A Correctness Condition for High-Performance Multiprocessors
(Preliminary Version)

by
H. Attiya and R. Friedman

Technical Report #719
March 1992
A Correctness Condition for High-Performance Multiprocessors*

(Preliminary Version)

HAGIT ATTIYA†
ROY FRIEDMAN
Department of Computer Science
The Technion
Haifa 32000, Israel

March 1992

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

Hybrid consistency, a new 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. An efficient implementation of hybrid consistency on distributed memory machines is presented. In this implementation, weak operations are executed instantaneously, while the response time for strong operations is linear in the network delay. (It is proven that this is within a constant 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 employ 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, at various system types and levels of abstraction. The major issue seems to be what is a good condition to be supported by the consistency mechanism: Which conditions can be implemented efficiently, which conditions can be used conveniently, and which conditions support faster programs.

Until recently, research on this subject addressed strong consistency conditions like sequential consistency and linearizability [7, 8, 9, 10, 11, 13, 14, 18, 21, 22, 25, 28, 30]. 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 requires a non-negligible cost (cf. [23, 5]). 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., [2, 4, 17, 19, 23]). Weak conditions 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., [3, 7, 8, 14, 15, 16, 28]). Hybrid conditions 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. 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.

\[^{1}\text{These conditions are similar in flavor to the notion of serializability from database theory ([6, 26]); however, serializability applies to transactions which aggregate many operations.}\]

1
Very few of the previous definitions of weak and hybrid consistency are formal. Moreover, most of them talk about which restrictions should be put on the hardware, rather than the way it appears to the programmer; hence, they are architecture dependent. This makes it difficult to evaluate and compare these conditions without regarding a specific implementation. Furthermore, it is hard to design portable algorithms that rely on these consistency conditions, or prove their correctness.

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. 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. [18]). 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 ([27]) 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 the algorithms.

Our second major contribution is an algorithm that implements hybrid consistency on distributed memory machines [7, 8, 22, 28]; our implementation 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 assigning timestamps to messages, and hence to operations. The algorithm guarantees that strong operations are executed at all processes in the order implied by their timestamps. A strong operation returns only after it is executed by all processes. Combined with the fact that operations by the same process are executed in the order they were invoked, this guarantees the second property of hybrid consistency. By adapting proof techniques from [5, 23], we show that the response time of our algorithm is within a constant factor of the optimum.

Weak consistency admits very efficient implementations ([23]) and the reader might ask why we need hybrid consistency at all. Indeed, Lipton and Sandberg present solutions, to
several coordination problems, that rely only on weak consistency, the fundamental of them is a mutual exclusion algorithm [23]. 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 it wishes to enter the critical section or not. 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—they require the participation of certain processes, whether they wish to enter the critical section or not. This holds even for slightly stronger consistency conditions such as causal memory [2, 4] and processor consistency [17]. 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. [12, 27, 29]).

Many memory consistency conditions have been defined. However, we lack the ability to evaluate them on a common scale. In order to do so, a wide spectrum of formal criteria must be defined and checked. In this paper, we take one such criteria, called compositionality\(^2\) and check it for 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.

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. Section 4 provides an example of programming with hybrid consistency. In Section 5, we present an implementation of 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 consisting of a collection of nodes \(P\) connected via a communication network. 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

\(^2\)originally defined by Herlihy and Wing ([18]) and called there locality.
several coordination problems, that rely only on weak consistency, the fundamental of them is a mutual exclusion algorithm [23]. 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 it wishes to enter the critical section or not. 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—they require the participation of certain processes, whether they wish to enter the critical section or not. This holds even for slightly stronger consistency conditions such as causal memory [2, 4] and processor consistency [17]. 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. [12, 27, 29]).

Many memory consistency conditions have been defined. However, we lack the ability to evaluate them on a common scale. In order to do so, a wide spectrum of formal criteria must be defined and checked. In this paper, we take one such criteria, called compositionality\textsuperscript{2} and check it for 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.

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. Section 4 provides an example of programming with hybrid consistency. In Section 5, we present an implementation of 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 consisting of a collection of nodes $P$ connected via a communication network. 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

\textsuperscript{2}originally defined by Herlihy and Wing ([18]) and called there locality.
A 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.

In more detail, the following events may occur at the mcs process on node $p_i$:

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 nodes $p_i$ and $p_j$: the mcs process on node $p_i$ receives message $m$ from the mcs process on node $p_j$.

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 $p_i$ sends message $m$ to the mcs process on node $p_j$.

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
\( p \)'s transition function acting on \( s \) and \( i \). A history of a process \( p \) is a mapping \( h \) from \( \mathbb{R} \) (real
time) to finite sequences of steps such that

1. for each real time \( t \), there is only a finite number of times \( t' < t \) such that the correspond-
ing 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 between the messages sent by \( p_i \) to \( p_j \) and the messages received by
\( p_j \) from \( p_i \), 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.)

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. (This is a restriction on the
application program.)

A history \( h' \) is a partial history of a history \( h \) if there exists a time \( t_0 \) such that \( h'(t) = h(t) \)
for every time \( t < t_0 \), \( h'(t_0) \) is a prefix of \( h(t_0) \) and there is no time \( t_1 > t_0 \) in which \( h'(t_1) \) is
nonempty. A partial execution is a collection \( \mathcal{E} \) of partial histories, one for every process, in
which for every message received by \( p_j \) from \( p_i \) in \( \mathcal{E} \) there exists one message sent by \( p_i \) to \( p_j \)
in \( \mathcal{E} \), for every two processes \( p_i \) and \( p_j \). Given a partial history \( h \) of a process \( p_i \), and assume
\( t_1 \) is the largest time for which \( h(t_1) \) is not empty, the local state of \( p_i \) at the end of \( h(t_1) \) is the
last state of \( p_i \) in \( h \). A global state is the collection of last states of all processes in a partial
execution. Given the global state of a partial execution \( \mathcal{E} \), a previous global state is the global
state of a partial execution \( \mathcal{E}' \) such that every partial history in \( \mathcal{E}' \) is a partial history of a
history in \( \mathcal{E} \). From now on, we will 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 \textit{op}; \textit{op terminates} when the mcs process issues the appropriate response for \textit{op}. An operation is \textit{pending} if it was invoked and did not terminate. We consider only systems in which 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 \textit{x} implemented by it, and an operation type \textit{op} on \textit{x}, we denote by \(|\text{op}(\textit{x})|\) the maximum time taken from the invocation to the termination of an operation of type \textit{op} on \textit{x} in any admissible execution. We denote by \(|\text{op}|\) the maximum of \(|\text{op}(\textit{x})|\) over all objects \textit{x} implemented by the mcs.

Every object is assumed to have a \textit{serial specification} (cf. [18]) defining a set \textit{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 \([\text{Read}_i(\textit{x}), \text{Return}_i(\textit{z}, \textit{v})]\) forms an \textit{operation} for any \(p_i, \textit{x},\) and \(v\), as does \([\text{Write}_i(\textit{z}, \textit{v}), \text{Ack}_i(\textit{x})]\). 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 \textit{legal} if, for every object \textit{x}, the restriction of \(\tau\) to operations of \(\textit{x}\) is in the serial specification of \(\textit{x}\).

Given an execution \(\sigma\), let \(\text{ops}(\sigma)\) 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 \(\sigma\) implies a partial order, \(\overset{\sigma}{\rightarrow}\), on the operations which appear in \(\sigma\); \(\textit{op}^1 \overset{\sigma}{\rightarrow} \textit{op}^2\) if \(\textit{op}^1\) completely precedes \(\textit{op}^2\) in \(\text{ops}(\sigma)\).

Given an execution \(\sigma\), a sequence of operations \(\tau\) is a \textit{serialization} of \(\sigma\) if it is a permutation of \(\text{ops}(\sigma)\). A serialization \(\tau\) of \(\sigma\) is a \textit{linearization} if it extends \(\overset{\sigma}{ightarrow}\); that is, if \(\textit{op}^1 \overset{\sigma}{\rightarrow} \textit{op}^2\) then \(\textit{op}^1 \overset{\tau}{\rightarrow} \textit{op}^2\).

Let \(\tau\) be an execution. Denote by \(\tau \upharpoonright j\) the restriction of \(\tau\) to operations invoked by \(p_j\); similarly, denote by \(\tau \upharpoonright \textit{x}\) the restriction of \(\tau\) to operations on object \(\textit{x}\).

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

\textbf{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 \(p_j\), \(\sigma \upharpoonright j = \tau \upharpoonright j\).

\textbf{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 \textit{strong}; all other operations are called weak. In the case of read/write objects this means that it is possible to use \textit{strong reads} and \textit{strong writes}. We reserve Read and Write for weak operations, and denote call events for strong
operations by SWrite\((x, v)\) and SRead\((x)\) and the respective response events by SAck\((x)\) and 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 definition we are about to give 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. Formally:

**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 \(p_j\), 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 \(op_1^i \overset{\rho}{\rightarrow} op_2^i\) and at least one of \(op_1^i\) and \(op_2^i\) is strong, then \(op_1^i \overset{\tau_j}{\rightarrow} op_2^i\), for any \(i\).
3. If \(op_1 \overset{\rho}{\rightarrow} op_2\) and \(op_1^i\) and \(op_2^i\) are strong, then \(op_1^i \overset{\tau_j}{\rightarrow} op_2^i\).
4. \(\tau_j \mid j = \sigma \mid j\).

This definition says that for every process, all operations can be ordered in a way that is consistent with this process' view. It is still possible that different processes see different orderings, but they should agree on the order of strong operations (specified by \(\rho\)), and on the ordering of a weak operation and a strong operation initiated by the same process. Thus, the views of different processes are quite consistent after a strong operation is executed; this property can be exploited when designing coordination algorithms.

The definition requires the existence of a linearization \(\rho\) of \(\sigma\). However, \(\rho\) is used only to force a global order for all the strong operations in each \(\tau_j\). 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 [5], we might be interested in examining hybrid models in which strong operations are only sequentially consistent. This leads to the following definition 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 $p_j$, there exists a legal sequence of operations $T_j$ with the following properties:

1. $T_j$ is a permutation of $\text{ops}(\sigma)$.
2. If $op_1 \rightarrow op_2$ and at least one of $op_1$ and $op_2$ is strong, then $op_1 \xrightarrow{T_j} op_2$, for any $i$.
3. If $op_1 \rightarrow op_2$ and $op_1$ and $op_2$ are strong, then $op_1 \xrightarrow{T_j} op_2$.
4. $T_j \mid j = \sigma \mid j$.

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

In this section we present a detailed comparison of hybrid consistency, as defined in this paper, with other definitions that distinguish between two kinds of operations.

The major drawback of most previously known definitions of hybrid consistency is that they are given by specifying, at some level 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.

We believe it is much better to define the consistency condition at the interface between the application program and the mcs: First, it is easier to write programs when the exact behavior of the hardware need not be kept in mind. Second, programs written for a particular consistency condition should be portable from one implementation of the consistency condition to another. Third, having no direct constraints on the hardware allows better implementations of the intended semantics of the consistency conditions.

We now turn to a more detailed comparison of our definition of hybrid consistency with related definitions.
Dubois, Scheurich and Briggs

The most related condition is weak ordering, defined by Dubois, Scheurich and Briggs ([14]). 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 variables 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 obeys the intended semantics of the definition and give better performance. For example, the implementation given in [14] could be improved by pipelining weak operations with strong operations. This improvement obeys 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) has the semantics intended by the definition in [14]:

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 $\text{ops}(\sigma)$ 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 $\text{op}_1^1 \in (\text{ops}(\sigma) \setminus X)$ or $\text{op}_2^2 \in (\text{ops}(\sigma) \setminus X)$ and $\text{op}_1^1 \rightarrow^\tau \text{op}_2^2$, then $\text{op}_1^1 \rightarrow^{\tau_j} \text{op}_2^2$.
3. if $\text{op}_1^1, \text{op}_2^2 \in (\text{ops}(\sigma) \setminus X)$ and $\text{op}_1^1 \rightarrow^\rho \text{op}_2^2$, then $\text{op}_1^1 \rightarrow^{\tau_j} \text{op}_2^2$.
4. $\tau_j \mid j = \text{ops}(\sigma) \setminus j$.

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 objects. This property is exploited in the mutual exclusion algorithm given in Section 4. In this algorithm, the reads from synchronization objects are weak. The result is a reduced number of messages in the system.
Adve and Hill

A different approach splits the responsibility for the consistency condition between software and hardware. For example, the following definition of *weak ordering* was given by Adve and Hill ([3]):

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

This definition 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. Then it is guaranteed that the programs will run as if the mcs was sequentially consistent. In this approach the mcs provides some guarantee about the results of memory access operations only for programs that obeys DRF0. 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.

Release Consistency

*Release consistency*, defined by Gharachorloo et al. ([15]), is also similar to hybrid consistency, and is supported by the Stanford DASH multiprocessor. The definition given in [15] assumes three types of operations: ordinary, sync and nsync. Each type of operation 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.

Aside from being given at the interface between the mcs and the network, this definition has one other major fault: it is given as a list of sufficient conditions rather than a formal abstract definition.
In [16] a formal definition of release consistency is presented. However, the definition in [16] appears to be a formal description of the transition function of the mcs automaton. This definition is quite complex, and makes reasoning about programs directly with release consistency a very difficult task. To overcome this problem, a synchronization model called PL is defined; this model is very similar to DRFO. It is then guaranteed that any program which obeys this synchronization model will run as if the mcs was sequentially consistent. This approach and the one taken by Adve and Hill both have the following drawback: The definition of the mcs 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).

There is a fundamental difference between our approach for defining hybrid consistency and the approach used in [16]. The latter starts with a loose definition and restricts it. We start with a global condition, e.g., linearizability, and relax it. In return, our definition is more simple and easier to understand.

4 Programming with Hybrid Consistency

Hybrid consistency supports a very simple method of programming: use only strong operations and "think sequential consistency" (or linearizability, 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 [23]. Unfortunately, this method turns out to be "too weak", in the sense that is discussed below, in Section 7.

We believe that the best programming method is to use weak operations most of the time and strong operations only in those places where global ordering is required. We do not have a well-developed and systematic method for doing so in an optimal manner, i.e., with the minimum possible number of strong operations. Instead, we give an example of a mutual exclusion algorithm based on hybrid consistency, and prove its correctness. This algorithm shows that programming with hybrid consistency is 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. [29]). 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.
is a deadlock. A deadlock occurs if
where
\[ s_{wh}(turn, 1 - i) \land s_{wo}(need[i], true) \land r(turn) = 1 - i \]
then goto 3;
<critical section>
4: \( s_{w}(need[i], false) \);
<remainder section>

Figure 2: Mutual exclusion using hybrid consistency – code for process \( p_i \), \( i = 0, 1 \).

Figure 2 presents a solution for two processes which guarantees fairness; it is a simple modification of Peterson’s algorithm \[27\]. 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 \( s_{w}(x, v) \) 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. Consider the last \( s_{w}(turn, 1) \) executed by \( p_0 \) before entering the critical section, and the last \( s_{w}(turn, 0) \) executed by \( p_1 \) before entering the critical section. Since \( \sigma \) is a hybrid execution, there exists a serialization \( \rho \) of \( \sigma \). Without loss of generality, \( s_{w}(turn, 1) \) appears before \( s_{w}(turn, 0) \) in \( \rho \). Consider now the legal serialization \( \rho \) of \( \sigma \), as guaranteed by hybrid consistency. Since \( p_1 \) entered its critical section, it follows that \( p_1 \)’s last \( r(\text{turn}) \) before entering returned 1. However, by assumption, \( s_{w}(turn, 1) \overset{\rho}{\rightarrow} s_{w}(turn, 0) \). By the definition of \( r(\text{turn}) \) it follows that \( s_{w}(turn, 1) \overset{\tau_1}{\rightarrow} s_{w}(turn, 0) \overset{\tau_1}{\rightarrow} r(\text{turn}, 1) \). Thus, \( \tau_1 \) is not legal.

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 \( r_0(\text{need}[1], true) \) and \( r_0(\text{turn}, 1) \) in Line 3 and \( p_1 \) continually reads \( r_1(\text{need}[0], true) \) 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
\[
 s_{w}(turn, 1) \overset{\rho}{\rightarrow} s_{w}(turn, 0) \overset{\rho}{\rightarrow} s_{w}(\text{need}[1], true),
\]
where \( s_{w}(turn, 1) \) are the strong writes by \( p_0 \) that precede the infinite loop in Line 3, and \( s_{w}(\text{need}[1], true) \) \( s_{w}(turn, 0) \) 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 $r_0'(\text{need}[1], \text{true})$ and $r_0'(\text{turn}, 1)$ be some pair of these read. Since $\tau_0$ is legal,

$$sw'_0(\text{turn}, 1) \xrightarrow{\tau_0} sw'_1(\text{turn}, 0) \xrightarrow{\tau_0} sw'_1(\text{need}[1], \text{true}) \xrightarrow{\tau_0} r'_0(\text{need}[1], \text{true}).$$

By the definition of $\tau_0$, $r'_0(\text{need}[1], \text{true}) \xrightarrow{\tau_0} r'_0(\text{turn}, 1)$. 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 **fair**, if 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 fair under hybrid consistency.

**Proof:** Assume, by way of contradiction, that there exists an infinite hybrid execution $\sigma$ of the algorithm in which one process is trying to enter the critical section, and never granted permission. We may assume, without loss of generality, that $p_0$ is this process, and let $sw'_0(\text{turn}, 1)$ be the last write of $p_0$ to $\text{turn}$. By Lemma 4.1, $p_1$ must enter the critical section infinitely often. Since $\sigma$ is a hybrid execution, there exists a serialization $\rho$ of all the strong operations in $\text{ops}(\sigma)$. Moreover, every operation in $\sigma$ must appear in $\rho$ after a finite number of operations. Thus, there exists an $i$ such that $sw'_0(\text{turn}, 1) \xrightarrow{\rho} sw'_1(\text{turn}, 0)$. Consider the 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:

$$sw'_0(\text{turn}, 1) \xrightarrow{\tau_1} sw'_1(\text{turn}, 0) \xrightarrow{\tau_1} r_1(\text{turn}, 1).$$

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

It suffices to use weak operations inside the critical section; the strong write at Line 4 guarantees that all the operations invoked inside the critical section are ordered in an order consistent with the order of entrance to the critical section by their invoking processes. Also, 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 Implementing Hybrid Consistency

The implementation presented in this paper supports read/write objects accessible by either strong or weak operations. We assume that the network is reliable, i.e., any message sent is eventually delivered; we also assume that the communication links are FIFO.$^3$

---

$^3$This assumption can be removed.
Each process holds a local copy of the entire memory, a local timestamp counter and an array which contains conservative estimates about the values of all other timestamp counters in the system. A weak operation is executed instantly on the local copy of the object. In case of writes, update messages are sent to all the other processes, which update their local copies of the memory upon receiving these messages.

Timestamps are used to enforce global ordering on the strong operations. Strong operations are timestamped with the local timestamp counter, and a message is sent to all processes; the initiating process then increments its local timestamp counter by 1. The execution of any strong operation is postponed until the timestamp of that operation is smaller than all the estimated timestamp counters of the system. If more than one strong operation can be executed together, they are executed according to their timestamps and ids in increasing order. The algorithm guarantees that if process $p_i$ estimates $p_j$'s counter as $z$, then the local timestamp of $p_j$ is at least $z$ (that is, the estimate is conservative). It can be shown that this implies that all strong operations ever invoked by $p_j$ bearing timestamp smaller than $z$ have arrived at $p_i$, and ensures that all strong operations are executed in the same order. A strong operation does not return before all processes acknowledge they have executed it. This ensures that weak operations which were invoked later are executed later.

In order to guarantee good response time, it is necessary that $p_i$ increments its local timestamp counter as a result of receiving a message about a strong operation. However, it can be shown that incrementing the local timestamp counter whenever such a message is received, causes many messages to be sent concurrently when many strong operations are initiated concurrently. To avoid congestion, before a process increases its own timestamp counter, it makes sure that the timestamp counters of all other processes are very close together. This complicates the code and the liveness proof, but results in an algorithm with low congestion.

### 5.1 A Detailed Description of the Algorithm

We assume a system of $n$ processes, connected by some interconnection network, each capable of holding a local copy of the entire memory.

Each process $p_i$ has a local timestamp counter, $lts_i$, initially 0, and an array $ts_i$ such that $ts_i[j]$ contains $p_i$'s estimate of $lts_j$. (We use the convention that the local version of a variable held by process $p_i$ is subscripted with $i$.)

Weak operations are executed locally and instantly. If a weak operation is a write of $v$ to object $x$, then update messages are sent to all processes. An update message includes the new value $v$ and the object $x$ to be updated. A process which receives an update message of $v$ to object $x$, updates its copy of object $x$ with $v$.

For any strong write operation, a strong-op message is sent to all other processes; this message contains the object $x$ to be updated, the new value $v$, and a timestamp $lts$. For any strong read operation, a strong-op message is sent to all other processes; this message contains $lts$. Process $p_i$ suspends the execution of a strong operation with timestamp $ts$, until it knows
that the counters of all processes are at least \( ts + 1 \). When several pending strong operations may be executed, they are executed according to their timestamps and \( ids \) in increasing order. Executing a strong write operation at process \( p_i \) is done by updating the proper object in the local copy. Executing a strong read operation at process \( p_i \) is done by reading the value of the appropriate object in the local copy of the memory, if the initiator is \( p_i \), and doing nothing otherwise. After the strong operation is executed, an \( \text{ack} \) message is sent to its initiator.

When a process receives \( \text{ack} \) messages from all processes, it generates a response for the pending strong operation. (The algorithm guarantees this operation is unique.)

Process \( p_i \) increases its timestamp in each of the following cases:

1. After \( p_i \) sends a strong-op message to all processes.
2. After \( p_i \) receives a strong-op message with timestamp equal to \( lts_i \) and for all \( j \), \( ts_i[j] \geq lts_i \).
3. A strong operation with \( ts = lts_i - 1 \) was executed in \( p_i \), and there exists \( k \) such that \( ts_i[k] > lts_i \).

In the last two cases, a \( \text{ts-update} \) message is sent to all other processes.

The following is a brief explanation of the above cases. In order to avoid deadlocks when \( p_i \) stops initiating strong operations, it is necessary that \( p_i \) will increment \( lts_i \) as a result of receiving a strong-op message from other process. However, it can be shown that incrementing \( lts_i \) whenever a strong-op is received, results in a high message complexity when many strong messages are initiated concurrently. To avoid congestion, Case 2 has more requirements. (These requirements are exploited in Section 5.4.1.) Unfortunately, as is, Case 2 may cause a deadlock in the following scenario: Process \( p_i \) receives a strong-op message with timestamp \( x \), corresponding to the strong operation \( sop^1 \), before executing a strong operation \( sop^0 \) with timestamp \( x - 1 \). Process \( p_i \) then executes \( sop^0 \). After a while, for all \( k \), \( ts_i[k] \geq x \), and there exists \( l \) such that \( ts_i[l] = x \). If no other process initiates a strong operation, \( sop^1 \) will never get executed. Case 3. takes care of this problem.

Vector timestamps are used in [4, 20, 24]. In these papers, as in our algorithm, timestamp vectors are used to preserve causality. However, in [4, 20, 24] the whole vector is sent with each message, while in our algorithm only the local timestamp counter is sent.

The implementation of hybrid consistency presented in [8] inspired our implementation of hybrid consistency. However, the main difference between the two implementations is that [8] use real time clocks and we use logical clocks [21]. The implementation in [8] use a mechanism called \textit{grey zone} to determine when a strong operation may get executed at a local node. This mechanism is not used explicitly in our implementation following optimizations we made in the original algorithm, and for clarity of presentation.
5.2 Pseudocode

The algorithm uses the following data types:

- **timestamp** = integer
- **pending_op** = record with fields
  - **obj**: name of an object (object to be updated)
  - **op**: name of an operation (read or write)
  - **val**: value of obj (value to be written)
  - **ts**: timestamp (assigned by initiator)
  - **id**: process id (of the initiator)

The state of each process $p_i$ consists of the following components:

- **ts**: array[1..n] of timestamps, all initially 0
  (estimates (from below) of timestamps of all processes)
- **pending**: list of records of type pending_op sorted by timestamp and id, initially empty
  (set of strong operations waiting to be executed at the local copies)
- **mem**: copy of every object, initially equal to its initial value

By $lts$, we denote $ts[i]$, the local timestamp counter.

The algorithm uses the procedure **check_list** to scan the list of pending strong operations for operations which can be executed and execute them (Figure 3). The procedure searches the list of pending (strong) operations for ones with timestamp smaller than $\min_j\{ts[j]\}$. Since the pending operations are ordered according to their timestamps and their initiator's id, this is also the order in which they will be executed. The procedure is activated whenever there is a change in one of the components of $ts$. The rest of the algorithm is described in an interrupt driven form (Figure 4).
Read(x):
generate Return(x, mem[x])

Write(x,v):
mem[x] := v
send update(x, v) to all other processes
generate Ack(x)

SRead(x):
send strong-op('read', lts, x, ⊥) to all processes (including self)
lts := lts + 1

SWrite(x,v):
send strong-op('write', lts, x, v) to all processes (including self)
lts := lts + 1

received update(x, v) from Pj:
mem[x] := v

received strong-op(op, t, x, v) from Pj:
add (x, op, v, t, j) to pending /* sorted by timestamp and id */
ts[j] := t + 1
if t = lts and min_j{ts[j]} ≥ lts then
lts := lts + 1
send ts-update(lts) to all other processes
endif
check_list

received ts-update(t) from Pj:
ts[j] := t
check_list

received ack from Pj:
if n ack messages were received then
generate the appropriate response (SAck or SReturn)
endif

Figure 4: The algorithm — code for process pi.
5.3 Proof of Correctness

For the rest of the proof, fix an arbitrary execution, $\sigma$, of the algorithm; all claims refer to $\sigma$. We say that an operation on some object is executed by a process when it is applied to the local copy of the object. We start by showing that every operation is eventually executed by all processes. We then show that any execution generated by the algorithm is hybrid.

Weak operations are executed instantly in the local copy of the memory. In case of writes, update messages are sent to all the other processes. Since we assumed reliable links, the update messages are eventually received by all the other processes, which execute them immediately. It remains to prove that strong operations are eventually executed by all processes. We start with:

Lemma 5.1 Let sop be a strong operation, invoked at some state, $s$, of $\sigma$, with timestamp $x > 0$. Then there exists a strong operation sop', which was invoked at a previous state, $s'$, of $\sigma$, with timestamp $x - 1$.

Proof: Assume, by way of contradiction, that there exists a strong operation sop with timestamp $x > 0$, and no strong operation is invoked in any of the previous states of $\sigma$ with timestamp $x - 1$.

Let $y$ be the smallest value for which there is a strong operation sop with timestamp $y > 0$, such that there is no strong operation with timestamp $y - 1$. By the algorithm, sop has timestamp $y$ if and only if lts$_i = y$ when sop$_i$ was invoked.

By the algorithm, $p_i$ could have increased lts$_i$ from $y - 1$ to $y$ in one of the following cases: (a) It has invoked a strong operation with timestamp $y - 1$. (b) It has executed a strong operation with timestamp $y - 1$. (c) It has received a strong-op message from some process $p_j$ with timestamp $y - 1$. By the minimality of $y$, none of these cases is possible. A contradiction.

Lemma 5.2 If process $p_i$ invokes a strong operation with timestamp $x$ during some state $s$ of $\sigma$, then eventually, for all processes $p_j$ and $p_k$, $ts_j[k] \geq x + 1$.

Proof: Assume, by way of contradiction, that for some strong operation with timestamp $x$, there exist $j$ and $k \in [n]$, such that $ts_j[k] \leq x$ forever. Let sop$_i$ be an operation with smallest timestamp, $y$, for which this happens.

If $y = 0$, then it is clear that for any $j$ and $k \in [n]$, $ts_j[k] \geq 0$. Otherwise, by Lemma 5.1, there is at least one strong operation with timestamp $y - 1$. By the minimality of $y$, it is clear that for any $j$ and $k \in [n]$, eventually $ts_j[k] \geq y$.

From this fact and the algorithm, each strong operation with a timestamp smaller than $y$ is eventually executed by all processes. Since the links are reliable, eventually all strong-op messages with timestamp $y$ will be received by all the processes, therefore eventually for any process $p_j$, lts$_j \geq y + 1$. Thus, eventually, for any two processes $p_j$ and $p_k$, $ts_j[k] \geq y + 1$. A contradiction.
From this lemma and the algorithm it follows:

**Lemma 5.3** Every strong operation that is invoked is executed by every process.

To prove that $\sigma$ is hybrid we first show that the strong operations can be linearized. Note that, by the algorithm, after a strong operation is timestamped, the value of the local timestamp counter, $\text{lt}_s$, is incremented by 1. Thus, we have:

**Lemma 5.4** Strong operations by the same process are assigned unique timestamps, according to their order of invocation.

**Lemma 5.5** Fix some local state $s$ of $p_i$ in $\sigma$, and assume $\text{lt}_s = z$, $z > 0$, in $s$. For every $0 < x \leq z$, there exists a unique previous local state of $p_i$ in $\sigma$ in which $p_i$ has sent either a strong-op message with timestamp $x - 1$, or a ts-update message with timestamp $x$; furthermore, $p_i$ never sends both.

**Proof:** By the algorithm, $\text{lt}_s = z$ only if it was incremented by one $z$ times, and had initial value of 0. Thus, for every value $x$ of $\text{lt}_s$, $0 < x \leq z$, $p_i$ incremented $\text{lt}_s$ from $x - 1$ to $x$ in one of the three cases stated in the proof of Lemma 5.1. In the first case, a strong-op message is sent with timestamp $x - 1$. In the other two cases, a ts-update message is sent with timestamp $x$. These are the only cases where strong-op and ts-update messages are sent, and they are disjoint.

Note that if $\text{lt}_s \geq x$ then $p_i$ does not send a strong-op message with timestamp $x - 1$.

**Lemma 5.6** Assume $t_{s_j}[i] = x$ in some state $s$ of $\sigma$. Then, in $s$, $\text{lt}_s \geq x$ and there are no strong-op messages with timestamp $x - 1$ that are in transit from $p_i$ to $p_j$.

**Proof:** The fact that if $t_{s_j}[i] = x$ then $\text{lt}_s \geq x$ follows immediately from the algorithm. Assume, by way of contradiction, that there is a strong-op message in transit from $p_i$ to $p_j$ with timestamp $x - 1$ in $s$. By Lemma 5.4 and the FIFO behavior of the links, and there is no previous state of $\sigma$ in which $p_j$ received a strong-op message from $p_i$ with timestamp $x - 1$. Since $t_{s_j}[i] = x$, $p_j$ must have received a ts-update from $p_i$ with $x$ in some previous state $s'$ of $\sigma$. Lemma 5.5 implies that there is no state of $\sigma$ in which $p_i$ sends a strong-op message with timestamp $x - 1$. A contradiction.

Since a process assigns distinct timestamps to strong operations it invokes, it follows that the pair (timestamp, initiator id) induces a total order on all strong operations; call this order the timestamp ordering.

**Lemma 5.7** Strong operations are executed by every process in timestamp ordering.
Proof: Assume, by way of contradiction, that there exist two strong operations \( s\text{op}_1 \) and \( s\text{op}_2 \), initiated by \( p_i \) with timestamp \( x_1 \), and \( s\text{op}_3 \), initiated by \( p_j \) with timestamp \( x_2 \); assume that \( s\text{op}_2 \) was executed before \( s\text{op}_1 \) in process \( p_k \), but \( (x_1, i) < (x_2, j) \). Let \( s \) be a state of \( \sigma \) in which process \( p_k \) has executed \( s\text{op}_2 \). By the algorithm, strong operations from the pending list are executed in increasing (timestamp, initiator id) order. Therefore, process \( p_k \) did not receive the strong-op message that corresponds to \( s\text{op}_1 \) before or in \( s \). By Lemma 5.6, in \( s \), \( ts_k[i] \leq x_1 \). Since \( x_1 \leq x_2 \), \( ts_k[i] \leq x_2 \). By the algorithm, \( s\text{op}_2 \) could have been executed only when for all \( m \), \( ts_k[m] > x_2 \). Thus, \( s\text{op}_2 \) was not allowed to be executed at \( p_k \) in \( s \). A contradiction. ■

Note, that for any two strong operations \( s\text{op}_1 \) and \( s\text{op}_2 \), if \( s\text{op}_1 \) is executed by some process before \( s\text{op}_2 \) then \( s\text{op}_2 \) cannot completely precede \( s\text{op}_1 \) in \( \sigma \). By this observation and Lemma 5.7, the timestamp order extends the real time order of strong operations in \( ops(\sigma) \). Moreover, all the strong operations is \( ops(\sigma) \) are executed in the same order by all processes. Let \( \rho \) be a linearization of \( ops(\sigma) \). This implies:

Lemma 5.8 \( \rho \) is an extension of the timestamp order.

Lemma 5.9 For any two processes \( p_i \) and \( p_j \), \( p_j \) executes all operations of \( p_i \) (strong ops and weak writes) in the order they were invoked.

Proof: Since we assumed FIFO links, and by the algorithm, the lemma holds for every pair of weak operations, or a weak operation followed by strong operation. As for strong operation before weak operation, by the algorithm, the weak operation is not invoked until the strong operation is executed by every process. ■

For the rest of the proof, fix an arbitrary process \( p_j \). Let the relation \( \xrightarrow{hybrid} \) on \( ops(\sigma) \) be the transitive closure of:

1. if \( op_j^1 \xrightarrow{ops(\sigma)} op_j^2 \) then \( op_j^1 \xrightarrow{hybrid} op_j^2 \).
2. if \( op_i^1 \xrightarrow{ops(\sigma)} op_i^2 \) and at least one of \( op_i^1 \) and \( op_i^2 \) is strong, for some process \( p_i \), then \( op_i^1 \xrightarrow{hybrid} op_i^2 \).
3. if \( op_i^1 \xrightarrow{\sigma} op_k^2 \) and both \( op_i^1 \) and \( op_k^2 \) are strong, for some processes \( p_i \) and \( p_k \), then \( op_i^1 \xrightarrow{hybrid} op_k^2 \).

This relation captures the requirements from a sequence \( \tau_j \) in the definition of hybrid consistency, except for being legal.

Lemma 5.10 \( \xrightarrow{hybrid} \) is an acyclic relation on \( ops(\sigma) \).
Proof: It suffices to show that \( \rightarrow^{\text{hybrid}} \) contains no cycles. Since \( \rho \) is a linearization of \( \sigma \), \( \text{op}^1 \) precedes \( \text{op}^2 \) in \( \rho \) only if \( \text{op}^2 \) does not precede \( \text{op}^1 \) in \( \text{ops}(\sigma) \). Moreover, there are no cycles in \( \sigma \). By the definition of \( \text{ops}(\sigma) \), an operation cannot precede itself. Thus, \( \rightarrow^{\text{hybrid}} \) contains no cycles.

Let the relation \( \rightarrow^{\text{legal}} \) on \( \text{ops}(\sigma) \) be the transitive closure of the minimal relation such that, for each \( r_k(x,v) \) in \( \sigma \) there exists a process \( p_i \) with the following properties:

a. \( w_i(x,v) \rightarrow^{\text{legal}} r_k(x,v) \), and
b. there is no process \( p_m \) and value \( u, u \neq v \) such that \( w_i(x,v) \rightarrow^{\text{legal}} w_m(x,u) \rightarrow^{\text{legal}} r_k(x,v) \).

This relation captures the requirements from a legal sequence.

Lemma 5.11 \( \rightarrow^{\text{legal}} \) is an acyclic relation on \( \text{ops}(\sigma) \).

Proof: By the algorithm, for every process \( p_k \), object \( x \) and value \( v \), if there exists a \( r_k(x,v) \), then there exists a \( w_l(x,v) \) for some process \( p_i \). Following this observation, all the operations can be grouped first by objects and then by values. In each group, all the writes are ordered before the reads, breaking further ties (inside the group) in any arbitrary way. Finally, we order all the groups in an increasing (object,value) order. Thus, there exists a total order on \( \text{legal} \) \( \text{ops}(\sigma) \) which obeys \( \rightarrow^{\text{legal}} \).

Lemma 5.12 For any two operations \( \text{op}^1 \) and \( \text{op}^2 \), if \( \text{op}^1 \rightarrow^{\text{hybrid}} \text{op}^2 \) then one of the following holds:

(i) \( k = l = j \) (\( p_j \) is the fixed process).
(ii) \( k = l \) and at least one of \( \text{op}^1 \) and \( \text{op}^2 \) is strong.
(iii) \( \text{op}^1 \rightarrow^{\text{hybrid}} \text{sop}_k \rightarrow^{\text{hybrid}} \text{sop}_l \rightarrow^{\text{hybrid}} \text{op}^2 \). (\( \text{op}^1 \) may be the same operation as \( \text{sop}_k \), \( \text{op}^2 \) may be the same operation as \( \text{sop}_l \).)

Proof: \( \text{op}^1 \rightarrow^{\text{hybrid}} \text{op}^2 \) can be derived by applying any finite composition of the rules in the definition of \( \rightarrow^{\text{hybrid}} \). This can be viewed as \( \text{op}^0 \rightarrow^{\text{hybrid}} \text{op}^1 \rightarrow^{\text{hybrid}} \cdots 
\rightarrow^{\text{hybrid}} \text{op}^F \), where \( \text{op}^0 = \text{op}^1 \) and \( \text{op}^F = \text{op}^2 \). The lemma is proved by showing that each possible composition will result in one of the cases listed in the lemma. This is done by induction on \( F \), the length of the composition.
For $F = 1$, these cases match the three rules in the definition of $\rightsquigarrow^{\text{hybrid}}$

Assume the lemma is correct for all $F < k$. We will now show that it is true for $F = k$. By the induction hypothesis, the $F-1$ leftmost rules can be represented as one of the cases listed in the lemma. It is left to be shown that the combination of the rightmost rule with any of the cases listed in the lemma, results in one of the cases listed in the lemma. This is done by checking every combination of a rule from the definition of $\rightsquigarrow^{\text{hybrid}}$ with a case from the lemma.

1. Rule 1 with Case (i). $op_0^{k_0}$ and $op_F^{k_F}$ are operations of process $p_j$. Therefore, this is Case (i).
2. Rule 1 with Case (ii). $op_0^{k_0}$ and $op_F^{k_F}$ are operations of process $p_j$. Therefore, this is Case (i).
3. Rule 1 with Case (iii). The rightmost strong operation in Case (iii) and $op_F^{k_F}$ must be operations of the same process. Therefore, this is Case (iii).
4. Rule 2 with Case (i). $op_0^{k_0}$ and $op_F^{k_F}$ are operations of process $p_j$. Therefore, this is Case (i).
5. Rule 2 with Case (ii). $op_{F-1}^{k_{F-1}}$ must be a strong operation. The $op_0^{k_0}$, $op_F^{k_F}$ and $op_{F-1}^{k_{F-1}}$ must be operations of the same process. Therefore, this is Case (iii).
6. Rule 2 with Case (iii). $op_F^{k_F}$ and the rightmost strong operation of Case (iii) must be operations of the same process. Therefore, this is Case (iii).
7. Rule 3 with Case (i). The leftmost strong operation in Rule 3 and $op_0^{k_0}$ must be operations of the same process. Therefore, this is Case (iii).
8. Rule 3 with Case (ii). $op_0^{k_0}$ and the rightmost operation of Rule 3 must be operations of the same process. Therefore, this is Case (iii).
9. Rule 3 with Case (iii). $op_F^{k_F}$ is a strong operation. Therefore, this is Case (iii).

Lemma 5.13 There are no cycles in the union of $\rightsquigarrow^{\text{hybrid}}$ and $\rightsquigarrow^{\text{legal}}$.

Proof: A cycle could have been created only if $\rightsquigarrow^{\text{hybrid}}$ forces sequences which are not legal. That is, it forces a sequence which corresponds to one of the following patterns:

(a) $r_k(x, v) \rightsquigarrow^{\text{hybrid}} w_l(x, v)$ for some processes $p_k$ and $p_l$, memory object $x$ and value $v$ such that for every $w_m(x, v)$ were $w_m(x, v) \rightsquigarrow^{\text{hybrid}} r_k(x, v)$ there exists $w_n(x, u)$ such that $u \neq v$ and $w_m(x, v) \rightsquigarrow^{\text{hybrid}} w_n(x, u) \rightsquigarrow^{\text{hybrid}} r_k(x, v)$, or
(b) \( w_k(x, v) \) \( \xrightarrow{\text{hybrid}} \) \( w_l(x, u) \) \( \xrightarrow{\text{hybrid}} \) \( r_m(z, v) \), for some processes \( p_k, p_l \) and \( p_m \), memory object \( z \) and values \( v \) and \( u \) and there is no \( w_n(x, v) \) such that \( w_l(x, u) \) \( \xrightarrow{\text{hybrid}} \) \( w_n(x, v) \) \( \xrightarrow{\text{hybrid}} \) \( r_m(z, v) \).

We derive a contradiction, by case analysis. We substitute each \( \xrightarrow{\text{hybrid}} \) in (a) and (b) with each one of the cases (i),(ii) and (iii) from Lemma 5.12. Therefore, \( \xrightarrow{\text{hybrid}} \) must force at least one of the \( 3 + 3 \times 3 = 12 \) patterns of subsequences listed below.

1. \( r_j(x, v) \) \( \xrightarrow{\text{hybrid}} \) \( w_j(x, v) \). (This is the combination of Case (i) with Case (a)). However, both the read and the write were executed by \( p_j \) in that order, and therefore the read could not have returned \( v \).

2. \( r_k(x, v) \) \( \xrightarrow{\text{hybrid}} \) \( w_k(x, v) \) and one of them is strong. (Case (ii) with Case (a)). However, both operations had to be executed by \( p_k \) in that order as well, by Lemma 5.9. Therefore, the read could not have returned \( v \).

3. \( r_k(x, v) \) \( \xrightarrow{\text{hybrid}} \) \( sop_k \) \( \xrightarrow{\text{hybrid}} \) \( sop_l \) \( \xrightarrow{\text{hybrid}} \) \( w_l(x, v) \). (Case (iii) with Case (a)). However, by Lemma 5.8 and Lemma 5.9, \( w_l(x, v) \) must have started after \( r_k(x, v) \) had finished. Therefore, the read could not have returned \( v \).

4. \( w_j(x, v) \) \( \xrightarrow{\text{hybrid}} \) \( w_j(x, u) \) \( \xrightarrow{\text{hybrid}} \) \( r_j(x, v) \). (Case (i) with both arrows of Case (b)). However, all those operations were executed in that order by \( p_j \), therefore the read could not have returned \( v \).

5. \( w_j(x, v) \) \( \xrightarrow{\text{hybrid}} \) \( w_j(x, u) \) \( \xrightarrow{\text{hybrid}} \) \( r_j(x, v) \) and at least one of \( w_j(x, u) \) or \( r_j(x, v) \) is a strong operation. (Case (i) to the first arrow of Case (b) and Case (ii) with the second arrow of Case (b)). Since they were all executed by \( p_j \), it is the same as Case 4.

6. \( w_j(x, v) \) \( \xrightarrow{\text{hybrid}} \) \( w_j(x, u) \) \( \xrightarrow{\text{hybrid}} \) \( sop_j \) \( \xrightarrow{\text{hybrid}} \) \( sop_k \) \( \xrightarrow{\text{hybrid}} \) \( r_k(x, v) \). (Case (i) with the first arrow of Case (b) and Case (iii) with the second arrow of Case (b)). However, by Lemma 5.8, Lemma 5.9 and the algorithm, the read in \( p_k \) must have been executed after both writes were executed, and these writes were executed in the same order as they appear in \( \xrightarrow{\text{hybrid}} \). Therefore, the read could not have returned \( v \).

7. \( w_j(x, v) \) \( \xrightarrow{\text{hybrid}} \) \( w_j(x, u) \) \( \xrightarrow{\text{hybrid}} \) \( r_j(x, v) \) and one of the writes is a strong operation. (Case (ii) with the first arrow of Case (b) and Case (i) with the second arrow of Case (b)). Since they were all executed by \( p_j \), it is the same as the Case 4.

8. \( w_k(x, v) \) \( \xrightarrow{\text{hybrid}} \) \( w_k(x, u) \) \( \xrightarrow{\text{hybrid}} \) \( r_k(x, v) \), at least one of \( w_k(x, v) \) and \( w_k(x, u) \) is strong and at least one of \( w_k(x, u) \) and \( r_k(x, v) \) is a strong operation. (Case (ii) with both arrows of Case (b)). However, by Lemma 5.8 and Lemma 5.9, these operations were executed in \( p_k \) in that order. Therefore, the read could not have returned \( v \).
9. \( w_k(x, v) \xrightarrow{\text{hybrid}} w_k(x, u) \xrightarrow{\text{hybrid}} s_{op_k} \xrightarrow{\text{hybrid}} s_{op_j} \xrightarrow{\text{hybrid}} r_l(x, v) \) and at least one of \( w_k(x, v) \) and \( w_k(x, u) \) is strong. (Case (ii) with the first arrow of Case (b) and Case (iii) with the second arrow of Case (b).) However, by Lemma 5.8, Lemma 5.9, and the algorithm, the read must have started after the both writes ended. Since at least one of the writes is strong, they were executed in \( p_i \) in that order. Therefore, the read could not have returned \( v \).

10. \( w_k(x, v) \xrightarrow{\text{hybrid}} s_{op_k} \xrightarrow{\text{hybrid}} s_{op_j} \xrightarrow{\text{hybrid}} w_j(x, u) \xrightarrow{\text{hybrid}} r_j(x, v) \). (Case (iii) with the first arrow of Case (b) and Case (i) with the second arrow of Case (b).) However, by Lemma 5.9, both writes were executed in \( p_j \) in that order, and the read was executed after both of them. Therefore, the read could not have returned \( v \).

11. \( w_k(x, v) \xrightarrow{\text{hybrid}} s_{op_k} \xrightarrow{\text{hybrid}} s_{op_l} \xrightarrow{\text{hybrid}} w_l(x, u) \xrightarrow{\text{hybrid}} r_l(x, v) \) and at least one of \( w_l(x, u) \) and \( r_l(x, v) \) is a strong operation. (Case (iii) with the first arrow of Case (b) and Case (ii) with the second arrow of Case (b).) However, by Lemma 5.8 and Lemma 5.9, all those operations were executed in \( p_i \) in that order, so the read could not have returned \( v \).

12. \( w_k(x, v) \xrightarrow{\text{hybrid}} s_{op_k} \xrightarrow{\text{hybrid}} s_{op_l} \xrightarrow{\text{hybrid}} w_l(x, u) \xrightarrow{\text{hybrid}} s_{op_m} \xrightarrow{\text{hybrid}} r_m(x, v) \). (Case (iii) with both arrows of Case (b).) However, by Lemma 5.8 and Lemma 5.9, all those operations were executed in \( p_m \) in that order. Therefore, the read could not have returned \( v \).

It follows that the transitive closure of the union of \( \xrightarrow{\text{hybrid}} \) or \( \xrightarrow{\text{legal}} \) is a partial order. ❄️

Let \( \tau_j \) be some linearization of the transitive closure of the union of \( \xrightarrow{\text{hybrid}} \) and \( \xrightarrow{\text{legal}} \). Clearly, \( \tau_j \) is legal and satisfies the conditions in the definition of hybrid consistency. This concludes the proof of correctness.

5.4 Complexity Analysis

5.4.1 Message complexity

Weak reads are executed locally without causing any message exchange. Each weak write generates \( n \) update messages. During the execution of a strong operation there are \( n \) strong-op messages and \( n \) ack messages. In addition, if the execution of this strong operation caused any process \( p_i \) to increase \( lts_i \), then \( p_i \) will send \( n - 1 \) ts-update messages. Thus, in the worst case, a strong operation generates up to \( (n - 1)^2 \) ts-update messages. Thus there are at most \( 2n + (n - 1)^2 = n^2 + 1 \) messages for each strong operation. However, this analysis does not capture the behavior of the algorithm completely. If more than one process invokes a strong operation at the same time, then these processes do not send many ts-update messages. This is formalized and proved below.
Lemma 5.14 For every state $s$ of $\sigma$, if for some process $p_i$, $lt_{s_i} > \min_j(t_{s_i[j]}[j]) + 1$, then there exists a previous state $s'$ in $\sigma$, such that $p_i$ invoked a strong operation with timestamp $lt_{s_i} - 1$ in $s'$.

Proof: Assume, by way of contradiction that there exists a state $s$ and a process $p_i$, in which $lt_{s_i} > \min_j t_{s_i[j]}$, but there is no previous state of $\sigma$ in which $p_i$ initiates a strong operation with timestamp $lt_{s_i} - 1$. Let $s''$ be the last state before $s$ in $\sigma$ in which $lt_{s_i}$ was incremented. Then, in $s''$ $lt_{s_i}$ was $\min_j(t_{s_i[j]}[j]) + 1$. Therefore, in $s''$, $\min_j(t_{s_i[j]}[j]) < lt_{s_i}$, and by the algorithm, $p_i$ could not increment $lt_{s_i}$ in $s''$. A contradiction. 

Lemma 5.15 In every state $s$ of $\sigma$, for every $i$, $1 \leq i \leq n$, $lt_{s_i} \leq \min_j(t_{s_i[j]}[j]) + 2$.

Proof: Denote $x = \min_j(t_{s_i[j]}[j])$. Assume, by way of contradiction, that there exists a state of $\sigma$ in which for some process $p_k$ $lt_{s_k} > x + 2$. Consider the earliest such state $s$ and let $p_k$ be a process for which $lt_{s_k} > x + 2$ in $s$. Then, according to Lemma 5.14, $p_i$ invoked a strong operation with timestamp $x + 2$ before $s$. From the assumption on $x$, there exists a process $p_j$ such that $lt_{s_j} = x$. By Lemma 5.6, $t_{s_i[j]}[j] \leq x$. Therefore, all strong operations with timestamp greater than $x$ were not executed in $p_i$ before or in $s$. Since a strong operation is not invoked before all previous operations by the same process have returned, and since $p_i$ has invoked a strong operation with timestamp $x + 2$, $p_i$ could not invoke any strong operation with timestamp $x$ or $x + 1$. Thus, $p_i$ incremented $lt_{s_i}$ from $x$ to $x + 1$ following the execution of a strong operation with timestamp $x$ or after receiving a $ts$-update message with timestamp $x + 1$. In both cases, by the algorithm, $t_{s_i[j]}[j] > x$. A contradiction.

Lemma 5.16 Let $T$ be the set of different values held by all the timestamp counters of all the processes in the system at a given state of $\sigma$. Then $|T| \leq 3$.

Proof: Assume that $x = lt_{s_i} = \min_k\{lt_{s_k}\}$. For any process $p_j$, $t_{s_j}[i] \leq x$, by Lemma 5.6. By Lemma 5.15, $lt_{s_j} \leq x + 2$. Since this holds for every $i$, it follows that $|T| \leq 3$.

Fix some system state $s$ in $\sigma$. Let $max_{lt}$ be the maximum over all the timestamps of strong-op messages sent until and in $s$, or $-1$ if none was sent. Let $max_{lt} = \max_i\{lt_{s_i}\}$.

Lemma 5.17 $max_{lt} = max_{lt} + 1$.

Proof: In the beginning of the execution this holds, since no strong-op messages were sent and for all $p_i$, $lt_{s_i} = 0$.

By Lemma 5.1, $lt_{s_i}$ can be incremented from $x - 1$ to $x$ only as a result of some strong operation with timestamp $x - 1$. Each time a strong-op message is sent by process $p_i$, it is timestamped by $lt_{s_i}$, and $lt_{s_i}$ is incremented by 1. Thus, the property is preserved.
Lemma 5.18 The timestamp of any pending operation is greater than or equal to \( \min \{ \{lts_i \} - 1 \).

From Lemmas 5.17, 5.18 and 5.16 it follows:

Lemma 5.19 Let \( T \) be the set of timestamps held by all pending operations in the system at a given state, \( s \), of \( \sigma \). Then \( |T| \leq 3 \).

Corollary 5.20 The total number of messages that are in transit at any state of \( \sigma \) is \( O(n^2) \).

Proof: By Lemma 5.19, the maximum difference between the timestamps held by all the pending strong operation in the system is 3. Thus, there could be ts-update messages for at most three different timestamps. Thus, there are at most \( 3(n - 1)^2 \) ts-update messages in transit in any state of \( \sigma \).

5.4.2 Time Complexity

We turn to analyze the time complexity of the algorithm, e.g., the maximum time required to complete any operation. For the purpose of time analysis, we assume that the time to execute an operation locally is negligible. Also, recall that the delay on the links is at most \( d \).

Clearly, \( |R| = |W| = 0 \), as weak operations are executed locally and instantaneously. For strong operations the analysis is slightly more complicated. Denote by \( t_{wait} \) the maximum time a strong operation may have to wait, from its arrival at a process, until it is executed locally.

Lemma 5.21 The time required to execute a strong operation, \( sop \), is at most \( 3d + t_{wait} \).

Proof: The strong-op message can take as long as \( d \) to reach all processes. The ts-update messages may take another \( d \) to arrive. The strong operation might be forced to wait \( t_{wait} \) until it is executed. The ack messages from all the other processes could take another \( d \) to be received by the initiator of \( sop \). The result is \( 3d + t_{wait} \).

Lemma 5.22 For each strong operation, \( t_{wait} \leq 2d \).

Proof: Let \( sop \) be a strong operation. If \( sop \) is not the only strong operation in the system, then \( sop \) might be forced to wait at \( p_i \) for other operations to execute locally. Let \( x \) be the timestamp of \( sop \). By Lemma 5.19, the smallest timestamp of any other pending strong operation in \( p_i \) is at least \( x - 2 \). Thus, there could be at most two more rounds of ts-update or strong-op messages in the system before \( sop \) could be executed. Thus, \( t_{wait} \leq 2d \).

From Lemma 5.21 and Lemma 5.22, we get the following theorem:

Theorem 5.23 The time required to execute a strong operation (a read or a write) is at most \( 5d \).
5.5 Improvements and Extensions

In this section we discuss possible optimizations to the algorithm.

First, it would be nice to remove the assumption that links are FIFO. In order to do so, some kind of FIFO simulation should be implemented for update messages, by timestamping them or by a ping-pong protocol. This should be done in order to avoid the following non-hybrid scenario

\[ w_i(x, v), w_i(x, u), sw_i(y, w), sr_j(y, w), r_j(x, v) , \]

which is otherwise possible. If update messages do not arrive in FIFO order, the update message of \( w_i(x, u) \) can bypass the update message of \( w_i(x, v) \). Thus, the last value written to \( x \) in the local copy of memory of \( p_j \) can be \( v \). A similar mechanism should be applied to make sure a weak operation invoked after a strong operation is not executed before the strong operation. This will complicate the algorithm and its proof of correctness.

Second, it is desirable to extend the implementation to systems where each process holds only a partial copy of the memory (e.g., directory-based schemes). Reads and writes may be directed to the nearest copy, and update messages sent only to processes holding copies of the object being modified. In strong operations, the strong-op messages will be sent to all processes. Processes which do not hold the relevant copy of memory will treat those messages like strong read; they will execute them by doing nothing. Maintaining directories of owners and avoiding irrelevant messages seems to be more complicated.

Lastly, it would be interesting to incorporate some sort of pipelining and allow weak operations to start before a previous strong operation had finished. In this case, weak operations should be suspended until all strong operations with smaller timestamps invoked by the same process are executed. It is not clear what are the performance benefits of such improvements.

6 Lower Bounds for Implementing Hybrid Consistency

It is known [5] 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. It is proven in [5] that sequential consistency can be implemented more efficiently than linearizability. This might lead one to think that hybrid consistency where the strong operations are sequentially consistent can be implemented more efficiently. 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 [23].

**Theorem 6.1** For any implementation of hybrid consistency, \( |R| + |SW| \geq d \).
Proof: Assume, by way of contradiction, that there is an implementation of hybrid consistency for which $|R| + |SW| \leq d$. 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 some admissible execution $\sigma_1$ such that $ops(\sigma_1)$ is

$$SWrite_1(x, 1) SAck_1(x) Read_1(y) Return_1(y, 0),$$

$SWrite_1(x, 1)$ occurs at real time 0 and $Read_1(y)$ occurs immediately after $SAck_1(x)$. 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 admissible execution $\sigma_2$ such that $ops(\sigma_2)$ is

$$[SWrite_2(y, 1), SAck_2(y)], [Read_2(x), Return_2(x, 0)],$$

$SWrite_2(y, 1)$ occurs at real time 0 and $Read_2(x)$ occurs immediately after $SAck_2(y)$. 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 $p_2$'s history with $p_2$'s history in $\sigma_2$ can be extended to an admissible hybrid execution $\sigma$. Then $ops(\sigma)$ consists of the operation $[SWrite_1(x, 1), SAck_1(x)]$ followed by $[Read_1(y), Return_1(y, 0)]$, and $[SWrite_2(y, 1), SAck_2(y)]$ followed by $[Read_2(x), Return_2(x, 0)]$.

Since $\sigma$ is hybrid, there exists a serialization $\rho$ of $ops(\sigma)$, such that for every $i \in \{1, 2\}$, there exists a legal serialization $\tau_i$ of $ops(\sigma)$, 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 $\rho$, $[SWrite_1(x, 1), SAck_1(x)]$ precedes $[SWrite_2(y, 1), SAck_2(y)]$. 

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 $\tau_2$, $[Read_2(x), Return_2(x, 0)]$ precedes $[SWrite_1(x, 1), SAck_1(x)]$, which, in turn, precedes $[SWrite_2(y, 1), SAck_2(y)]$. Hence, in $\tau_2$, $[Read_2(x), Return_2(x, 0)]$ precedes $[SWrite_2(y, 1), SAck_2(y)]$, but then $\tau_2 \not\preceq \sigma \not\preceq 2$. A contradiction. 

The interested reader can compare this proof with the proof of the similar result in [5, Theorem 3.1], and note 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, $|W| + |SR| \geq d$. 

28
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 \( |SW| \geq d/2 \).

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

### 7 Weak Consistency

This section motivates hybrid consistency. A fundamental coordination problem which has a solution based on hybrid consistency, is proved to be unsolvable if the only guarantee about the ordering of operations is 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 \( \tau_j \mid j = \sigma \mid j \).

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

An implementation of weak consistency is provided by the Pipelined RAM model of Lipton and Sandberg ([23]). In this implementation, each process holds a complete copy of all the 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 ([23]) 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 $exc(A)$.

**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 $exc(A)$; otherwise, the algorithm is cooperative.

Note that all the solutions in which a process wishing to enter the critical section must communicate with some subset of the processes, regardless of whether they are trying to enter (or inside) the critical section or not are cooperative. In particular, centralized solutions (like the one presented in [23]) 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 is cooperative.

**Proof:** Consider a system with two processes, 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. Let $X$ be the exclusion set of $A$, and let $p_1$ and $p_2$ be the processes running $A$.

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 $p_2$ is in the remainder section, and since $A$ is non-cooperative, $p_2$ does not write to any variable in $X$ 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 $X$ while $p_1$ only reads variables from $X$, 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 $\sigma_2'$, $p_2$ and $p_1$ 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 $X$ 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 $X$ while $p_2$ only reads variables from $X$, it follows that $p_3$'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 $p_2$'s history with $p_3$'s history in $\sigma_2$ can be extended to an admissible 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, an undesirable property.

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.

### 7.1 Other Weak Consistency Models

In this section we present other weak consistency conditions known to us. Some of these models are weaker than the model we have already presented and some are stronger. We show that the impossibility result holds for all of them.

**Afek, Brown and Merritt**

The following definition by Afek, Brown and Merritt was given as Definition 3 in [1]:

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

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

This definition may seem very similar to Definition 7.1, given above. However, in our definition, every process must eventually see all the operations of all the other processes. In contrast, according to the definition of [1], a process need only see its own operations. It is trivial that the impossibility result holds for this definition, too.
Processor Consistency

We have no knowledge about any formal definition of processor consistency. Intuitively [17], processor consistency requires that the operations of each process will appear to be executed by all the processes in the order they were invoked. 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 \( p_i \) 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 [23] is processor consistent. Thus, the impossibility result holds for processor consistency.

Causal Memory

A consistency condition called causal memory was suggested in [4]. An operation \( op^1_p \) precedes \( op^2_q \) (denoted \( op^1_p \rightarrow op^2_q \)) if either \( p = q \) and \( op^1_p \) was invoked before \( op^2_q \), or \( op^1_p = wp(x, v) \) for some memory location \( x \) and value \( v \) and \( op^2_q = r_p(x, v) \). Let \( \rightarrow \) denote the transitive closure of \( \rightarrow \); \( op^1_p \) and \( op^2_q \) are concurrent if \( op^1_p \nRightarrow op^2_q \).

Definition 7.4 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 7.5 (Causal consistency) An admissible 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 holds for the implementation of causal memory given in [4], with the minor modification that the frequency in which discard is called is also \( d \). Thus, the impossibility result holds for causal memory, too.

\(^4\)This definition assumes the no value is written twice to the same memory location.
8 Compositionality of Various Consistency Conditions

Compositionality was first defined by Herlihy and Wing ([18]). A consistency condition $C$ is compositional 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 execution $\sigma$, $\sigma$ satisfies $C$ if and only if $\sigma \mid X$ satisfies $C$, for every object $X$.

It is shown in [18] 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

$$\text{ops}(\sigma) = r_1(x, 1), sr_1(y, 0), sw_2(y, 1), w_2(x, 1).$$

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 $(\tau_j \mid x) \mid j = (\sigma \mid x) \mid j$.

The execution $\sigma \mid x$ is weakly consistent if, for every process $p_j$, there exists a legal serialization $\tau^*_j$ of $\sigma \mid x$ such that $\tau^*_j \mid j = (\sigma \mid x) \mid j$. Thus, any merging of $\tau^*_j$, for any object $x$, 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 one of then is compositional.

**Theorem 8.3** Causal Memory is not compositional.
and application program; that is, extending our definition to allow an application program to have more than one operation pending at a time. Implementations which allow this sort of consistency faster than algorithms for solving the same problem that rely on other consistency conditions? If so, how much? This can be either by theoretical or experimental methods.

In other words, we would like to check how the labeling of memory accesses in a given program affects its run time.

hybrid consistency. We intend to use this implementation to check the relation between the way memory accesses are labeled (as either weak or strong) and the performance of the system.

This paper presents a theoretical study of hybrid consistency. This study is motivated by recent 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 another based on linearizability. We have presented an algorithm to support 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 delay weak operations (which are within constant factors of the bounds achieved by our algorithm).

To complement this theoretical study, we have developed a software implementation of hybrid consistency. We intend to use this implementation to check the relation between the way memory accesses are labeled (as either weak or strong) and the performance of the system. In other words, we would like to check how the labeling of memory accesses in a given program affects its run time.

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.

It would be interesting to quantify what are the performance benefits of programming with hybrid consistency. That is, are algorithms that solve specific problems that use hybrid consistency faster than algorithms for solving the same problem that rely on other consistency conditions? If so, how much? This can be either by theoretical or experimental methods.

A very interesting issue for further research is pipelining at the interface between the mcs and application program; that is, extending our definition to allow an application program to have more than one operation pending at a time. Implementations which allow this sort

Proof: Consider the following execution

\[ \sigma = r_1(y, 5), w_1(z, 2), r_2(z, 2), w_2(y, 5). \]

Note that

\[ \sigma \mid x = w_1(z, 2), r_2(z, 2) \]
\[ \sigma \mid y = r_1(y, 5), w_2(y, 5). \]

\( r_1(y, 5) \) must be ordered after \( w_2(y, 5) \), which is possible in the projection for \( y \). Thus, \( \sigma \mid y \) is a causally consistent execution. Similarly, \( r_2(z, 2) \) must be ordered after \( w_1(z, 2) \), 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 \( r_2(z, 2) \) must be ordered before \( w_2(y, 5) \) and \( r_1(y, 5) \) must be ordered before \( w_1(z, 2) \) are added. Together, these four requirements create a cycle. Thus, \( \sigma \) is not a causally consistent execution.

9 Discussion and Further Research

This paper presents a theoretical study of hybrid consistency. This study is motivated by recent 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 another based on linearizability. We have presented an algorithm to support 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 delay weak operations (which are within constant factors of the bounds achieved by our algorithm).

To complement this theoretical study, we have developed a software implementation of hybrid consistency. We intend to use this implementation to check the relation between the way memory accesses are labeled (as either weak or strong) and the performance of the system. In other words, we would like to check how the labeling of memory accesses in a given program affects its run time.

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.

It would be interesting to quantify what are the performance benefits of programming with hybrid consistency. That is, are algorithms that solve specific problems that use hybrid consistency faster than algorithms for solving the same problem that rely on other consistency conditions? If so, how much? This can be either by theoretical or experimental methods.

A very interesting issue for further research is pipelining at the interface between the mcs and application program; that is, extending our definition to allow an application program to have more than one operation pending at a time. Implementations which allow this sort
of pipelining are likely to be more efficient. This property has been explored for sequential consistency, and is supported by most new processors.

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. However, these proofs 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 [31].)

The ultimate goal of this research is to acquire a better understanding of memory consistency conditions, so we would be able 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 earlier versions of this paper. Thanks also to Sarita Adve, Eran Aharonson, Kourosh Gharachorloo, Maurice Herlihy, Martha Kosa, Rivka Ladin, Friedemann Mattern and Tali Shintel for helpful remarks.
References


