{ sr}_i(x, v) \) is delivered in all processes before the **update** message of \( \text{ w}_i(x, u) \). Hence, the **strong-read** message of \( \text{ sr}_i(x, v) \) is delivered in \( \text{ p}_k \) before \( \text{ sop}_k \) terminates. A contradiction to the assumption that \( \text{ sop}_k \) terminates before \( \text{ sr}_i(x, v) \) begins.

**Case 2:** \( \text{ sr}_i(x, v) \) is dragged by some other strong read \( \text{ sr}_q(y, w) \) before \( \text{ sop}_k \). Thus, the **strong-read** message of \( \text{ sr}_q(y, w) \) is delivered inside the potential execution interval of some weak write \( \text{ w}_i(y, z) \) that includes both \( \text{ del}_l(\text{ sop}_k) \) and \( \text{ del}_l(\text{ sr}_i(x, v)) \). Therefore, the **ack** message for \( \text{ sop}_k \) is delivered in \( \text{ p}_k \) after the **update** message of \( \text{ w}_i(y, z) \), while the **strong-read** message
of \( sr_i(x,v) \) is delivered in all processes before the update message of \( w_i(y,z) \). Hence, the strong-read message of \( sr_i(x,v) \) is delivered in \( p_k \) before \( sop_k \) terminates. A contradiction to the assumption that \( sop_k \) terminates before \( sr_i(x,v) \) begins.

Lemma 5.10 For every sequence \( \tau'_i, \tau''_i | j = \sigma | j \).

Proof: Consider any two operations \( op_j^1 \) and \( op_j^2 \) such that \( op_j^1 \overset{e}{\rightarrow} op_j^2 \). We show that \( op_j^1 \overset{r}{\rightarrow} op_j^2 \) by a simple case analysis. If neither \( op_j^1 \) nor \( op_j^2 \) is a strong read, then the claim holds by Lemma 5.2 and since the insertion of strong reads does not change the order of other operations. If both \( op_j^1 \) and \( op_j^2 \) are strong reads, then the claim holds by Lemma 5.9. If only \( op_j^1 \) is a strong read, then the claim holds since strong reads are inserted before their corresponding delivery event. If, on the other hand, only \( op_j^2 \) is a strong read, then by Lemma 5.8, \( del_j(\rel(op_j^2)) \overset{\tau''_i}{\rightarrow} op_j^2 \). By construction of \( \tau''_i \), \( op_j^1 \overset{\tau'_i}{\rightarrow} \) \( del_j(\rel(op_j^2)) \). Thus, \( op_j^1 \overset{\tau'_i}{\rightarrow} op_j^2 \) and the claim holds. (Note that we have covered all the cases since we still do not have weak reads by other processes.)

Lemma 5.11 For each sequence of operations \( \tau''_i \) and every pair of operations \( op_k^1 \) and \( op_k^2 \) in \( \tau''_i \) such that either \( op_k^1 \) or \( op_k^2 \) is strong, \( op_k^1 \overset{\tau'_i}{\rightarrow} op_k^2 \) if and only if \( op_k^1 \overset{e}{\rightarrow} op_k^2 \).

Proof: By Lemma 5.10, the claim holds for \( k = j \). Thus, we assume for the rest of the proof that \( k \neq j \). Assume, by way of contradiction, that for some sequence \( \tau''_i \) and two operations \( op_k^1 \) and \( op_k^2 \) such that either \( op_k^1 \) or \( op_k^2 \) is strong, \( op_k^1 \overset{\tau'_i}{\rightarrow} op_k^2 \) but \( op_k^1 \overset{e}{\rightarrow} op_k^2 \). By Lemma 5.5 and Lemma 5.9 and the construction of \( \tau''_i \), this can only happen in one of the following cases:

Case 1: \( op_k^1 \) is a strong read and \( op_k^2 \) is a weak write. By the construction of \( \tau''_i \), \( op_k^2 \) is ordered before \( del_j(\rel(op_k^2)) \). In particular, \( op_k^2 \) is ordered in \( \tau''_i \) before \( del_j(\rel(op_k^1)) \). By Lemma 5.8, \( op_k^1 \) is ordered in \( \tau''_i \) after \( del_j(\rel(op_k^1)) \). Thus, \( op_k^1 \overset{\tau'_i}{\rightarrow} op_k^2 \). A contradiction.

Case 2: \( op_k^2 \) is a strong read and \( op_k^1 \) is a weak write that does not influence \( p_k \). By definition, \( op_k^1 \) is ordered before \( del_j(\rel(op_k^2)) \) in every sequence \( \tau''_i \). Assume, without loss of generality, that \( op_k^1 \) writes to \( x \), and let \( w_i(x,w) \) be the last write that influences \( p_k \) before the delivery of the update message of \( op_k^1 \). Thus, \( op_k^1 \) is invoked before the delivery of the update or strong-write message of \( w_i(x,w) \) in \( p_k \). Moreover, \( op_k^1 \) is ordered in \( \tau''_i \) before \( op_k^2 \) because \( w_i(x,w) \) is ordered in \( \tau''_i \) before \( op_k^2 \). Since \( op_k^2 \) does not return before all of its ack messages are delivered, and since every ack message is sent using sendabc, the ack message for \( op_k^2 \) is delivered in \( p_k \) after the update or strong-write message of \( w_i(x,w) \). Thus, \( op_k^1 \) is invoked after the delivery of the update or strong-write message of \( w_i(x,w) \). A contradiction.
We now insert the weak reads. A sequence $\tau_j'''$ is constructed from each sequence $\tau_j''$, by inserting into $\tau_j''$ all the weak reads that are invoked by other processes than $p_j$, one after the other as follows: Every weak read $r_k(x,v), k \neq j$, is inserted in the first possible place such that it will be ordered after the previous strong operation by $p_k$ (if there is one) and after the write it reads from (if there is one). Note that this is well defined: if there is no previous strong operation and the weak read returns the initial value of the object, then the weak read is inserted at the beginning of the sequence.

**Lemma 5.12** For each sequence of operations $\tau_j'''$ and every pair of operations $op_k^1$ and $op_k^2$ such that either $op_k^1$ or $op_k^2$ is strong, $op_k^1 \rightarrow \tau_j''' \rightarrow op_k^2$ if and only if $op_k^2 \rightarrow \tau_j''' \rightarrow op_k^1$.

**Proof:** Assume, by way of contradiction, that there exist a sequence $\tau_j'''$ and two operations $op_k^1$ and $op_k^2$ such that either $op_k^1$ or $op_k^2$ is strong, $op_k^1 \rightarrow \tau_j''' \rightarrow op_k^2$ but $op_k^2 \rightarrow \tau_j''' \rightarrow op_k^1$. By Lemma 5.10, the claim holds if $k = j$. By Lemma 5.11 and since the insertion of weak reads does not change the order of other operations in $\tau_j'''$, the claim holds if neither $op_k^1$ nor $op_k^2$ is a weak read. Moreover, since weak reads are inserted after the previous strong operation by the same process, the only case in which $op_k^1$ may be ordered in $\tau_j'''$ before $op_k^2$ is when $op_k^2$ is a weak read, $op_k^1$ is a strong operation, $k \neq j$, and the write $op_k^2$ reads from is ordered after $op_k^1$ in $\tau_j'''$.

Assume, without loss of generality, that $op_k^2$ reads from $w_l(x,v)$. Lemma 5.11 and the assumption that $op_k^2$ reads from $w_l(x,v)$ imply that $l \neq k$. Thus, $w_l(x,v)$ influences every process and is therefore ordered in $\tau_j'''$ immediately before $del_j(w_l(x,v))$. If $op_k^1$ is a strong write, then it is ordered in $\tau_j'''$ immediately before $del_j(op_k^1)$ and $del_j(w_l(x,v))$. Thus, $del_j(op_k^1) \rightarrow \tau_j''' \rightarrow w_l(x,v) \rightarrow \tau_j''' \rightarrow del_j(w_l(x,v))$. If, on the other hand, $op_k^1$ is a strong read, then by Lemma 5.8, $op_k^1$ is ordered in $\tau_j'''$ after $del_j(rel(op_k^1))$ and $del_j(w_l(x,v))$. Thus, $del_j(rel(op_k^1)) \rightarrow \tau_j''' \rightarrow w_l(x,v) \rightarrow \tau_j''' \rightarrow del_j(w_l(x,v))$. In either case, by the use of atomic broadcast, the update or strong-write message of $w_l(x,v)$ is delivered in $p_k$ after $op_k^2$ is executed. A contradiction to the assumption that $op_k^2$ reads from $w_l(x,v)$.

**Lemma 5.13** Every read is legal in every sequence $\tau_j'''$.

**Proof:** By Lemma 5.7, all strong reads are legal in any of the sequences $\tau_j'''$. Thus, assume, by way of contradiction, that there exist a sequence $\tau_j'''$ and a weak read $r_i(x,v)$ such that $r_i(x,v)$ is not legal in $\tau_j'''$. Denote by $w_l(x,v)$ the write $r_i(x,v)$ reads from and by $sop_k$ the next strong operation by $p_k$. By Lemma 5.3 and since the insertion of (strong or weak) read operations does not change the order of write operations (by any process) or the order of weak reads by $p_i, i \neq j$. Thus, by the rules for inserting weak reads, there exists an obliterating write $w_k(x,u)$ such that $w_l(x,v) \rightarrow \tau_j'' \rightarrow w_k(x,u) \rightarrow \tau_j''' \rightarrow sop_k$.

If $k = j$, then the update or strong-write message of $w_k(x,u)$ is delivered at $p_k$ after the execution of $w_l(x,v)$ and before the delivery of the ack message of $sop_k$. Thus, either $w_k(x,u)$
or another write to \( x \) is executed by \( p_i \) between the execution of \( w_l(x, v) \) and the invocation of \( r_i(x, v) \). A contradiction to the assumption that \( r_i(x, v) \) reads from \( w_l(x, v) \).

Thus, \( k \neq j \). We may assume, without loss of generality, that \( w_k(x, u) \) influences \( p_j \). Otherwise, there exists another write \( w_q(x, w) \) that influences \( p_j \) such that \( w_k(x, u) \xrightarrow{\tau''_j} w_l(x, w) \xrightarrow{\tau''_j} sop_k \) and we could have chosen \( w_q(x, w) \). Thus, \( w_k(x, u) \) is ordered in \( \tau''_j \) immediately before \( del_j(w_l(x, u)) \). By Lemma 5.4, \( w_k(x, u) \) influences every process. By the construction of \( \tau''_j \), \( sop_k \) is ordered before \( del_j(sop_k) \). Thus, the update or strong-write message of \( w_k(x, u) \) is delivered at \( p_i \) after the execution of \( w_l(x, v) \) and before the delivery of the strong-write or strong-read message of \( sop_k \). Thus, \( w_k(x, u) \) is executed by \( p_i \) between the execution of \( w_l(x, v) \) and the invocation of \( r_i(x, v) \). A contradiction to the assumption that \( r_i(x, v) \) reads from \( w_l(x, v) \).

Finally, from each process \( p_j \), create a new sequence of operations \( \tau'_j \) by removing all delivery events from \( \tau''_j \).

**Theorem 5.14** Every execution generated by the algorithm is hybrid.

**Proof:** For an execution \( \sigma \), we have constructed a set of sequences \( \{\tau_j\}_{j=1}^n \). We now show why this set satisfies the requirements of the definition of hybrid consistency:

- By Lemma 5.13, every sequence \( \tau_j \) is legal.
- By construction, every sequence \( \tau_j \) is a permutation of \( ops(\sigma) \), which satisfies Condition 1 in the definition of hybrid consistency.
- By Lemma 5.12, for every pair of operations \( op_1 \) and \( op_2 \) such that either \( op_1 \) or \( op_2 \) is strong, \( op_1 \xrightarrow{\tau} op_2 \) if and only if \( op_1 \xrightarrow{\tau} op_2 \), which satisfies Condition 2 in the definition of hybrid consistency.
- By Lemma 5.9, there exists a linearization \( \rho \) of all operations in \( ops(\sigma) \) such that for every pair of strong operations \( sop_1 \) and \( sop_2 \), \( sop_1 \xrightarrow{\tau} sop_2 \) if and only if \( sop_1 \xrightarrow{\tau} sop_2 \), which satisfies Condition 3 in the definition of hybrid consistency.
- By Lemma 5.10, for every sequence \( \tau_j \), \( \tau_j | j = ops(\sigma) | j \), which satisfies Condition 4 in the definition of hybrid consistency.

\[ \blacksquare \]
5.3 Complexity Analysis

Since the algorithm uses an atomic broadcast mechanism, the actual time and message complexity depends on the complexity of this mechanism. We will make the complexity analysis with respect to \( t_{abc} \)—the time required to broadcast a message using \( abc \), \( t_{send} \)—the time required to send a message using \( send_{abc} \), \( n_{abc} \)—the number of physical messages required to perform \( abc \) and \( n_{send} \)—the number of physical messages required to perform \( send_{abc} \).

Weak reads are executed locally and cause no messages to be sent. Weak writes are executed instantaneously. Each weak write requires \( n_{abc} \) physical messages in order to broadcast its update message and \( n \cdot n_{send} \) physical messages for the corresponding \( ack \) messages.

A strong operation must wait for all \( ack \) messages of previous weak write operations and dummy messages, which could take \( t_{abc} + t_{send} \) time. Then, it broadcasts its own message and must wait until all \( ack \) messages are delivered, which could take an additional \( t_{abc} + t_{send} \) time. The total time required to execute a strong operation is therefore \( 2(t_{abc} + t_{send}) \).

A strong write operation requires \( n_{abc} \) physical messages in order to broadcast its \( strong\text{-write} \) message and \( n \cdot n_{send} \) physical messages for the corresponding \( ack \) messages. The total number of physical messages caused by a strong write operation is therefore \( n_{abc} + n \cdot n_{send} \).

A strong read operation requires \( n_{abc} \) physical messages in order to broadcast its \( dummy \) message and \( n \cdot n_{send} \) physical messages for the corresponding \( ack \) messages if the last previous operation was a weak read. Then, an additional \( n_{abc} \) physical messages are required to broadcast its \( strong\text{-read} \) message plus \( n \cdot n_{send} \) physical messages for the corresponding \( ack \) messages. The total number of physical messages caused by every strong read operation is therefore \( 2(n_{abc} + n \cdot n_{send}) \).

5.4 Atomic Broadcast

The algorithm for providing hybrid consistency described in this section is theoretical in nature. Its main purpose is to show that hybrid consistency can be provided such that weak operations are executed instantaneously while the time required to execute strong operations is linear with the network delay. This implementation is based on an atomic broadcast mechanism. Thus, our goal in this subsection is to find a sufficiently good atomic broadcast algorithm, not necessarily the best one; this algorithm should guarantee that the delivery time of messages is proportional to the network delay.

Much work has been done on atomic broadcast, and several atomic broadcast algorithms have been developed [8, 10, 15, 17, 21]. In particular, some of them deliver messages in \( 2d \) time, meeting our time complexity requirement.

In the rest of this section, we show an efficient centralized atomic broadcast algorithm. The delivery time of this algorithm is \( 2d \) for both \( abc \) and \( send_{abc} \). The message complexity is \( n + 1 \) messages for \( abc \) and 2 messages for \( send_{abc} \). Thus, if this algorithm is used in
the implementation of hybrid consistency, then we can take $n_{abc} = n + 1$, $n_{send} = 2$ and $t_{abc} = t_{send} = 2d$. By using these figures in the calculations of Section 5.3, we get a message complexity of $3n + 1$ physical messages for both strong and weak write operations, and $6n + 2$ physical messages for strong read operations. The time complexity of the strong operations is $8d$.

Recall that we assume a system of $n$ processes connected by some interconnection network. We assume that the links are reliable and FIFO. A special process, $p_0$, is used to order the messages of all processes. Thus, $abc$ is implemented by sending the message to the special process which forwards this message to all processes. Similarly, $send_{abc}$ is implemented by sending the message to the special process which forwards the message to the targeted process. Whenever a process receives a message from the special process, it delivers the message to the mcs. The precise code of the algorithm is given in Figure 5.

We are aware that on some specific architectures, this algorithm behaves worse than other distributed algorithms. On the other hand, in our abstract model, a virtual message sent from one process to the other is always counted as one physical message and its delivery time is not affected by the processes ids. Our algorithm requires no bookkeeping and, in our model, it is as efficient as any other atomic broadcast algorithm we are aware of. The two other main problems of centralized algorithms, namely load balancing and fault tolerance, can be solved using round robin on the job of the special process and backoffs [21]. These issues, however, are beyond the scope of this paper and are not discussed any further.
6 Lower Bounds for Implementing Hybrid Consistency

It is known ([10]) that in any (asynchronous) implementation of linearizability, the response time of ("strong") read and write operations is $\Omega(d)$. Thus, our algorithm shows that it is possible to implement hybrid consistency in such a way that weak operations are extremely fast, without sacrificing the response time of strong operations.

Our implementation of hybrid consistency supports Definition 3.1, that is, hybrid consistency where the strong operations are linearizable. In contrast, sequential consistency can be implemented more efficiently than linearizability [10]. This might lead one to think that hybrid consistency where the strong operations are sequentially consistent can be implemented with a better asymptotic response time than our implementation. The following theorems show that this is not the case, at least if weak operations are required to be "fast". These theorems are a variation of a similar theorem which appears in [39].

**Theorem 6.1** For any implementation of hybrid consistency, $|\text{read}| + |\text{swrite}| \geq d$.

**Proof:** Assume, by way of contradiction, that there is an implementation of hybrid consistency for which $|\text{read}| + |\text{swrite}| < d$. Let $p_1$ and $p_2$ be two processes that access $x$ and $y$. Without loss of generality, assume that $x$ and $y$ are initially 0.

By the specification of $y$, there is a hybrid execution $\sigma_1$ such that $\text{ops}(\sigma_1)$ is

$$[\text{SWrite}_1(x, 1), \text{SAck}_1(x), [\text{Read}_1(y), \text{Return}_1(y, 0)]],$$

$\text{SWrite}_1(x, 1)$ occurs at real time 0 and $\text{Read}_1(y)$ occurs immediately after $\text{SAck}_1(x)$. The delay of all messages in $\sigma_1$ is exactly $d$. By assumption, the real time at the end of $\sigma_1$ is less than $d$. Hence, no message is received at any node during $\sigma_1$.

Similarly, by the specification of $x$, there is some hybrid execution $\sigma_2$ such that $\text{ops}(\sigma_2)$ is

$$[\text{SWrite}_2(y, 1), \text{SAck}_2(y), [\text{Read}_2(x), \text{Return}_2(x, 0)]],$$

$\text{SWrite}_2(y, 1)$ occurs at real time 0 and $\text{Read}_2(x)$ occurs immediately after $\text{SAck}_2(y)$. The delay of all messages in $\sigma_2$ is exactly $d$. By assumption, the real time at the end of $\sigma_2$ is less than $d$. Hence, no message is received at any node during $\sigma_2$.

Since no message is ever received in $\sigma_1$ and $\sigma_2$, the partial execution obtained from $\sigma_1$ by replacing $p_1$’s history with $p_2$’s history in $\sigma_2$ can be extended to a hybrid execution $\sigma$. Then $\text{ops}(\sigma)$ consists of the operation $[\text{SWrite}_1(x, 1), \text{SAck}_1(x)]$ followed by $[\text{Read}_1(y), \text{Return}_1(y, 0)]$, and $[\text{SWrite}_2(y, 1), \text{SAck}_2(y)]$ followed by $[\text{Read}_2(x), \text{Return}_2(x, 0)]$.

Since $\sigma$ is hybrid, there exists a serialization $\rho$ of $\text{ops}(\sigma)$, such that for every $i \in \{1, 2\}$, there exists a legal serialization $\tau_i$ of $\text{ops}(\sigma)$, that preserves: (a) the order of operations at $p_i$,
(b) the order between strong and weak operations at all processes, and (c) the order between strong operations in \( \rho \). (As formalized in Definition 3.2.) Without loss of generality, in \( \rho \), \([\text{SWrite}_1(x, 1), \text{SAck}_1(x)]\) precedes \([\text{SWrite}_2(y, 1), \text{SAck}_2(y)]\).

Consider \( \tau_2 \). Since \( \tau_2 \) is legal, each read should precede the strong write to the same variable (by the other process) in \( \tau_2 \). Thus, in \( \tau_2 \), \([\text{Read}_2(x), \text{Return}_2(x, 0)]\) precedes \([\text{SWrite}_1(x, 1), \text{SAck}_1(x)]\), which, in turn, precedes \([\text{SWrite}_2(y, 1), \text{SAck}_2(y)]\). Therefore, in \( \tau_2 \), \([\text{Read}_2(x), \text{Return}_2(x, 0)]\) precedes \([\text{SWrite}_2(y, 1), \text{SAck}_2(y)]\), but then \( \tau_2 \models 2 \neq 2 \). A contradiction.

The interested reader can compare this proof with the proof of the similar result in [10, Theorem 3.1] to see the extra care needed when arguing about properties of hybrid executions.

Using the same arguments as in the proof of Theorem 6.1, only reversing the roles of reads and writes, we can prove:

**Theorem 6.2** For any implementation of hybrid consistency, \(|\text{write}| + |\text{read}| > d/2\).

We define a **fast write implementation** to be one in which the time required to execute weak write operations is strictly less than \( d/2 \). Likewise, a **fast read implementation** is an implementation in which the time required to execute weak read operations is strictly less than \( d/2 \). Note that the implementation given in the previous section is fast—weak operations return instantaneously. We immediately obtain:

**Corollary 6.3** Let \( I \) be a fast read implementation of hybrid consistency. Then \(|\text{write}| \geq d/2\).

**Corollary 6.4** Let \( I \) be a fast write implementation of hybrid consistency. Then \(|\text{read}| \geq d/2\).

### 7 Weak Consistency

Intuitively, weak consistency requires that for every process, all operations can be ordered in a way that is consistent with this process' view. However, it is possible that different processes see different orderings. Weak consistency implies that updates made by some process will eventually be executed by all processes. It does not guarantee that the order in which updates take place is the same at all processes or respects the order at which they were issued. Formally:

**Definition 7.1 (Weak consistency)** An execution \( \sigma \) is weakly consistent if for every process \( p_j \) there exists a legal serialization \( \tau_j \) of \( \sigma \) such that \( \tau_j \models j = \sigma \models j \).

Note that \( \tau_j \) represents the ordering of operations that is consistent with the view of \( p_j \).

An implementation of weak consistency is provided by the Pipelined RAM model of Lipton and Sandberg [39].\(^8\) In this implementation, each process holds a complete copy of all the

\(\footnote{In fact, the Pipelined RAM of Lipton and Sandberg supports a slightly stronger consistency condition than weak consistency. However, this only strengthen the impossibility result of this section.}\)
memory. An operation is executed immediately on the local copy of the object. In the case of a write operation, update messages are sent to all processes. A response is generated, although update messages still propagate in the network. When a process receives an update message it writes the new value to its local copy of the object.

Lipton and Sandberg ([39]) argue that fundamental coordination problems can be solved using weak consistency. To support their claim, they present a centralized solution to the mutual exclusion problem, that relies on weak consistency. In this section we show that, roughly speaking, all solutions to the mutual exclusion problem that rely only on weak consistency must be centralized. Centralized solutions are considered bad practice in concurrent programming; they cannot sustain a failure or a slow-down of the centralized controller, even when it is outside the critical section.

The mutual exclusion problem was formally defined in Section 4. To prohibit centralized solutions, we require that processes that are in the remainder section do not participate in the algorithm. Only processes that are trying to enter or leaving the critical section participate in the algorithm.

The exclusion set of a mutual exclusion algorithm \(A\) is the set of shared variables read from inside the entry or exit sections of \(A\); this set is denoted \(\text{exc}(A)\).

**Definition 7.2** A mutual exclusion algorithm \(A\) is non-cooperative if every process which executes the remainder section of \(A\) does not write any variable in \(\text{exc}(A)\); otherwise, the algorithm is cooperative.

In particular, all solutions in which a process wishing to enter the critical section must communicate with a fixed subset of the processes, even if these processes are not trying to enter (or are inside) the critical section, are cooperative. In particular, centralized solutions (like the one presented in [39]) are cooperative. The main result of this section is:

**Theorem 7.1** Any algorithm for the mutual exclusion problem which is based on weak consistency must be cooperative.

**Proof:** Consider a system with two processes \(p_1\) and \(p_2\), and assume, by way of contradiction, that there exists a non-cooperative algorithm \(A\), based on weak consistency, which solves the mutual exclusion problem. \(A\) must operate correctly under any implementation of weak consistency, in particular under the Pipelined RAM implementation of weak consistency [39]. (Later in the proof we assume that all the operations can be executed as fast as we want. Thus, we need to consider an implementation of weak consistency that has this property; the Pipelined RAM is an example of such an implementation.)

Consider the following execution \(\sigma^*_1\) of \(A\). In \(\sigma^*_1\), \(p_1\) and \(p_2\) take steps at each real time \(i\), for any integer \(i \geq 0\). The delay of each message in \(\sigma^*_1\) is \(d\). At time 0, \(p_1\) makes the transition from the remainder section to the entry section. On the other hand, \(p_2\) remains in the remainder
section. Since $A$ guarantees that there is no deadlock, after some finite number $t_1$ of its own steps $p_1$ makes the transition from the entry section to the critical section. Since $p_1$ is in the remainder section, and since $A$ is non-cooperative, $p_2$ does not write to any variable in $exec(A)$ during $\sigma'_1$. Let $\sigma_1$ be the steps in $\sigma'_1$ up to and including time $t_1$, when all steps are rescheduled to occur at time $< d$. This can be done since each process takes only a finite number of steps in a finite time. Since, in $\sigma'_1$, $p_2$ does not write any variable in $exec(A)$ while $p_1$ reads only variables from $exec(A)$, it follows that $p_1$’s history in $\sigma'_1$ is equal to $p_1$’s history in $\sigma_1$, until time $t_1$ (except for the timing of events).

Now consider the symmetric execution $\sigma'_2$ of $A$. In $\sigma'_2$, $p_1$ and $p_2$ take steps at each real time $i$, for any integer $i \geq 0$. The delay of each message in $\sigma'_2$ is $d$. At time 0, $p_2$ makes the transition from the remainder section to the entry section. On the other hand, $p_1$ remains in the remainder section. Since $A$ guarantees that there is no deadlock, after some finite number $t_2$ of its own steps $p_2$ makes the transition from the entry section to the critical section. Since $p_1$ is in the remainder section, and since $A$ is non-cooperative, $p_1$ does not write to any variable in $exec(A)$ during $\sigma'_2$. Let $\sigma_2$ be the steps in $\sigma'_2$ up to and including time $t_2$, when all steps are rescheduled to occur at time $< d$. This can be done since each process takes only a finite number of steps in a finite time. Since, in $\sigma'_2$, $p_1$ does not write any variable in $exec(A)$ while $p_2$ reads only variables from $exec(A)$, it follows that $p_2$’s history in $\sigma'_2$ is equal to $p_2$’s history in $\sigma_2$, until time $t_2$ (except for the timing of events).

Since no message is ever received in $\sigma_1$ and $\sigma_2$, the partial execution $\sigma$ obtained from $\sigma_1$ by replacing $p_1$’s history with $p_2$’s history in $\sigma_2$ can be extended to an execution. However, at the last state of $\sigma$ both $p_1$ and $p_2$ are in the critical section. A contradiction.

This proof relies on the fact that there is no lower bound on the step time of the processes. A similar proof can be proved for the case where such a lower bound exists, but there is no upper bound on the message propagation time. If none of these properties hold, i.e., there is a lower bound on the step time of the processes and there is an upper bound on the message propagation time, then the claim does not hold; that is, one can design a non-cooperative algorithm for the mutual exclusion problem. However, this algorithm must explicitly rely on timing information to provide correctness, a behavior which is better avoided.

Note that our proof does not rely on any sort of fairness. Also, the proof holds for both deterministic and randomized algorithms; this only makes the impossibility result stronger.

In a sense, some sort of cooperativeness must exist at some level. If a consistency condition is weak enough to be implemented in such a way that the hardware is “non-cooperative”, then the software must compensate for that and use cooperative algorithms to achieve synchronization. On the other hand, if the consistency condition is strong enough to support non-cooperative synchronization algorithms, then its implementation must be cooperative (cf. the atomic broadcast used in the algorithm of Section 5). Although both alternatives may be acceptable, we think designers should be aware of this tradeoff.
7.1 Other Weak Consistency Models

In this section we discuss several weak consistency conditions. Some of these conditions are weaker than the model we studied earlier in this section and some are stronger. We show that the impossibility result holds for all of them.

7.1.1 Afek, Brown and Merritt

The following definition by Afek, Brown and Merritt was given as Definition 3 in [4]:

A memory $M$ is consistent only if, for each of its executions $\sigma$ and all processes $i$, there exists an execution $\sigma_s$ of $M_{\text{serial}}$ such that $\sigma[i] = \sigma_s[i]$. 

Where $M_{\text{serial}}$ is a sequentially consistent memory.

This definition may seem very similar to Definition 7.1. However, our definition contains a liveness part and requires that every process eventually observes all operations of all processes. In contrast, according to the definition of [4], a process need only see its own operations.

It is trivial that the impossibility result holds for this definition, too.

7.1.2 Processor Consistency

Until recently, there was no a formal definition of processor consistency. Informally, processor consistency requires that the operations of each process will appear to be executed by all the processes in the order they were invoked [34]. The following is a formal definition which we believe captures this intuitive semantics.

Denote by $\sigma|\text{writes}$ all the write operations in a sequence of operations.

**Definition 7.3 (Processor Consistency)** An execution $\sigma$ is processor consistent if for every process $p_j$ there exists a legal serialization $\tau_i$ of $\text{ops}(\sigma)$ such that

1. $\tau_i[i] = \sigma[i].$
2. for every process $p_j$, $(\tau_i|\text{writes})[j] = (\sigma|\text{writes})[j].$

Note that the Pipelined RAM of [39] is processor consistent, according to Definition 7.3. Thus, the impossibility result holds for Definition 7.3 of processor consistency.

Recently, two formal definitions of processor consistency, PCG and PCD, were suggested [5]. These definitions have a slightly different interpretation of processor consistency. Both

---

PCD is supported by the Stanford DASH multiprocessor [29]. The D in PCD stands for DASH and the G in PCG stands for Goodman.
PCG and PCD are stronger than Definition 7.3 and the impossibility result does not hold for them. Below we present the definition of PCG in our framework, and show that for any implementation of PCG, the time required to execute two write operations and one read operation is at least $d$, where $d$ is the network delay. Similarly, it can be shown that for any implementation of PCD, the time required to execute two write operations and two read operations is at least $d$. (We do not present this proof since a formal definition of PCD requires additional notation, while the proof of the lower bound for PCD is almost identical to the proof of the lower bound for PCG.)

In a sense, this lower bound implies that implementations of PCG and PCD are no better (asymptotically) than implementations of sequential consistency. On the other hand, it is shown in [5] that not every program that was written for sequentially consistent hardware will run correctly on PCG or PCD hardware. Thus, it is unclear what are the benefits of implementing processor consistency according to definitions PCG or PCD.

To define PCG, we need some additional notation. Given a sequence of operations $\tau$, we denote by $\tau|(j+\text{writes})$ the restriction of $\tau$ to operations that are either write operations or operations by $p_j$. Intuitively, PCG requires that each process will view its own operations in the order they were invoked and that all processes will agree on the order of write operations to the same object. Formally:

**Definition 7.4 (PCG)** An execution $\sigma$ is PCG if for every process $p_j$ there exists a legal serialization $\tau_j$ of $\sigma|(j+\text{writes})$ such that

1. For every two operations $op_j^1$ and $op_j^2$ such that $op_j^1 \rightarrow op_j^2$ or $op_j^1 \rightarrow op_j^2$.

2. For every two write operations $w_j^1(x, v)$ and $w_j^2(x, u)$, $w_j^1(x, v) \rightarrow w_j^2(x, u)$ if and only if $w_j^1(x, v) \rightarrow w_j^2(x, u)$ for every object $x$, values $v$ and $u$ and process $p_j$.

The proof of the lower bound for PCG is along the lines of the proof of the lower bounds in [10] and in the previous section. Here, we only present its outline. Consider the following program:

$p_1$'s program

\[
\begin{align*}
& w_1(x, 1) \\ & w_1(z, 1) \\ & r_1(y)
\end{align*}
\]

$p_2$'s program

\[
\begin{align*}
& w_2(y, 1) \\ & w_2(z, 2) \\ & r_2(x)
\end{align*}
\]

If all three operations are executed faster than the network delay, then both reads must return 0 (assuming this is the initial value of all objects). In order to create the legal serializations, we need to choose an order between the writes to $z$. If $w_2(z, 2)$ is ordered before $w_1(z, 1)$, then we cannot construct a legal serialization for $p_1$. This is because in order for the serialization of $p_1$ to be legal, $w_2(y, 1)$ and therefore $w_2(z, 2)$ must be ordered after $r_1(y, 0)$. Since
must view its own operations in order they were invoked, \( r_1(y, 0) \) is ordered after \( w_1(x, 1) \) and \( w_1(z, 1) \). This contradicts to the requirement that \( w_2(z, 2) \) is ordered after \( w_1(z, 1) \) in all serializations. If \( w_1(z, 1) \) is ordered before \( w_2(z, 2) \), then symmetric arguments show that we cannot construct a legal serialization for \( p_2 \).

7.1.3 Causal Memory

A consistency condition called causal memory was suggested in [7]. An operation \( \text{op}_i^1 \) precedes \( \text{op}_j^2 \) (denoted \( \text{op}_i^1 \rightarrow \text{op}_j^2 \)) if either \( i = j \) and \( \text{op}_i^1 \) was invoked before \( \text{op}_j^2 \), or \( \text{op}_i^1 = w_i(x, v) \) for some memory location \( x \) and value \( v \) and \( \text{op}_j^2 = r_i(x, v) \).\(^\text{18}\) Let \( \rightarrow \) denote the transitive closure of \( \rightarrow \); \( \text{op}_i^1 \) and \( \text{op}_j^2 \) are concurrent if \( \text{op}_i^1 \not\rightarrow \text{op}_j^2 \).

Definition 7.5 Given a read operation \( r(x) \) and a write operation \( w(x, v) \), the value \( v \) is live for \( r(x) \) if either:

1. \( w(x, v) \) is concurrent with \( r(x) \), or
2. \( w(x, v) \) precedes \( r(x) \) with no intervening read or write of \( x \) with value \( v' \neq v \).

Definition 7.6 (Causal consistency) An execution is causally consistent if the value returned by any read operation in the execution is live for that read.

Causal consistency is strictly stronger than weak consistency. However, the proof of the impossibility result relies on the fact that the program cannot impose any global ordering on operations by different processes. Thus, the proof of the impossibility result holds for the implementation of causal memory given in [7], with some minor modifications. Thus, the impossibility result holds for causal memory, too.

8 Compositionality of Various Consistency Conditions

Compositionality is a property of consistency conditions defined by Herlihy and Wing ([35]), and called there locality. If \( C \) is a compositional consistency condition then a shared memory supporting \( C \) can be composed of independent implementations of each object (or group of objects), each supporting \( C \). Clearly, this is a very desirable property. More formally:

Definition 8.1 A correctness condition \( C \) is compositional if for every execution \( \sigma \), \( \sigma \) satisfies \( C \) if and only if \( \sigma \mid X \) satisfies \( C \) for every object \( X \).

\(^{18}\)This definition assumes that no value is written twice to the same memory location.
It is shown in [35] that linearizability is compositional, while sequential consistency is not compositional. We show:

**Theorem 8.1** Hybrid consistency is not compositional.

**Proof:** Consider the execution \( \sigma \) such that
\[
\sigma = r_1(x, 1), sr_1(y, 0), sw_2(y, 1), w_2(x, 1)
\]
where both \( x \) and \( y \) are initialized to 0. Note that
\[
\sigma \mid x = r_1(x, 1), w_2(x, 1) \quad \text{and} \quad \sigma \mid y = sr_1(y, 0), sw_2(y, 1).
\]
Clearly, \( \sigma \) is not a hybrid execution, while both \( \sigma \mid x \) and \( \sigma \mid y \) are hybrid executions.

This proof applies to both definitions of hybrid consistency. In contrast:

**Theorem 8.2** Weak consistency is compositional.

**Proof:** We need to show that \( \sigma \) is a weakly consistent execution, if and only if \( \sigma \mid x \) is a weakly consistent execution for each object \( x \). The execution \( \sigma \) is weakly consistent if, for every process \( p_j \), there exists a legal serialization \( \tau_j \) of \( \sigma \) such that \( \tau_j \mid j = \sigma \mid j \). Since \( \tau_j \) is legal, for each object \( x \), \( \tau_j \mid x \) is in the specification of \( x \). The execution \( \tau_j \mid x \) is a permutation of \( \sigma \mid x \) and \( \tau_j \mid j = \sigma \mid j \).

The execution \( \sigma \mid x \) is weakly consistent if, for every process \( p_j \), there exists a legal serialization \( \tau_j^x \) of \( \sigma \mid x \) such that \( \tau_j^x \mid j = (\sigma \mid x) \mid j \). Thus, any merging of the sequences \( \tau_j^x \), for all objects \( x \), forms a legal sequence \( \tau_j \) for process \( p_j \). The execution \( \tau_j \) is a serialization of \( \sigma \) and \( \tau_j \mid j = \sigma \mid j \).

Thus, hybrid consistency (based on linearizability) is not compositional although it is composed of two types of operations, weak and strong, each of which is compositional. We can also show:

**Theorem 8.3** Causal memory is not compositional.

**Proof:** Consider the following execution
\[
\sigma = r_1(y, 5), w_1(x, 2), r_3(x, 2), w_2(y, 5)
\]
\( ^{11} \)any interleaving of the operations of all the sequences that obeys the per sequence orderings

40
where both \( x \) and \( y \) are initialized to 0. Note that

\[
\sigma \mid x = w_1(x, 2), r_2(x, 2)
\]

\[
\sigma \mid y = r_1(y, 5), w_3(y, 5)
\]

\( r_1(y, 5) \) must be ordered after \( w_3(y, 5) \), which is possible in the projection for \( y \). Thus, \( \sigma \mid y \) is a causally consistent execution. Similarly, \( r_2(x, 2) \) must be ordered after \( w_1(x, 2) \), which is possible in the projection for \( x \). Thus, \( \sigma \mid x \) is a causally consistent execution. When examining \( \sigma \) as a whole, the requirements that \( r_2(x, 2) \) must be ordered before \( w_1(x, 2) \) and \( r_1(y, 5) \) must be ordered before \( w_1(x, 2) \) are added. Together, these four requirements create a cycle. Thus, \( \sigma \) is not a causally consistent execution.

\[
9 \quad \text{Discussion and Further Research}
\]

This paper presents a theoretical study of hybrid consistency. This study is motivated by several new processors and multiprocessors that support some sort of weak and strong operations; it is also motivated by recent theoretical results on the cost of supporting global consistency conditions and the inadequacy of weak consistency conditions. We have presented a formal and precise definition of hybrid consistency in two flavors, one based on sequential consistency and the other based on linearizability. We have presented an algorithm for providing hybrid consistency, proved its correctness and analyzed its performance. We have also shown lower bounds on the response time of any implementation of hybrid consistency that does not delay weak operations (which are within constant factors of the bounds achieved by our algorithm).

Our definition of hybrid consistency (Definition 3.2) assumes that operations of the same process are executed one at a time and in program order. Recently, our definition has been generalized to allow non-sequential executions [9], i.e., executions in which operations may be executed in parallel, in pipeline and even out of order. The definition in [9] also incorporates a formal treatment of control operations, which is not part of our definition. Naturally, the result is a slightly more complicated definition. We feel that Definition 3.2 is still interesting since it sheds light on hybrid consistency: how to program with it, what are its costs and possible algorithms for providing it. Furthermore, to the best of our knowledge, there is no formally verified algorithm for providing hybrid consistency, when operations can be issued in parallel, in pipeline and out of order.

Our work leaves open several theoretical, as well as practical, questions.

We would like to obtain tighter bounds (for time, messages and congestion) for implementations of hybrid consistency. In this work we assumed that local computations are negligible and that messages can be sent to the network without any limitations. It is important to obtain lower and upper bounds on implementations of hybrid consistency and other consistency conditions in a more realistic model. The Postal ([11]) or the LogP ([23]) models might serve as a middle ground between our theoretical assumptions and realistic architectures.
It would be interesting to quantify what are the performance benefits of programming with hybrid consistency. Specifically, are algorithms for solving a specific problem that rely on hybrid consistency faster than algorithms for solving the same problem that rely on other consistency conditions? If so, how much? These questions can be answered either by theoretical or experimental methods.

In this paper, we have made a moderate step towards developing proof techniques for hybrid consistency, by showing how to argue about the correctness of an example program. Subsequent work ([9]) presents several techniques for transforming programs written for sequentially consistent hardware into correct and efficient programs for hybrid consistent hardware. However, these proofs and transformations are somewhat ad-hoc and a more general approach should be developed.

Another interesting and important issue is the design of compiler techniques for using hybrid consistency. Such a compiler can, for example, take a program which was written assuming a sequentially consistent or linearizable memory, and decide automatically for each memory access operation whether it should be a weak or strong. This should be done without affecting the correctness of the program and while achieving the maximal possible speedup. (Such techniques for a sequentially consistent memory that allows pipelining appear in [47].)

The problem of verifying whether a given execution is sequentially consistent was investigated by Gibbons and Korach [30, 31]. They have shown that for most interesting cases, this problem is NP-complete, although under some restrictions it can be solved in polynomial time. It is clear that the problem of verifying whether a given execution is hybrid consistent is at least as hard. However, it is unclear whether verifying hybrid consistency is harder than verifying sequential consistency.

The ultimate goal of our research is to obtain a better understanding of memory consistency conditions, so it would be possible to evaluate several conditions and decide which is best suited for a specific application and architecture. In order to do so, a wide spectrum of properties need to be considered, such as performance, computation power and complexity of the possible implementations. It is necessary to formalize these properties and develop criteria for evaluating consistency conditions.

Acknowledgements: Jennifer Welch collaborated with us in the early stages of this research. We are indebted to her for her contributions to the definition of hybrid consistency and for many helpful discussions and comments on several versions of this paper. Thanks also to Sarita Adve, Eran Aharonson, Ken Birman, Kourosh Gharachorloo, Maurice Herlihy, Martha Kosa, Rivka Ladin, Friedemann Mattern, Lihu Rappoport and Tali Shintel for helpful remarks.
References


