A WEAKER HYBRID CONSISTENCY CONDITION FOR SHARED MEMORY OBJECTS

An Undergraduate Research Scholars Thesis

by

SHIHUA ZHENG

Submitted to Honors and Undergraduate Research
Texas A&M University
in partial fulfillment of the requirements for the designation as

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Research Advisor: Jennifer L. Welch

December 2014

Major: Computer Science

TABLE OF CONTENTS

ABSTRACT

A Weaker Hybrid Consistency Condition for Shared Memory Objects. (December 2014)

Shihua Zheng
Department of Computer Science and Engineering
Texas A&M University

Research Advisor: Dr. Jennifer L. Welch
Department of Computer Science and Engineering

A consistency condition defines the behavior of a shared data object in both parallel and sequential operations. Hybrid consistency is a consistency condition that allows strong consistency and weak consistency to exist together. In hybrid consistency, the operators are classified as strong or weak. The operators with different consistency levels have different amounts of restrictions on the ordering. It tends to obtain both high performance and strong consistency level. A widely used condition in the industry that provides a very weak guarantee is eventual consistency. This thesis proposes a new consistency condition by relaxing hybrid consistency. The proposed hybrid eventual consistency condition allows weak operations more flexibility. An algorithm is proposed to implement hybrid eventual consistency in a message-passing system. In the simulation, we evaluate its average performance. Our results show that it produces a higher throughput and lower latency than hybrid consistency.

ACKNOWLEDGMENTS

I sincerely thank my research advisor, Dr. Jennifer L. Welch. Her guidance of my research is the key to this result. She gave me great insights that enabled me to keep making progress. I feel nothing but fortunate to have Dr. Welch as my advisor. Also, I would appreciate all other members of the Distributed Computing Group. I have learned a lot from you since I joined the group two years ago.

Finally, I thank my parents for their always support in my life.

CHAPTER I

INTRODUCTION

Background

The limitation of a single machine on its computational capacity leads to the development of distributed computing. Message passing and shared memory are two dominant communication models in distributed computing systems. Shared memory, which allows operations from different processors to be performed on a single shared object, is a high-level abstraction. It makes the programming easier because the manipulation of data on a shared memory object is very similar to doing the same on a sequential machine. Sometimes, physically sharing a single centralized memory is not efficient or feasible. Distributed shared memory (DSM) allows the system to be built on a message passing model to simulate a shared memory system. Several DSM systems have been implemented by researchers, including Munin (Bennett et al. [1]), TreadMarks (Amza et al. [2]) and Midway (Bershad et al. [3]).

For performance considerations, we could allow operations to be performed concurrently. However, a consistency condition is needed to specify the behaviour of the shared memory object when operations overlap. Atomicity (Lamport [4]), also known as linearizability (Herlihy and Wing [5]), and sequential consistency (Bennett et al. [1], Lamport [6], Scheurich and Dubois [7]) are two well-studied strong consistency conditions. Given that implementing a strong consistency condition is usually costly (Attiya and Welch [8], Gupta et al. [9]) and more than needed, researchers have made many attempts to discover new consistency conditions, for example, PRAM consistency (Lipton and Sandberg [10]), cache consistency (Goodman [11]), processor consistency (Goodman [11]), causal consistency (Ahamad et al. [12]) and local consistency (Bataller and Bernabeu [13]). In addition, single-writer regularity (Lamport [4]) and multi-writer regularity (Shao et al. [14]) were proposed to set a standard for a weak but well-behaved correctness condition.

Eventual consistency (Terry et al. [15], Burckhardt et al. [16]) is a widely used consistency condition, especially in geographically distant distributed database systems (DeCandia et al. [17]). Eventual consistency requires that, if updates stop being invoked, the shared memory will eventually reach a consistent state.

Hybrid consistency (Attiya and Friedman [18]) was proposed in 1992. In hybrid consistency, each operation is assigned a consistency level, either strong or weak. Informally, strong operations are ordered in some sequential order. The ordering of a strong operation and a weak operation executed on the same process is preserved. Hybrid consistency allows weak operations to be reordered between two adjacent strong operations. However, if strong operations are invoked frequently, weak operations might essentially have no flexibility to be reordered, resulting in performance issues. Formally, various theoretical lower bounds relating to hybrid consistency have been found. It has been proven that, in many cases, the overhead of using hybrid consistency is not better than using linearizability (Kosa [19]).

Overview

This research proposes a consistency condition for arbitrary data types by modifying hybrid consistency. Like hybrid consistency, the proposed consistency condition requires all operations to be defined as either strong or weak. Strong operations are still well-ordered as in hybrid consistency. Weak operations are less restrictive. It is no longer required that the weak operations cannot be reordered further than a future strong operation. This change makes the weak operations behave similarly to eventual consistency.

We will give a formal definition of hybrid eventual consistency condition in Chapter 2. Here we give the important properties of the proposed consistency condition:

1. Globally, strong operations appear to be executed in a linearizable order.
2. All weak operations will be eventually observed by all processes.

Similar to linearizability and hybrid consistency, the first property guarantees that all strong operations must observe all preceding strong operations. Because the execution is infinite, this property promises that all weak operations will be eventually observed by all other operations. Hence, the weak operations meet the informal definition of the eventual consistency condition. We therefore obtain both the strong consistency and the eventual consistency in one consistency condition.

The performance to implement hybrid eventual condition differs with respect to the data type. An analysis of the proposed consistency condition reveals that it could be faster than linearizability when the data type has several properties. An algorithm is given to show that strong operations are quicker than the ones of hybrid consistency.

Organization

Chapter II contains a formal definition of the proposed consistency condition. Chapter III discusses and proves the bounds relating to the proposed consistency condition. Chapter IV gives an algorithm to implement the new consistency condition and proves the correctness of the algorithm. Chapter V evaluates the performance of the algorithm by comparing the simulation results of hybrid eventual consistency and hybrid consistency, as well as linearizability and sequential consistency, for a distributed shared register. Chapter VI concludes this thesis.

CHAPTER II

CONCEPTS AND DEFINITIONS

System model

In this thesis, we consider a virtual shared memory system. The system consists of n nodes. The nodes communicate by sending messages though an asynchronous inter-connection network. This network never loses any messages. In addition, we assume that there is an upper bound d for the message delay on this network. The nodes in this network form a complete graph, which means that any pair of nodes in the system are directly connected.

At each node, a copy of the shared memory object, along with other information needed to perform synchronization, is stored. A *VSM process* is running on every node. These processes simulate one single shared memory object by providing an interface that takes invocations of operations on the underlying shared object. The VSM process performs the operations by reading its local state and communicating with other nodes (if necessary). Afterwards, the corresponding response is returned. In addition, communications may take place even if there are no active operations.

Related concepts

A *VSM event* is a 4-tuple $(i, t_{invoke}, t_{return}, r)$, where p_i is the index of the VSM node, t_{invoke} is the time an operation is invoked, t_{return} is the time a response is returned and r is the response. A VSM event describes an operation by the application to access the shared memory object. A set of VSM events is called a *VSM execution*. The message events initated by the VSM processes are referred as *low-level events*.

A consistency condition defines the correct behavior of the program.

Definition II.0.1 *(Legal serialization). A serialization of an execution* σ *is a permutation of the operations of* σ*. A serialization* ρ *is legal if and only if it is permissible according to the specification of the data structure.*

Definition II.0.2 *(Partial order). Let* α *and* β *both be VSM events.* $\alpha < \beta$ *if and only if* t_{return} *of* β *is larger than* t_{invoke} *of* α*.*

Definition II.0.3 *(Linearizability). An execution* ρ *is linearizable if there exists a legal serialization* σ *of* ρ *such that: For any two events* α *and* β *of* ρ*,* $\alpha < \beta$ *implies that* α *precedes* β *in* σ*.*

The principle of hybrid consistency and the proposed hybrid eventual consistency is to treat operations in different ways. For this purpose, all the operations are marked either strong or weak. The operations marked as strong have better properties than weak ones.

Definition II.0.4 *(Hybrid consistency). An execution* ρ *is hybrid consistent if there exists a serialization* σ *of the strong operations of* ρ *such that for each process* p_i*, there exists a legal sequence of operations* (τ_p) *such that:*

1. τ_p *is a permutation of the operations of* p*.*
2. *If* op_1 *and* op_2 *are both executed by the process* p_i*,* $op_1 < op_2$ *in* ρ *and at least one of* op_1 *and* op_2 *is strong, then* op_1 *precedes* op_2 *in* τ_p*.*
3. *If* op_1 *precedes* op_2 *in* σ *and* op_1 *and* op_2 *are both strong, then* op_1 *precedes* op_2 *in* τ_p*.*
4. $\tau | i = \rho | i$*.*

Eventual consistency is not an accurately defined term. It generally means that in an updatable replicated database, eventually all copies of each data item converge to the same value (Bernstein and Das [20]).

Definition of hybrid eventual consistency

The implementation of hybrid consistency widely uses the atomic broadcasting (Attiya and Friedman [18]), which is costly considering that the weak operations typically do not need much consistency. Therefore, we propose hybrid eventual consistency condition. By giving weak operations more flexibility, they could obtain some performance benefits from eventual consistency.

Definition II.0.5 *(Hybrid eventual consistency). An execution ρ is hybrid eventually consistent if there exists a serialization σ of the strong operations of ρ such that for each process p_i, there exists a legal sequence of operations (τ_p) such that:*

1. *τ_p is a permutation of the operations of p.*
2. *If op_1 precedes op_2 in σ and op_1 and op_2 are both strong, then op_1 precedes op_2 in τ_p.*
3. *$\tau|i = \rho|i$.*

CHAPTER III

LOWER BOUNDS FOR HYBRID EVENTUAL CONSISTENCY

Concepts

We, at first, introduce some concepts, including the commutativity, cyclic dependency and interleavability. Then, we will give proof of some properties for hybrid eventual consistency. The proof techniques we use are similar to Kosa [19]. The proof heavily relies on the fact that there must be a legal serialization of all operations. Although the proposed consistency condition is weaker, the lower bounds are also the same.

Definition III.0.6 *(Do not commute) Let OP_1 and OP_2 be two operations. OP_1 and OP_2 do not commute if there exists a sequence of operations α, an instance of each operation op_1 and op_2 such that both $\alpha \circ op_1$ and $\alpha \circ op_2$ are legal and:*

1. *$\alpha \circ op_1 \circ op_2$ is not legal, or*
2. *$\alpha \circ op_2 \circ op_1$ is not legal, or*
3. *There exists a sequence of operations β such that $\alpha \circ op_1 \circ op_2 \circ \beta$ is legal and $\alpha \circ op_2 \circ op_1 \circ \beta$ is not legal, or*
4. *There exists a sequence of operations β such that $\alpha \circ op_2 \circ op_1 \circ \beta$ is legal and $\alpha \circ op_1 \circ op_2 \circ \beta$ is not legal.*

Definition III.0.7 *(Immediately do not commute) Let OP_1 and OP_2 be two operations. OP_1 and OP_2 immediately do not commute if there exists a sequence of operations α, an instance of each operation op_1 and op_2 such that both $\alpha \circ op_1$ and $\alpha \circ op_2$ are legal and:*

1. *$\alpha \circ op_1 \circ op_2$ is not legal, or*
2. *$\alpha \circ op_2 \circ op_1$ is not legal.*

Definition III.0.8 *(Cyclic dependent) Let OP_1 and OP_2 be two operations. OP_1 and OP_2 are cyclic dependent if there exists a sequence of operations α, an instance of each operation op_1 and op_2 such that both $\alpha \circ op_1$ and $\alpha \circ op_2$ are legal and:*

1. *$\alpha \circ op_1 \circ op_2$ is not legal, and*
2. *$\alpha \circ op_2 \circ op_1$ is not legal.*

Definition III.0.9 *(n-cyclic dependent) A set of n operations, $OP_1 ... OP_n$ are n-cyclic dependent if there exists a sequence of operations α, an instance of each operation op_i ($i = 1...n$) such that:*

1. *For any $i = 1...n$, $\alpha \circ op_i$ are legal, and*
2. *For any permutation β of $op_1 ... op_n$, $\alpha \circ \beta$ is not legal.*

Definition III.0.10 *(Doubly non-interleavable) Let AOP, OP_1 and OP_2 be three operations. OP is doubly non-interleavable with respect to OP_1 and OP_2 if there exists a sequence of operations α, an instance of operations op_1, op_2, Aop^1 and Aop^2, where Aop^1 and Aop^2 are instances of AOP and op_1 and op_2 are instances of OP_1 and OP_2 respectively, such that:*

1. *$\alpha \circ op_1 \circ Aop^1$ is legal, and*
2. *$\alpha \circ op_2 \circ Aop^2$ is legal, and*
3. *If we place both Aop_1 and Aop_2 after α in $\alpha \circ op_1 \circ op_2$, it must be illegal, and*
4. *If we place both Aop^1 and Aop^2 after α in $\alpha \circ op_2 \circ op_1$, it must be illegal.*

An example

We consider a widely used data structure, the FIFO queue. Assume that the FIFO queue has three operations, *push* (enqueue), *pop* (deque) and *front* (return the front object without poping). Its sequential specification is clearly defined.

Let $\alpha = push(1)$, and $op_1 = op_2 = pop(1)$. Then, $\alpha \circ op_1 = \alpha \circ op_2 = push(1) \circ pop(1)$ are both legal.

However, $\alpha \circ op_1 \circ op_2 = \alpha \circ op_2 \circ op_1 = push(1) \circ pop(1) \circ pop(2)$ are both illegal.

Therefore, *push* and *pop* of FIFO queue do not commute, immediately do not commute and are cyclic dependent (*see* Definition III.0.6, Definition III.0.7 and Definition III.0.8).

Let β be empty, $op_1 = pop(\phi)$, $op_2 = front(\phi)$, $Aop^1 = push(2)$ and $Aop^2 = push(3)$.

Then, $\beta \circ op_1 \circ Aop^1 = pop(\phi) \circ push(2)$ is legal. $\beta \circ op_2 \circ Aop^2 = front(\phi) \circ push(3)$ is also legal.

If we put both Aop^1 and Aop^2 before, we must pop either 2 or 3. However, op_1 and op_2 implies that the queue must be empty. We cannot put both Aop^1 and Aop^2 before $\beta \circ op_1 \circ op_2$ and $\beta \circ op_2 \circ op_1$ and it is still legal. Therefore, *pop* is doubly non-interleavable with respect to *push* and *front*.

Hybrid eventual consistency

Next, we give some lower bounds for the proposed hybrid eventual consistency condition. We assume a perfectly synchronized clock.

Theorem III.0.1 *If every operation of the data type has a strong version, then, for any operation OP that immediately does not commute with itself, we have* $|OP| \geq d$ *for hybrid eventual consistency.*

Proof Suppose, in contradiction, that there exists such OP such that $|OP| < d$.

Because OP immediately does not commute, there is a sequence ρ of operations and an operation instance op, such that $\rho \circ op$ is legal but $\rho \circ op \circ op$ is illegal.

We build two admissible executions of two processors as follows, based on ρ.

α_1: Invoke all but the last operations of ρ sequentially on p_1. After completion, invoke the last operation of ρ on p_2. All operations complete at time t_0. At time t_1 ($t_1 > t_0 + \epsilon$), invoke a strong instance op in p_1. This op will terminate before $t_1 + d$.

α_2: α_2 is constructed similarly to α_1, but *op* is invoked in p_2. This *op* will also complete before $t_1 + d$.

We then combine α_1 and α_2, forming α_3. Since the completion of both *op* takes less than d time, both p_1 and p_2 observe no difference in this execution. It will do the same thing as before.

However, we must have a legal linearization of strong *ops* (Condition 4 of hybrid eventual consistency). It could be only $p \circ op \circ op$, which is not legal. This contradicts the admissible condition. ■

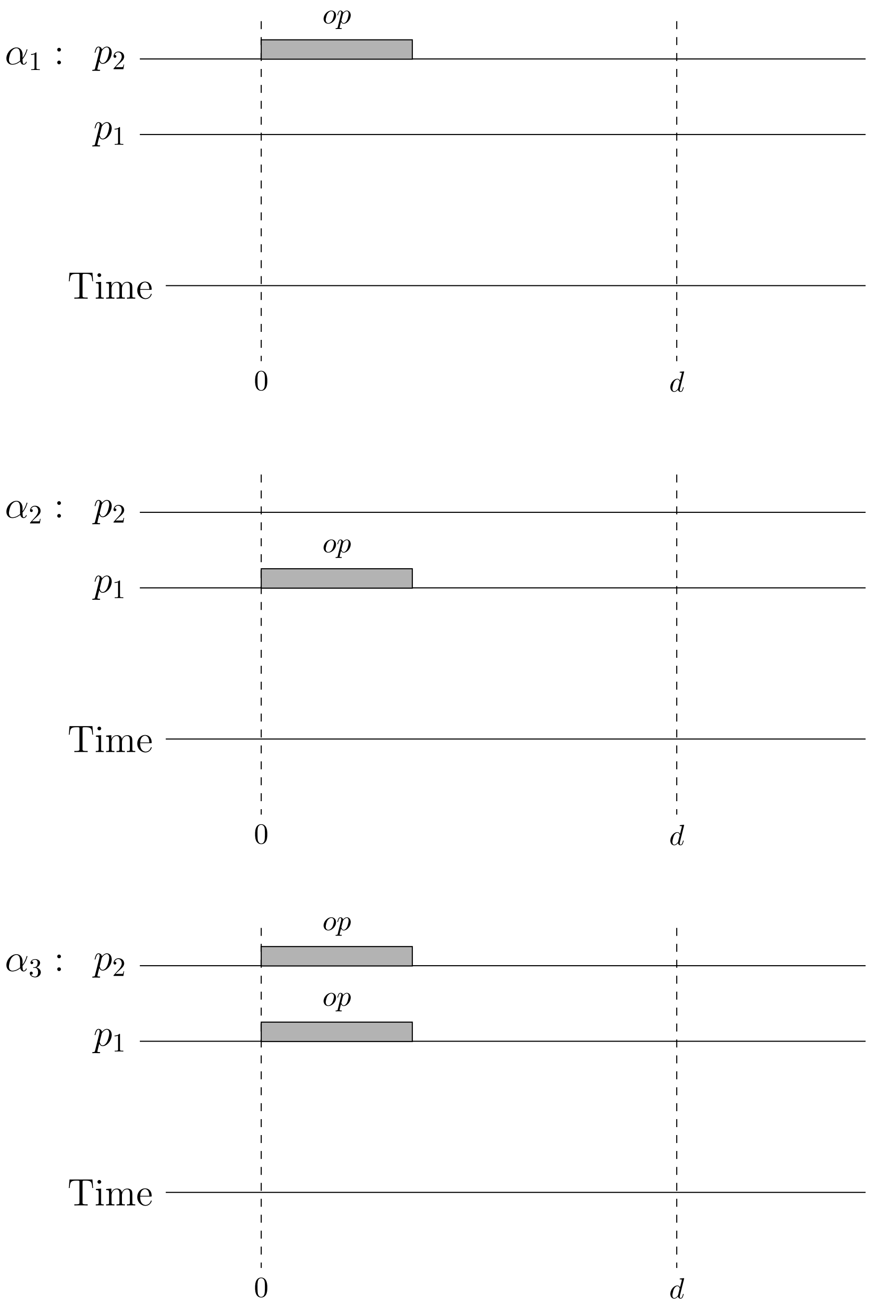

Fig. III.1. Counterexample in the proof of Theorem III.0.1

Theorem III.0.2 *If every general operation of the data type has a strong version, then, for a series of n-cyclic dependent operations OP_1 to OP_n, we have at least one operation OP_i such that $|OP_i| \geq d$ for hybrid eventual consistency.*

Proof The proof is an expansion of Theorem III.0.2.

Suppose, in contradiction, that there exists such operations $OP_1 ... OP_n$ such that $|OP_i| < d$ for any i.

We construct n executions $\alpha_1 ... \alpha_n$ of n processes like the one in Theorem III.0.1. Afterwards, we merge these executions to generate a new execution α. n processors will not perceive the difference between the old executions and the new execution. Therefore, it will behave the same as α_1 through α_n. However, because of the n-cyclic dependence, we cannot have an admissible execution. This contradicts the assumption. ■

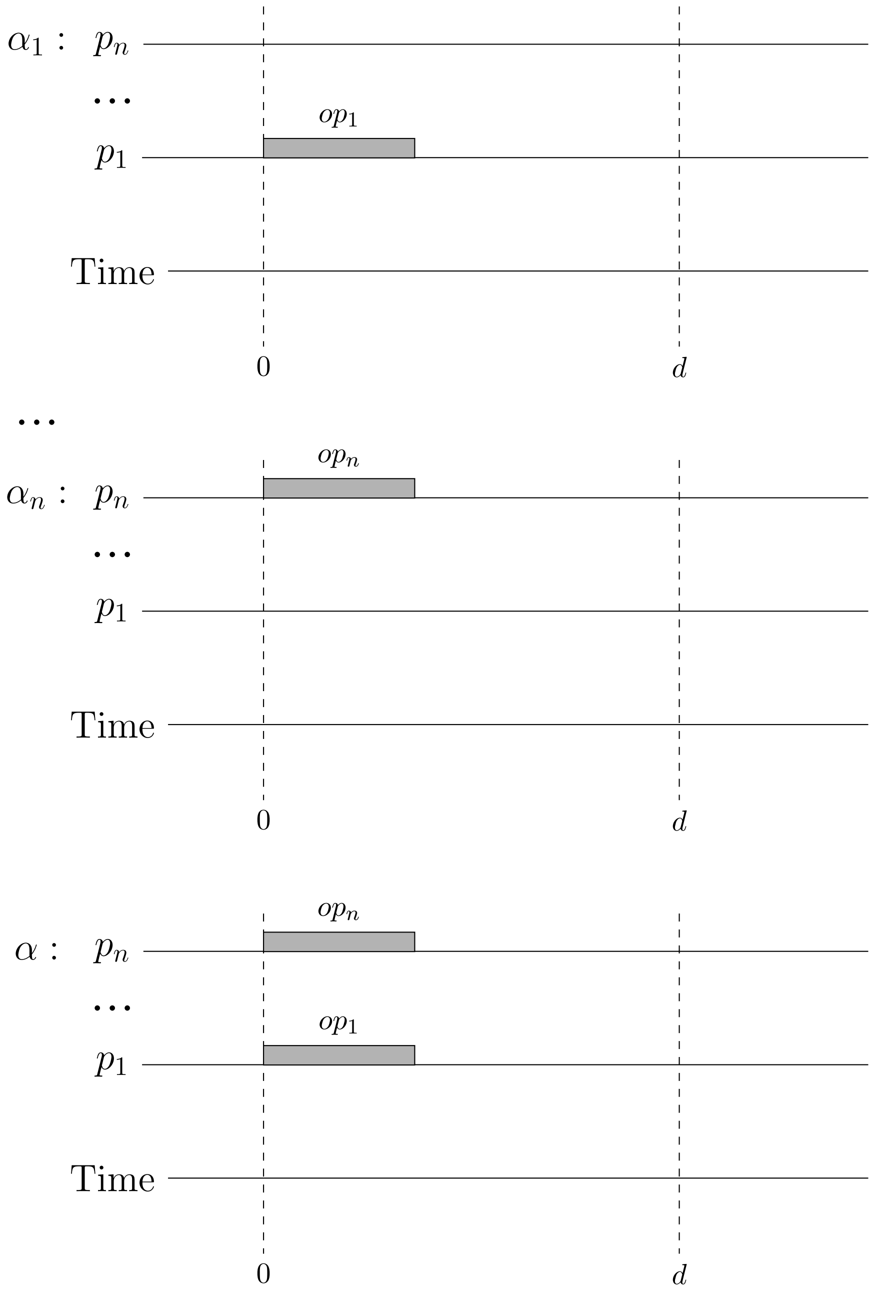

Fig. III.2. Counterexample in the proof of Theorem III.0.2

Theorem III.0.3 *If every general operation of the data type has a strong version, then, for 3-cyclic dependent operations* OP_1*,* OP_2 *and* OP_3*, we have* $|OP_1| + |OP_2| + |OP_3| \geq 2d$ *for hybrid eventual consistency.*

Proof Suppose, in contradiction, that there exists such operations OP_1, OP_2 and OP_3 such that $|OP_1| + |OP_2| + |OP_3| < 2d$.

From Theorem III.0.2, we already know that $|OP_1|$, $|OP_2|$ and $|OP_3|$ cannot be all smaller than d. We assume, without loss of generality, that $|OP_1| \geq d$ and $|OP_2| + |OP_3| < d$.

We construct 3 executions α_1, α_2 and α_3, each of which contains 3 processes. We then merge these executions to generate a new execution α. Because no processors will perceive the difference between the old executions and the new execution, the new execution α must be legal. Let ρ be the initial state of op_1 through op_3. Therefore, there is a permutation β of op_1, op_2 and op_3 such that $\rho \circ \beta$ is legal.

Since OP_1, OP_2 and OP_3 are 3-cyclic dependent, there exists a ρ such that no $\rho \circ \beta$ is legal. This contradicts the assumption.

■

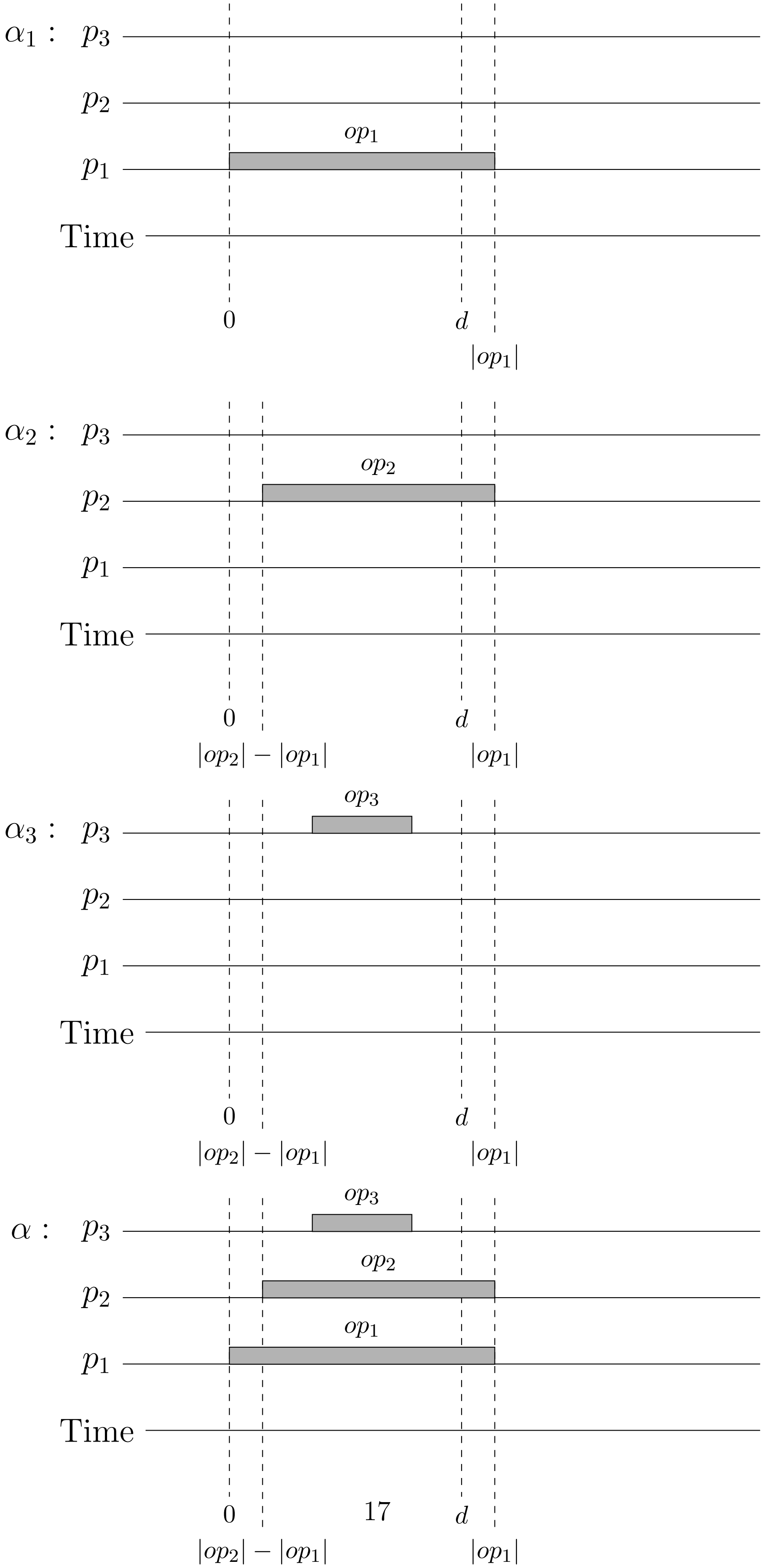

Fig. III.3. Counterexample in the proof of Theorem III.0.3

Theorem III.0.4 *Assume that every general operation of the data type has a strong version. If AOP is doubly non-interleavable with respect to OP_1 and OP_2, $|WOP_1| + |WAOP| \geq d$ or $|WOP_2| + |WAOP| \geq d$ in any hybrid eventual consistency system.*

Proof Suppose, in contradiction, that there exists such operations OP_1, OP_2 and AOP such that $|WOP_1| + |WAOP| < d$ and $|WOP_2| + |WAOP| < d$.

Because AOP is doubly non-interleavable with respect to OP_1 and OP_2, there is a sequence of operation ρ, instances of AOP, aop^1 and aop^2, and an instance of both OP_1 and OP_2, op_1 and op_2, such that $\rho \circ op_1 \circ aop^1$ and $\rho \circ op_2 \circ aop^2$ are legal, but we cannot put both aop^1 and aop^2 after ρ to obtain a legal sequence of operations.

We construct two executions, α_1 and α_2 as shown on the figure. Then, we merge them into α. Because of the longer message delay, it must remain admissible in hybrid eventual consistency.

However, it is not possible because AOP is doubly non-interleavable with respect to OP_1 and OP_2. After considering all legal sequences, there is no way that we can create a τ_1 that is legal. This contradicts the assumption. ■

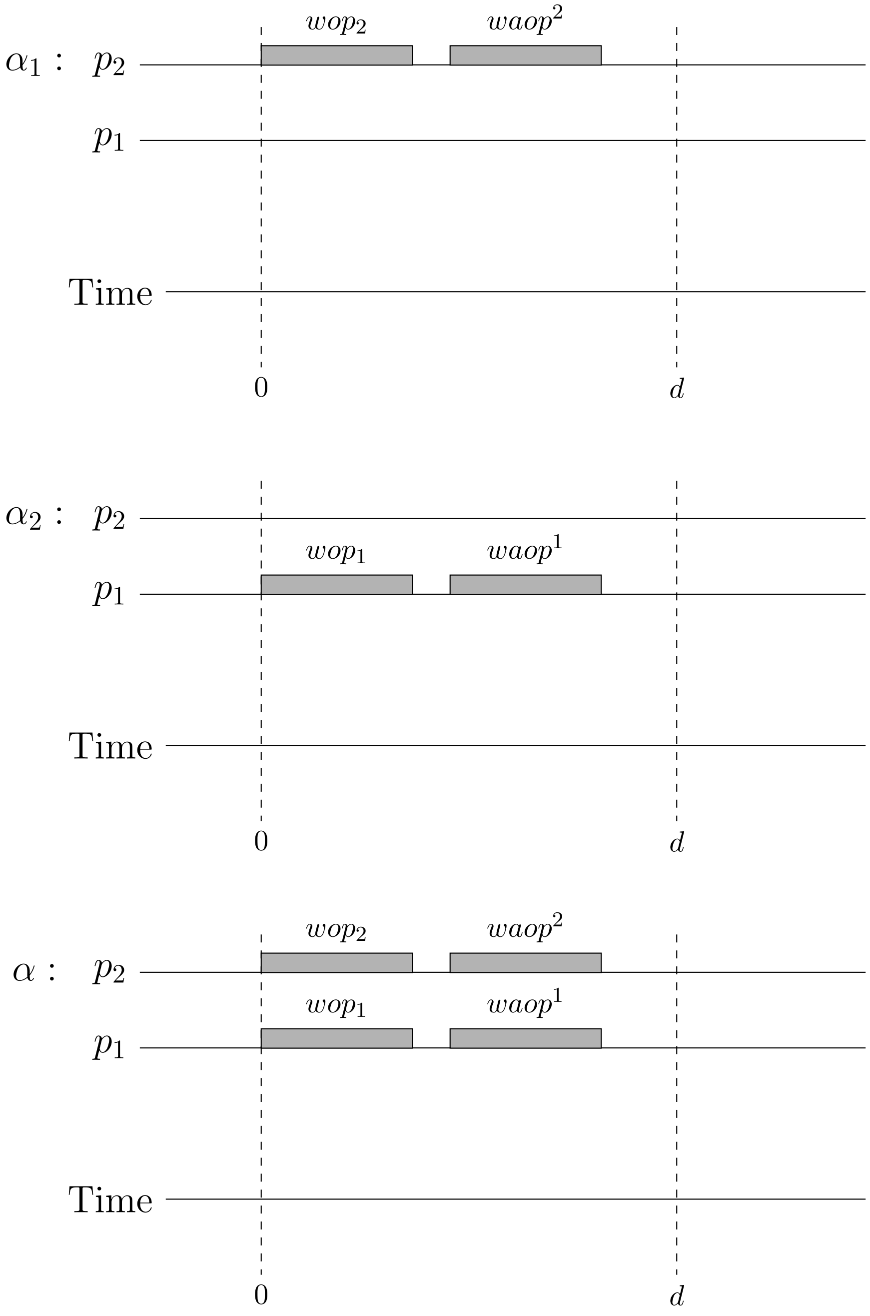

Fig. III.4. Counterexample in the proof of Theorem III.0.4

Comparison with hybrid consistency

In this chapter, we showed some lower bounds for the hybrid eventual consistency condition. These results are equal to hybrid consistency (Kosa [19]), which means that we are unable to improve the worst case running time in cases demonstrated above. However, for a relaxed data structure that does not fit into one of cases above, we might still be able to utilize the benefit of more flexibility to obtain performance gains.

CHAPTER IV

ALGORITHM

We give an algorithm that implements a shared register that satisfies hybrid eventual consistency condition. The system model follows the one described in Chapter II. There are n processes, $p_1, \ldots, p_n$. There is an inter-connection network with complete graph topology. We assume no message loss. Clocks are approximately synchronized. We give an algorithm that implements a shared register that satisfies hybrid eventual consistency condition. The system model follows the one described in Chapter II. There are n processes, $p_1..p_n$. There is an inter-connection network with complete graph topology. We assume no message loss. Clocks are approximately synchronized.

Following hybrid eventual consistency condition, we define four operations, WRead (weak read), WWrite (weak write), SRead (strong read) and SWrite (strong write).

Request	Response
WRead	Return(obj)
WWrite	Ack
SRead	SReturn(obj)
SWrite	SAck

Table IV.1

There is a function, *generate*, that generates the response for the corresponding read/write request. Table IV.1 shows the corresponding response fore each type of request.

In this algorithm, we use three message-passing primitives, *bcast* (Broadcast), *abcast* (Atomic broadcast) and *asend* (Atomic send). *abcast* broadcasts a message to all processes that satisfies the total order (Hadzilacos and Toueg [21]). A total order broadcast delivers all broadcast messages to all processes in the same order. *bcast* performs a broadcast without any guarantee on the message ordering. Both *abcast* and *asend* operations satisfy single-source FIFO order (Garcia-Molina and Spauster [22]). A single-source FIFO broadcast delivers messages sent from the same process in the order consistent with how it is sent. We give a formal condition below.

Properties of *abcast* and *asend*

Let S_b be the set of *abcast* operations, S_s be the set of *asend* operations. Let A_i be the set of *abcast* messages received by the process p_i. Let $B_{i,j}$ be the set of *abcast* and *asend* messages sent by the process p_i and received by p_j.

1. (Total Order Guarantee) For any two messages $m_1, m_2 \in A_i$ such that the process p_i receives m_1 before m_2, if $m_1, m_2 \in A_j$, p_j also receives m_1 before m_2 for any $j \in 1..n$.
2. (Single-source FIFO Guarantee) For any two message $m_1, m_2 \in B_{i,j}$ such that the process p_i sends m_1 before m_2, p_j receives m_1 before m_2.

Theoretically, we can avoid using *asend* by using a abcast call and disposing the messages delivered to all but the target process. However, this approach can be implemented more efficiently (Cristian et al. [23]).

Algorithm

The algorithm we propose is inspired by the algorithm for hybrid consistency (Attiya and Friedman [18]). It simulates a shared register that provides a strong version and a weak version for both read and write operations. In both algorithms, the weak read and strong write are the same and the other two differ. In our weak write routine, *abcast* is replaced by normal broadcast. In order to ensure all strong reads observe the same set of weak writes, we add a write back into the strong read routine. In addition, because we no longer use *abcast* for weak writes, the original global clock in the algorithm for hybrid consistency does not work for ours. Hence, we replace it with a vector clock.

The weak read operation performs a local read. The weak write operation broadcasts the new value and then returns immediately. Strong operations are more complicated. They need to make use of the ordering properties of *abcast* to ensure that the execution is admissible (valid) under hybrid eventual consistency.

In our algorithm, each process stores a copy of the memory *obj*, an integer *wait_acks* denoting how many messages have not been acknowledged, the result of a strong read *result*, a boolean *last_wr* denoting whether the last operation is a weak read, an enumerate variable *pending* denoting what is the pending strong operation, a vector clock *latest_update* and the process ID the last weak write came from (*latest_id*).

The weak read is done by reading the local memory and updating *last_wr*. The weak write performs a broadcast and then updates *latest_id*, *latest_update*, *wait_acks* and *last_wr* accordingly.

To execute a strong read, we firstly send a total order broadcast and then wait until the broadcast message arrives at the process itself. Afterwards, we send a broadcast message with the current memory object and stores it locally as the return value of this strong read. The response will not be returned until all *ack* messages come back.

The strong write is the same as the write algorithm for linearizability. We perform a total order broadcast and wait for one message to come back to the process itself before returning the response.

Algorithm 1 An implementation of shared register satisfying hybrid eventual consistency

```
function INITIALIZE$_i$                                ▷ Initialize the parameters for each process $p_i$
    $obj_i \leftarrow \perp$                           ▷ The simulated register variable to $\perp$ (Initial value)
    $wait\_acks_i \leftarrow 0$                        ▷ An integer denoting how many acks are missing.
    $result_i \leftarrow \perp$                        ▷ Type of register variable
    $last\_wr_i \leftarrow false$                      ▷ A Boolean
    $pending_i \leftarrow None$                        ▷ An enumerate type: None/SR/SW
    $latest\_update_i \leftarrow [0, 0, .., 0]$        ▷ An array of size $n$
    $latest\_id_i \leftarrow 0$                        ▷ An integer (Process ID)
end function

function WREAD$_i$                                     ▷ Invoke weak read on $p_i$
    $last\_wr_i \leftarrow true$
    generate(Return($obj$))
end function

function WWRITE$_i$($value$)                           ▷ Invoke weak write on $p_i$
    bcast(update, $value$, $i$, $latest\_update_i[i]$)
    $latest\_id_i \leftarrow i$
    $latest\_update_i[i] \leftarrow latest\_update_i[i] + 1$
    $wait\_acks_i \leftarrow wait\_acks_i + n$
    $last\_wr_i \leftarrow false$
    generate(Ack)
end function

function SREAD$_i$                                     ▷ Invoke strong read on $p_i$
    if $last\_wr_i$ then
        abcast(strong-read-wait)
        $wait\_acks_i \leftarrow wait\_acks_i + n$
    end if
    while $wait\_acks_i > 0$ do
        wait                                           ▷ Non-atomic
    end while
    abcast(strong-read)
    $wait\_acks_i \leftarrow wait\_acks_i + n$
    $pending_i \leftarrow SR$
end function

function SWRITE$_i$($value$)                           ▷ Invoke strong write on $p_i$
    while $wait\_acks_i > 0$ do
        wait                                           ▷ Non-atomic
    end while
    abcast(strong-write, $value$)
    $wait\_acks_i \leftarrow wait\_acks_i + n$
    $pending_i \leftarrow SW$
end function
```

```
function $\textsc{Received}_{i,j}$(update, $value$, $k$ , $clock$)    ▷ $p_i$ receives update message from $p_j$
    $\text{asend}_j$(ack)
    if $clock > latest\_update_i[k]$ then
        $obj_i \leftarrow value$
        $latest\_update_i[k] \leftarrow clock$
        $latest\_id_i \leftarrow k$
    end if
end function

function $\textsc{Received}_{i,j}$(strong-read-wait) ▷ $p_i$ receives strong-read-wait message from $p_j$
    $\text{asend}_j$(ack)
end function

function $\textsc{Received}_{i,j}$(strong-read)    ▷ $p_i$ receives strong-read message from $p_j$
    $\text{asend}_j$(ack)
    if $i = j$ then
        $result_i \leftarrow obj_i$
        bcast(update, $obj_i$, $latest\_id_i$, $latest\_update_i[latest\_id_i]$)
        $wait\_acks_i = wait\_acks_i + n$
    end if
end function

function $\textsc{Received}_{i,j}$(strong-write, $value$)    ▷ $p_i$ receives strong-write message from $p_j$
    $\text{asend}_j$(ack)
    $obj_i \leftarrow value$
end function

function $\textsc{Receive}_{i,j}$(ack)    ▷ $p_i$ receives strong-read message from $p_j$
    $wait\_acks_i \leftarrow wait\_acks_i - 1$
    if $wait\_acks_i = 0$ then
        if $pending_i = SW$ then
            generate(SAck))
        else if $pending_i = SR$ then
            generate(SReturn($result_i$))
        end if
        $pending_i \leftarrow None$
    end if
end function
```

Proof of Correctness

We prove the correctness of the algorithm by explicitly constructing the sequences τ_i as defined in hybrid eventual consistency condition.

We construct a sequence τ_i of operations and delivery events for every process p_i. Here, the delivery event happens whenever a message is delivered.

For each process p_i, let τ_i have all delivery events on p_i, ordered by the delivery time. Then, all weak read operations on p_i are inserted into τ_i according to the time of the invocation. Afterwards, all write operations executed by p_i are inserted into p_i immediately before the corresponding message delivery event (strong-write or update message). All weak writes that are not executed by p_i are then inserted before the last write preceding it. The execution of the weak write is determined by a logical clock (Lamport [24]) so that there must exist such previos weak write if an update message is ignored. We insert the strong reads immediately before its strong-read message delivery event. If the weak update message the strong read sr_j reads from has not arrived p_i, we drag the corresponding weak write prior to that strong read. Finally, all weak reads performed by other processes are ordered immediately after the write operation it reads from and delivery events are removed. There must exist such write operation.

Lemma IV.0.5 *τ_i is a legal sequence of operations for all p_i.*

Proof The legality of τ_i comes from the way we insert these operations. The reads are inserted immediately after the writes they read from. The ignored weak writes are inserted before another write so that it has no effect.

Formally, we assume, in contradiction, that there exists an illegal sequence τ_i. Therefore, there exists an operation $read_j(a)$ in p_j such that the latest write operation preceding $read_j(a)$ in τ_i is not the write operation it reads from.

1. $read_j(a)$ is a weak read

It is impossible if $i \neq j$, since $read_j(a)$ is inserted into τ_i, at the last step, directly after its corresponding write operation.

Therefore, we consider $i = j$. Denote $write^1(a)$ the write operation it reads from, $write^2(b)$ the latest write operation preceding $read_j(a)$. τ_i contains $write^1(a)...write^2(b)...read_i(a)$. $write^2(b)$ must not be executed on p_i. However, it contradicts with the algorithm we insert the unexecuted weak writes.

2. $read_j(a)$ is a strong read

   For the same reason, τ_i must contain $write^1(a)...write^2(b)...read_j(a)$. We discuss how $write^2(a)$ is inserted into τ_i. Since $read_j(a)$ reads from $write^1(a)$, this write operation must have been executed on p_j.

   It is impossible that the write message (update or strong-write) has not arrived before $read_j(a)$ is performed. If the write operation is strong, the strong-write message must have arrived because the total order of strong-write and strong-read messages. If the write operation is weak, the construction method will move the unarrived weak write prior to $read_j(a)$.

   The possibility remains is that $write^2(b)$ is not performed on p_j. Such unexecuted reads should be placed immediately before another write. This contradicts with the assumption that $write^2(b)$ is the last write operation prior to $read_j(a)$.

Consequently, τ_i is legal for allp_i. ■

Lemma IV.0.6 *τ_i is a permutation of the operations.*

Proof This is straightforward since we insert an operation into τ_i for any i exactly once. ■

Lemma IV.0.7 *If op_1 precedes op_2 in σ and op_1 and op_2 are both strong, then op_1 precedes op_2 in τ_i.*

Proof Both op_1 and op_2 need to use a total order broadcast. op_1 precedes op_2 implies that the broadcast message of op_1 must arrive at any process earlier than the one of op_2. Based on the way we construct τ_i, op_1 precedes op_2 on τ_i as well. ■

Lemma IV.0.8 $\tau|i = \rho|i$

Proof This can be seen from the construction of τ_i that τ_i has only operations from ρ. ■

Lemma IV.0.9 *The algorithm implements hybrid eventual consistency.*

Proof From the above lemmas, we can conclude that the algorithm satisfies hybrid eventual consistency. ■

CHAPTER V

SIMULATION

In this chapter, we analyze the performance of our algorithm for hybrid eventual consistency by a comparison with efficient algorithms that implement hybrid consistency, linearizability and sequential consistency respectively. Attiya and Friedman [18] proposed an algorithm that implements hybrid consistency, which is one of our comparison targets. Attiya and Welch [8] discussed algorithms for linearizability and sequential consistency. In our simulation, we use the traditional quorum algorithm for linearizability and local-read algorithm for sequential consistency.

System Model

We simulate the asynchronous distributed system by using discrete event simulation (Bolch et al. [25]). We maintain an event queue, which consists of the meta data of future events. In the discrete event simulation, we simply keep processing the earliest possible event in the event queue, and, if necessary, insert new ones into the event queue.

We also need to construct our own network model in order to perform this experiment. Since all these algorithms require an inter-connection network with complete graph topology, the only major problem left to us is the message delay. It is easy to assign a fixed delay or a randomly selected number from some distribution. However, these choices have inherent flaws. If we apply either of these models, we will find that, no matter how frequently we send messages, they will ultimately be delivered in a fixed expected delay. It can be inferred that we have infinite network bandwidth, which far deviates from actual computer systems. In fact, if the system is able to absorb infinite messages, the throughput would become infinite as well. Therefore, to make our experiment more realistic, we choose to make the message delay from process p_i to p_j a function of the link utilization of both p_i and p_j. This follows the normal circumstance that there is a network interface card (NIC), with limited bandwidth, in a computer and the whole bandwidth is shared by all traffic through the NIC.

We test the algorithms by keeping making read/write calls from each process. When a process is idle, we randomly choose an operation to invoke with a given distribution. To control the congestion, we stop calling when the current link utilization exceeds a given threshold. Then, the simulation program will follow the algorithm. We record the time cost in performing each strong operation.

Experiment has shown that the message delay has a roughly inverse relation with the link utilization (Kurose and Ross [26]). Inspired by this result, we apply the function

$$d(u) = min\{\frac{1}{1-u} - 0.98, 3\}$$

where d is the message delay in seconds and $0 \leq u < 1$ is the link utilization. Figure V.1 shows how the message delay increases when the link is becoming saturated.

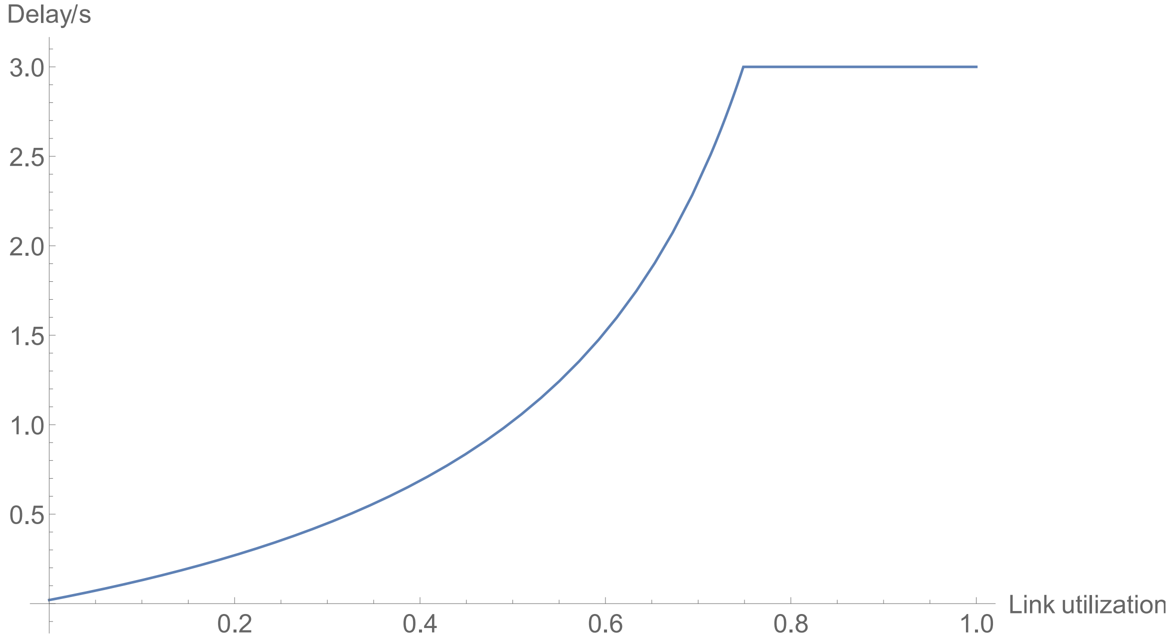

Fig. V.1. link utilization and transmission delay

Some of the algorithms depend on the total-order broadcast. It is implemented with a centralized algorithm. There is an additional process, having the same bandwidth restriction, being used to relay all the *abcast* and *asend* messages in a proper order.

Measurements

We conduct experiments with different n (number of processes), different algorithms and different mixture of strong and weak operations. In each simulation, we measure the throughput (the amount of operations performed in a second) and the average response time (the average response time for all strong operations performed in a given execution). Since weak operations are always returned immediately, we only need to measure the response time for all strong operations. Below we give the results extracted from all the experiments. The original data is also given in the Appendix A.

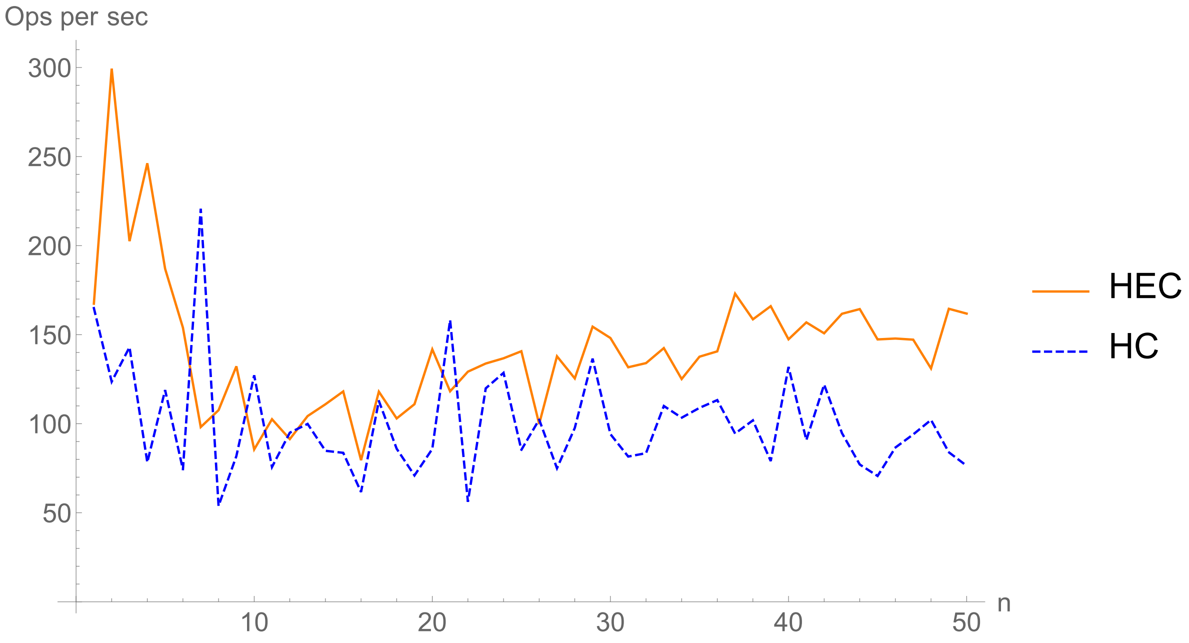

Fig. V.2. Throughput: Hybrid eventual consistency (HEC) and hybrid consistency (HC) (Strong/Weak=1)

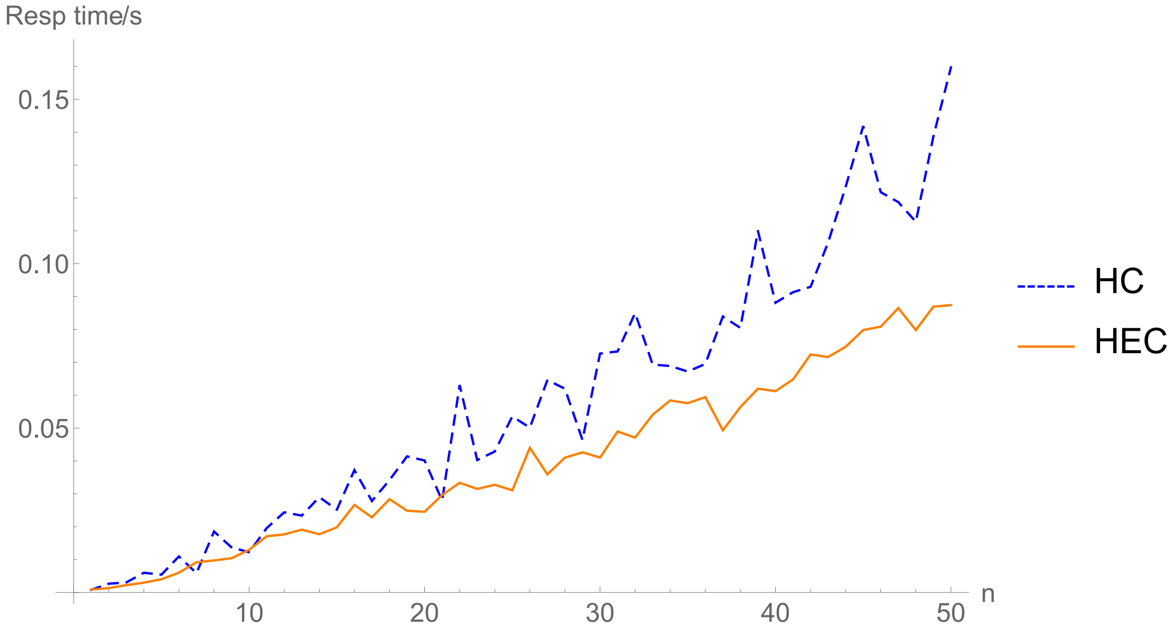

Fig. V.3. Average response time: Hybrid consistency (HC) and hybrid eventual consistency (HEC) (Strong/Weak=1)

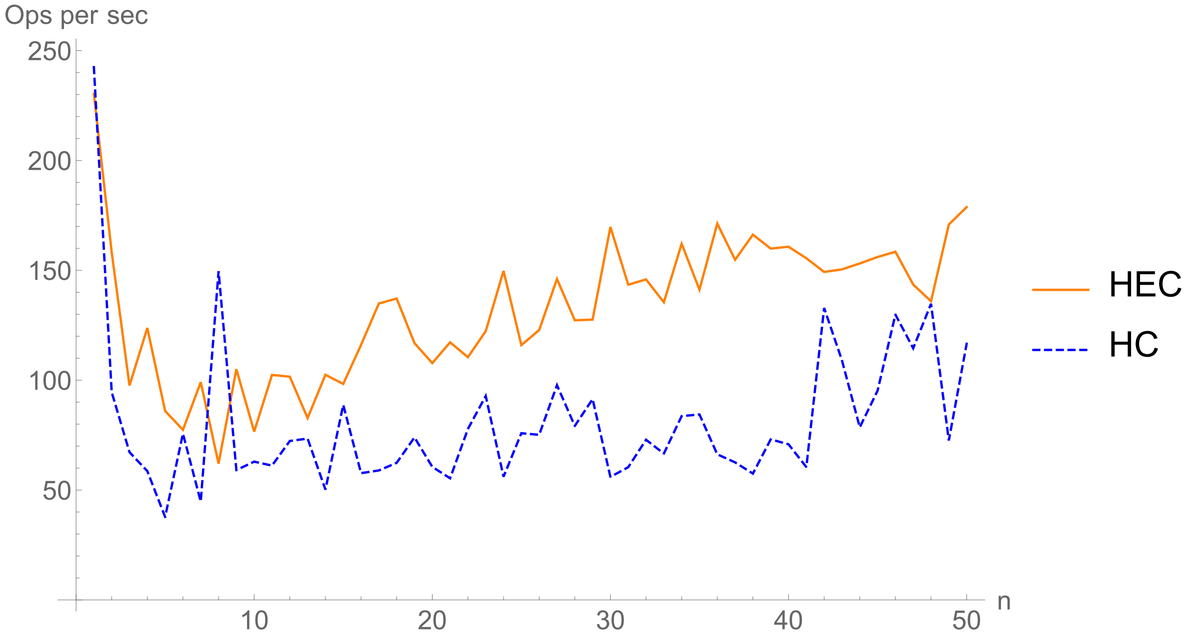

Fig. V.4. Throughput: Hybrid eventual consistency (HEC) and hybrid consistency (HC) (Strong/Weak=2)

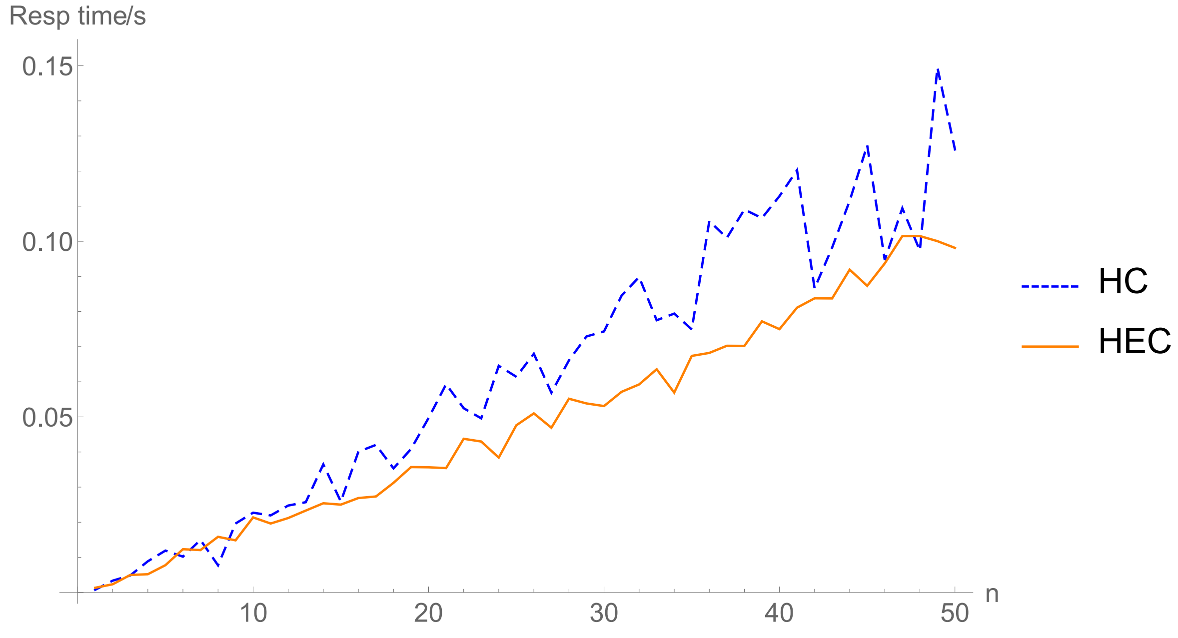

Fig. V.5. Average response time: Hybrid consistency (HC) and hybrid eventual consistency (HEC) (Strong/Weak=2)

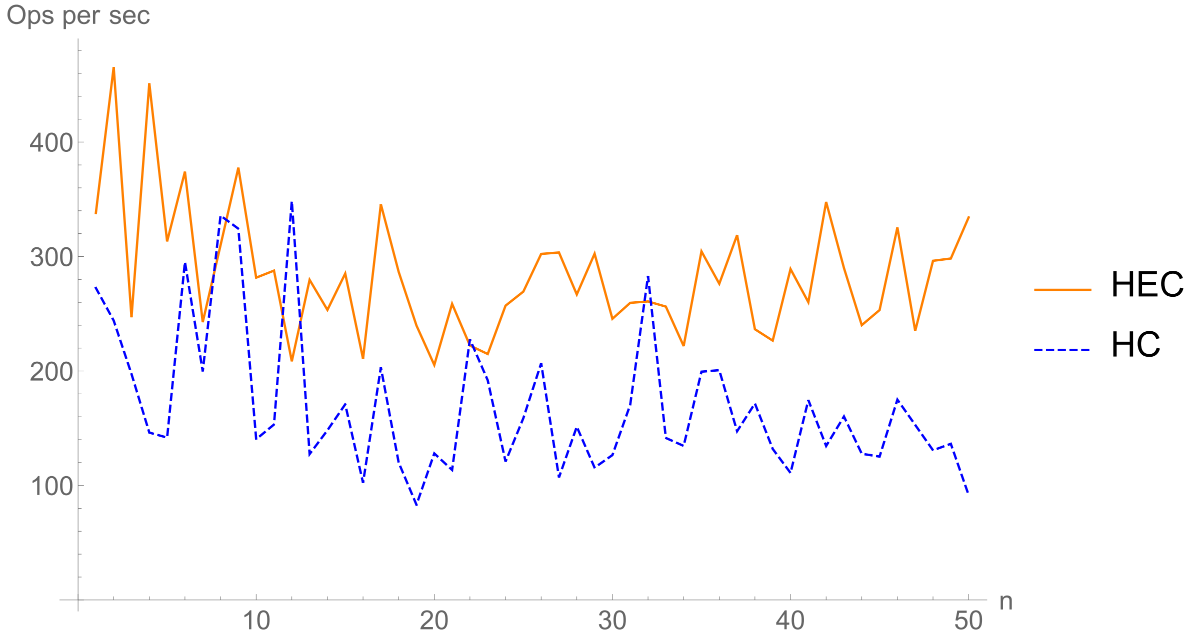

Fig. V.6. Throughput: Hybrid eventual consistency (HEC) and hybrid consistency (HC) (Strong/Weak=0.5)

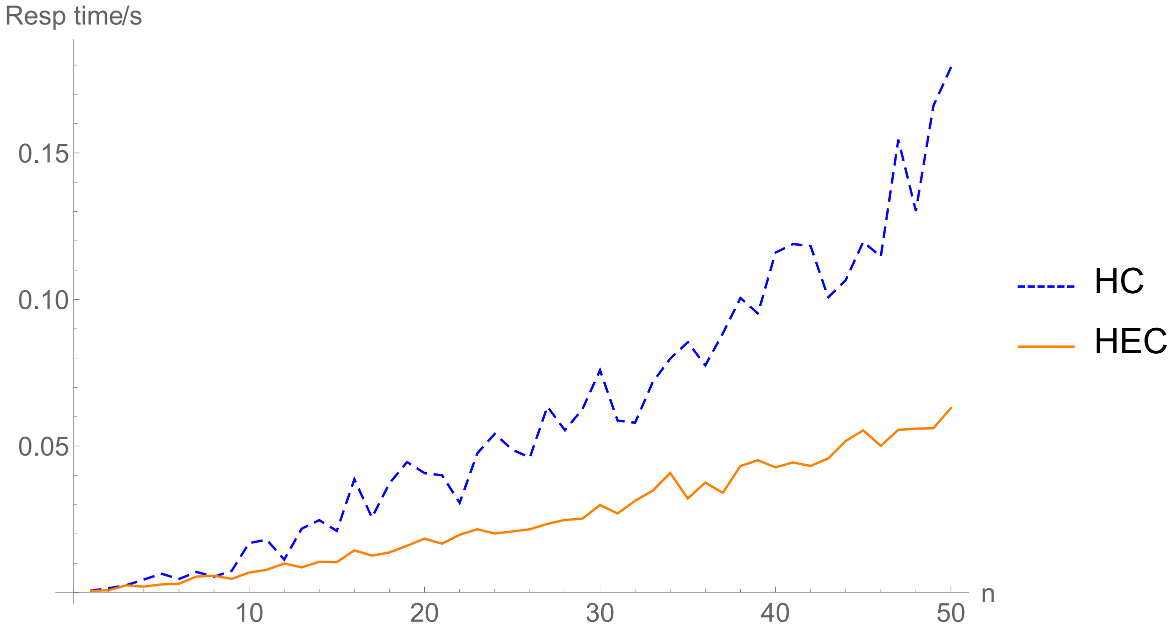

Fig. V.7. Average response time: Hybrid consistency (HC) and hybrid eventual consistency (HEC) (Strong/Weak=0.5)

In almost all the cases, our algorithm has a performance advantage over the implementation of hybrid consistency. This result is not too surprising as hybrid eventual consistency is weaker than hybrid consistency. The reason why our algorithm perform better is most likely because of, compared to the algorithm for hybrid consistency, the reduced use of total order broadcasts. The centralized total order broadcast is costly for the reason that all broadcast messages have to go through a rate-limited dedicated relaying process. We can see, from our experiment result, that this is the bottleneck of the system.

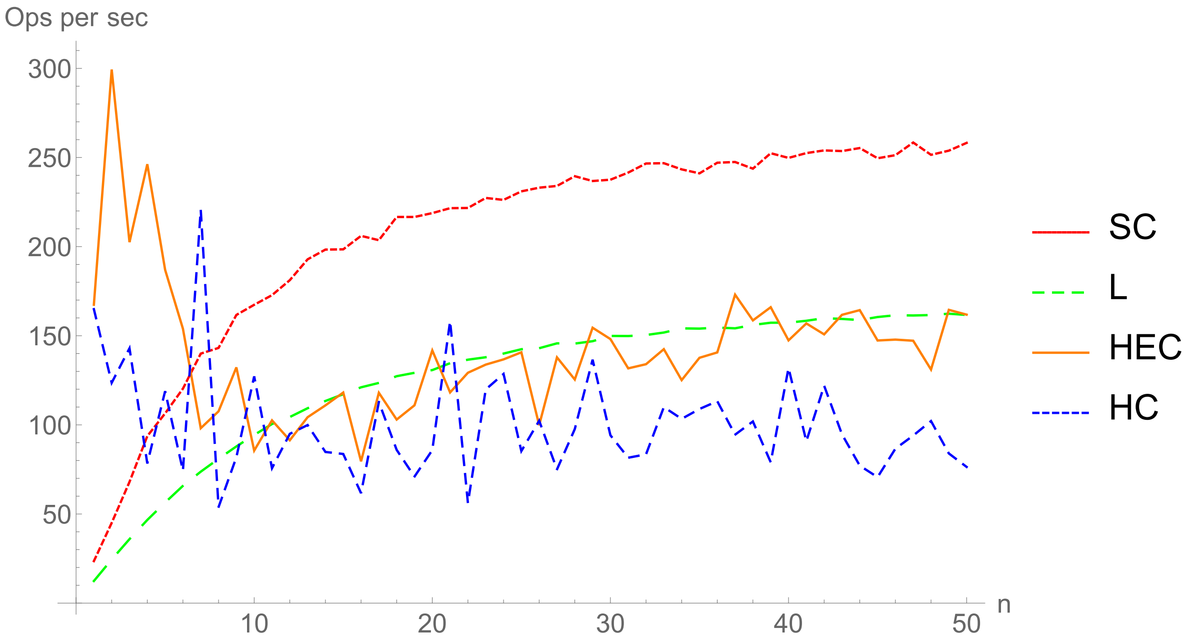

Fig. V.8. Throughput: Sequential consistency (SC), linearizability (L), hybrid eventual consistency (HEC) and hybrid consistency (HC) (Strong/Weak=1)

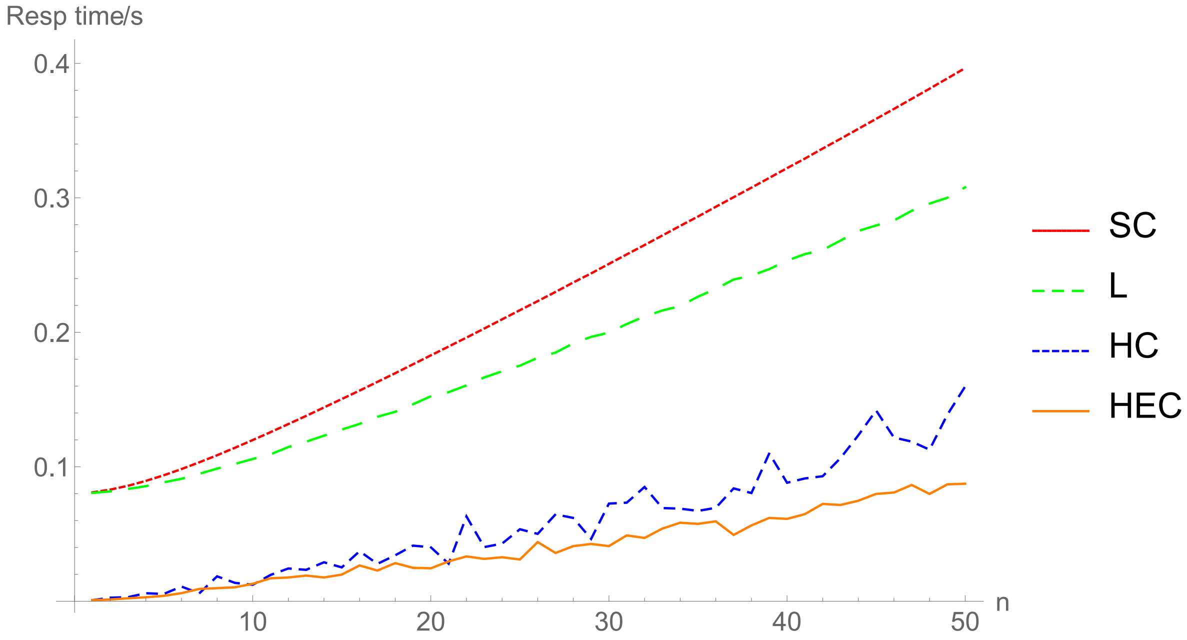

Fig. V.9. Average response time: Sequential consistency (SC), linearizability (L), hybrid consistency (HC) and hybrid eventual consistency (HEC) (Strong/Weak=1)

We observe more fluctuations from the data related to hybrid consistency as well as hybrid eventual consistency. This is likely because both algorithms offer zero-latency weak operations and long-latency strong operations. It is possible to have tens of weak operations or few strong operations done in few seconds, which is decided randomly. For linearizability and sequential consistency, there are only long-latency operations. No matter which operation is chosen, the overall amount of operations done is stable. Therefore, we can see smoother curves on Figure V.8 and Figure V.9.

Comparing our algorithm with the ones that implement linearizability and sequential consistency respectively, we see that the throughput of our algorithm is comparable to the one of linearizability and falls far below the one of sequential consistency. However, the average response time of our algorithm outperforms all other algorithms. From our observation, if we invoke more strong operations, the throughput will increase and so will the latency (latency is solely dependent on link utilization). Hence, we are able to invoke more operations and beat the throughput generated by linearizability, while having a similar average latency.

Overall, we believe that the performance of the proposed algorithm is competitive and useful.

CHAPTER VI

CONCLUSION

This thesis presents a theoretical study of hybrid eventual consistency. Motivated by hybrid consistency, we produce a working definition of hybrid eventual consistency in order to reach a better performance. Moreover, we prove lower bounds on the time complexity for operations under hybrid eventual consistency for certain types of data structures. We demonstrate an algorithm that implements hybrid eventual consistency efficiently and give a formal proof of its correctness. In this algorithm, weak operations return immediately. Compared to the algorithm for hybrid consistency, our algorithm uses significantly fewer total order broadcasts, which may be a bottleneck for the whole distributed system. From extensive experiments, we find that our algorithm provides better performance, in terms of response time and throughput, over hybrid consistency. However, it is not as fast as the algorithm implementing sequential consistency.

Our work leaves several questions. Is there a tight bound, in terms of communication complexity or time complexity, for hybrid eventual consistency condition? Are there important applications? Can we quantify the performance benefits of a more relaxed or a more restrictive version of our consistency condition? If so, how much? Also, it might be interesting to find a correct algorithm for consistency conditions while not compromising delay. Finally, because we are relying a lot more on distributed computing than before, it would be very interesting and extremely useful to have a better view of memory consistency condition so that we do not need to implement an excessively strong consistency condition for an application.

References

[1] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, *Munin: Distributed shared memory based on type-specific memory coherence.* ACM, 1990, vol. 25, no. 3.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *Computer*, vol. 29, no. 2, pp. 18–28, 1996.

[3] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, *The Midway distributed shared memory system.* IEEE, 1993.

[4] L. Lamport, "On interprocess communication," *Distributed computing*, vol. 1, no. 2, pp. 86–101, 1986.

[5] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[6] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 690–691, 1979.

[7] C. Scheurich and M. Dubois, "Correct memory operation of cache-based multiprocessors," in *Proceedings of the 14th annual international symposium on Computer architecture.* ACM, 1987, pp. 234–243.

[8] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 2, pp. 91–122, 1994.

[9] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber, *Comparative evaluation of latency reducing and tolerating techniques.* ACM, 1991, vol. 19, no. 3.

[10] R. J. Lipton and J. S. Sandberg, *PRAM: A scalable shared memory.* Princeton University, Department of Computer Science, 1988.

[11] J. R. Goodman, *Cache consistency and sequential consistency.* University of Wisconsin-Madison, Computer Sciences Department, 1991.

[12] M. Ahamad, J. E. Burns, P. W. Hutto, and G. Neiger, *Causal memory.* Springer, 1992.

[13] J. Bataller and J. Bernabeu, "Synchronized dsm models," in *Euro-Par'97 Parallel Processing.* Springer, 1997, pp. 468–475.

[14] C. Shao, J. L. Welch, E. Pierce, and H. Lee, "Multiwriter consistency conditions for shared memory registers," *SIAM Journal on Computing*, vol. 40, no. 1, pp. 28–62, 2011.

[15] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, *Managing update conflicts in Bayou, a weakly connected replicated storage system.* ACM, 1995, vol. 29, no. 5.

[16] S. Burckhardt, A. Gotsman, and H. Yang, "Understanding eventual consistency," Technical Report MSR-TR-2013-39, Microsoft Research, Tech. Rep., 2013.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[18] H. Attiya and R. Friedman, "A correctness condition for high-performance multiprocessors," in *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing.* ACM, 1992, pp. 679–690.

[19] M. J. Kosa, "Time bounds for strong and hybrid consistency for arbitrary abstract data types," *Chicago Journal of Theoretical Computer Science*, vol. 1999, p. 9, 1999.

[20] P. A. Bernstein and S. Das, "Rethinking eventual consistency," in *Proceedings of the 2013 international conference on Management of data.* ACM, 2013, pp. 923–928.

[21] V. Hadzilacos and S. Toueg, “Distributed systems (2nd ed.),” S. Mullender, Ed. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, ch. Fault-tolerant Broadcasts and Related Problems, pp. 97–145.

[22] H. Garcia-Molina and A. Spauster, “Ordered and reliable multicast communication,” *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 3, pp. 242–271, 1991.

[23] F. Cristian, H. Aghili, R. Strong, and D. Dolev, *Atomic broadcast: From simple message diffusion to Byzantine agreement.* Citeseer, 1986.

[24] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[25] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi, *Discrete Event Simulation.* John Wiley and Sons, Inc., 2006, pp. 607–656.

[26] J. F. Kurose and K. W. Ross, *Computer Networking: A top-down approach featuring the Internet.* Addison-Wesley Reading, 2011.

APPENDIX A

SIMULATION RESULTS

Columns

n: Number of processes

SW Ct: Number of strong writes executed

SR Ct: Number of strong reads executed

Write Ct/Count: Number of writes executed

Read Ct/Count: Number of reads executed

WW Ct: Number of weak writes executed

WR Ct: Number of weak reads executed

SW Delay: Average delay of strong writes

SR Delay: Average delay of strong reads

W/Write Delay: Average delay of writes

R Delay: Average delay of reads

SW Stdev Delay: Standard deviation of sample delays of all strong writes

SR Stdev Delay: Standard deviation of sample delays of all strong reads

W/Write Stdev Delay: Standard deviation of sample delays of all writes

R Stdev Delay: Standard deviation of sample delays of all reads

Hybrid eventual consistency (Strong/Weak=1)

n	SW Ct	SR Ct	WW Ct	WR Ct	SW Delay	SR Delay	SW Stdev Delay	SR Stdev Delay

1	1282	1239	1278	1223	0.00077	0.00102	0.00442	0.00383
2	2294	2243	2159	2285	0.00128	0.00138	0.00467	0.00476
3	1522	1543	1555	1455	0.00211	0.00234	0.0059	0.00639
4	1851	1811	1901	1824	0.00244	0.00352	0.00912	0.01485
5	1385	1402	1492	1331	0.00357	0.00444	0.00967	0.0124
6	1132	1212	1144	1130	0.00526	0.00674	0.01687	0.02828
7	743	715	767	719	0.00798	0.01043	0.01757	0.02403
8	778	774	800	876	0.00743	0.01207	0.01352	0.02288
9	958	991	984	1034	0.00927	0.0115	0.01983	0.02494
10	637	629	651	650	0.01117	0.0148	0.02237	0.02976
11	753	783	758	782	0.01438	0.01969	0.034	0.04715
12	684	709	687	661	0.01488	0.02035	0.03528	0.0419
13	798	770	802	759	0.01535	0.023	0.03583	0.06425
14	853	847	862	766	0.01382	0.02161	0.02957	0.05168
15	877	854	921	892	0.01639	0.02334	0.03888	0.05314
16	616	583	599	588	0.02535	0.02799	0.04966	0.05369
17	860	885	889	906	0.01949	0.0261	0.04584	0.05887
18	800	769	786	734	0.02576	0.03109	0.05756	0.07155
19	761	830	902	836	0.02022	0.02913	0.04643	0.06413
20	1036	1079	1074	1064	0.02069	0.02823	0.04951	0.06385
21	836	950	896	864	0.02508	0.03364	0.05744	0.07558
22	993	966	893	1025	0.02861	0.03824	0.06552	0.08669
23	992	1004	1006	1012	0.02629	0.03664	0.06	0.08411
24	999	990	1093	1020	0.0269	0.03867	0.0661	0.09333
25	1075	984	1080	1082	0.02928	0.03302	0.06846	0.08586
26	728	757	753	757	0.03527	0.05242	0.07276	0.11791
27	1007	1028	1046	1055	0.03008	0.04163	0.07268	0.09798

28	960	929	965	909	0.0375	0.04464	0.08875	0.09941
29	1144	1159	1159	1173	0.03782	0.04735	0.09069	0.10572
30	1074	1075	1111	1184	0.03524	0.04683	0.08241	0.10578
31	991	967	1015	978	0.04475	0.05329	0.09565	0.11581
32	965	992	1025	1039	0.04055	0.05346	0.08851	0.11399
33	1033	1088	1056	1097	0.04463	0.06301	0.10323	0.13681
34	942	898	947	966	0.04691	0.07046	0.10402	0.14171
35	1074	996	1025	1034	0.05237	0.06318	0.10466	0.1329
36	1035	1088	1028	1068	0.05	0.06839	0.11261	0.14009
37	1338	1255	1309	1289	0.03896	0.06038	0.09334	0.13654
38	1148	1209	1182	1219	0.04726	0.06509	0.11026	0.14338
39	1257	1192	1270	1261	0.05763	0.06656	0.12943	0.14454
40	1101	1078	1134	1109	0.05427	0.06838	0.12466	0.14961
41	1189	1166	1151	1201	0.05627	0.0735	0.13053	0.16474
42	1089	1106	1158	1170	0.06361	0.08104	0.1404	0.16436
43	1212	1182	1233	1225	0.065	0.07841	0.13723	0.16236
44	1195	1213	1259	1264	0.06245	0.08687	0.14385	0.19373
45	1116	1114	1083	1107	0.07001	0.08966	0.14128	0.17545
46	1085	1076	1110	1164	0.06996	0.09182	0.14543	0.17671
47	1096	1153	1105	1062	0.07646	0.09605	0.15542	0.18341
48	964	1006	999	962	0.07112	0.08814	0.15652	0.18342
49	1236	1257	1224	1219	0.07454	0.09914	0.15285	0.19416
50	1208	1228	1175	1244	0.07903	0.09565	0.16549	0.19742

Hybrid eventual consistency (Strong/Weak=2)

n	SW Ct	SR Ct	WW Ct	WR Ct	SW Delay	SR Delay	SW Stdev Delay	SR Stdev Delay
1	2300	2299	1130	1180	0.0014	0.00126	0.0086	0.0078

2	1630	1581	756	781	0.00188	0.00281	0.00819	0.01135
3	1014	925	520	468	0.00493	0.00504	0.0138	0.01672
4	1232	1236	606	639	0.00433	0.00605	0.01263	0.01695
5	873	871	431	405	0.00718	0.00835	0.01852	0.02188
6	830	764	355	373	0.01041	0.01439	0.0251	0.03207
7	959	999	502	513	0.01045	0.01362	0.02576	0.03641
8	632	616	331	282	0.01416	0.01758	0.02848	0.03516
9	1061	1020	553	516	0.01269	0.01708	0.03007	0.04271
10	802	760	359	377	0.01747	0.02549	0.0339	0.04694
11	1002	1042	523	505	0.01709	0.02207	0.03615	0.0508
12	1018	1035	482	513	0.01734	0.02499	0.03829	0.05446
13	814	828	446	395	0.02155	0.02505	0.04119	0.05029
14	1038	1020	520	496	0.02413	0.02669	0.05113	0.05606
15	978	985	515	471	0.02266	0.02735	0.04721	0.05608
16	1145	1149	595	593	0.02337	0.0304	0.05162	0.06297
17	1312	1373	681	681	0.02348	0.03101	0.05174	0.06272
18	1400	1347	713	655	0.02764	0.03493	0.06019	0.07343
19	1136	1167	594	609	0.03309	0.03825	0.06842	0.07782
20	1048	1105	525	555	0.03144	0.03963	0.06127	0.07551
21	1137	1209	567	604	0.0322	0.03845	0.06655	0.07811
22	1151	1074	523	566	0.03838	0.04953	0.0714	0.09087
23	1174	1238	631	626	0.03548	0.0501	0.07147	0.0996
24	1453	1539	761	739	0.03471	0.04189	0.07838	0.09118
25	1126	1187	581	585	0.03975	0.05506	0.07929	0.10358
26	1234	1229	622	600	0.04813	0.05388	0.09433	0.10394
27	1380	1498	763	740	0.04043	0.05289	0.08383	0.10892
28	1265	1309	630	614	0.0454	0.06461	0.08942	0.1159

29	1318	1244	651	614	0.04748	0.06056	0.09558	0.12268
30	1688	1704	862	838	0.04771	0.05841	0.10054	0.11874
31	1386	1442	740	736	0.04935	0.06462	0.10046	0.12712
32	1491	1421	741	723	0.04989	0.06906	0.10056	0.13277
33	1306	1358	689	711	0.05951	0.06747	0.10707	0.1205
34	1555	1663	820	825	0.05261	0.06095	0.11133	0.12476
35	1381	1426	712	717	0.05931	0.07517	0.11508	0.14233
36	1719	1661	879	880	0.05981	0.07692	0.12067	0.14833
37	1577	1530	728	809	0.0659	0.07473	0.12343	0.14247
38	1644	1662	827	854	0.06024	0.08009	0.12479	0.15779
39	1613	1626	785	773	0.06796	0.08639	0.13645	0.16445
40	1521	1611	876	814	0.06685	0.08267	0.12813	0.15747
41	1522	1583	753	807	0.07115	0.09068	0.14085	0.16595
42	1511	1464	732	771	0.07877	0.08893	0.15042	0.17465
43	1508	1558	727	720	0.07309	0.09404	0.14033	0.17233
44	1459	1638	735	762	0.07602	0.10611	0.14355	0.18724
45	1574	1574	815	719	0.07834	0.09632	0.14999	0.18466
46	1567	1514	851	822	0.08502	0.10272	0.165	0.19867
47	1463	1418	704	721	0.09124	0.11203	0.16299	0.19358
48	1337	1297	729	715	0.09267	0.11061	0.17145	0.19743
49	1712	1732	832	850	0.09565	0.10441	0.17067	0.1901
50	1766	1832	895	870	0.08584	0.10998	0.1719	0.20573

Hybrid eventual consistency (Strong/Weak=0.5)

n	SW Ct	SR Ct	WW Ct	WR Ct	SW Delay	SR Delay	SW Stdev Delay	SR Stdev Delay
1	1715	1762	3351	3314	0.00045	0.00059	0.00292	0.00364
2	2265	2250	4830	4618	0.0006	0.00096	0.00206	0.0036

3	1196	1307	2467	2437	0.0021	0.00277	0.00987	0.0114
4	2275	2384	4420	4462	0.00181	0.00217	0.00761	0.00897
5	1641	1644	3026	3085	0.00245	0.00308	0.00664	0.00842
6	1847	1769	3794	3814	0.00242	0.00359	0.00547	0.00986
7	1158	1239	2460	2423	0.00422	0.0066	0.0113	0.01968
8	1553	1527	3113	3104	0.00428	0.0072	0.01295	0.02877
9	1896	1882	3736	3815	0.00354	0.00576	0.01017	0.02461
10	1399	1402	2867	2778	0.00555	0.00806	0.01609	0.02439
11	1414	1473	2832	2912	0.00658	0.00887	0.02258	0.02814
12	1034	1089	2041	2090	0.00841	0.01127	0.02256	0.02953
13	1379	1353	2854	2803	0.00713	0.01006	0.01813	0.03008
14	1238	1247	2562	2554	0.00877	0.01219	0.02922	0.02894
15	1390	1413	2904	2847	0.00753	0.01312	0.02067	0.04006
16	1111	1018	2114	2078	0.01226	0.01673	0.04056	0.04897
17	1703	1750	3479	3438	0.00941	0.01567	0.03431	0.06119
18	1431	1443	2837	2888	0.01152	0.0158	0.04048	0.06438
19	1211	1158	2360	2462	0.01434	0.01766	0.03427	0.05629
20	1003	1057	2003	2095	0.0158	0.02079	0.04302	0.06099
21	1313	1319	2527	2601	0.01505	0.01827	0.05587	0.04897
22	1161	1104	2181	2222	0.01588	0.02371	0.05354	0.06808
23	1069	1094	2095	2187	0.01999	0.02315	0.06051	0.06193
24	1267	1296	2590	2559	0.01672	0.0235	0.06244	0.08408
25	1338	1295	2686	2762	0.01649	0.02534	0.04991	0.09609
26	1487	1472	3042	3067	0.01631	0.02691	0.05251	0.10429
27	1475	1461	3117	3053	0.02027	0.02663	0.07942	0.10094
28	1317	1361	2621	2707	0.02024	0.02914	0.07155	0.10665
29	1562	1497	3036	2979	0.01759	0.03315	0.0643	0.12262

30	1224	1217	2405	2522	0.02466	0.0351	0.08764	0.10969
31	1242	1322	2617	2605	0.02569	0.02817	0.08602	0.09341
32	1323	1314	2580	2606	0.02483	0.03785	0.08661	0.11451
33	1305	1239	2603	2541	0.03099	0.03869	0.10007	0.13705
34	1138	1126	2173	2217	0.02924	0.05241	0.08814	0.15945
35	1531	1484	2993	3127	0.02842	0.0359	0.10451	0.11801
36	1409	1321	2788	2769	0.02755	0.0481	0.09719	0.15728
37	1579	1590	3227	3165	0.02893	0.03909	0.11481	0.13363
38	1150	1199	2400	2346	0.03466	0.05133	0.12248	0.15213
39	1037	1204	2268	2287	0.0317	0.05669	0.08984	0.18927
40	1395	1442	2908	2927	0.03503	0.05014	0.14017	0.16837
41	1296	1288	2608	2611	0.03572	0.0531	0.13041	0.17216
42	1629	1710	3571	3520	0.03048	0.05531	0.12096	0.18707
43	1391	1410	2969	2929	0.03402	0.05729	0.12016	0.2064
44	1185	1212	2376	2429	0.04035	0.06282	0.12724	0.19045
45	1296	1222	2585	2494	0.04958	0.06144	0.1759	0.20643
46	1504	1619	3219	3420	0.03749	0.06169	0.14928	0.24108
47	1206	1163	2360	2322	0.04386	0.06763	0.13454	0.19131
48	1458	1488	3004	2938	0.04696	0.06468	0.16891	0.24459
49	1463	1511	2961	3013	0.03881	0.07277	0.14135	0.24075
50	1671	1619	3395	3340	0.05127	0.07506	0.19778	0.28122

Hybrid consistency (Strong/Weak=1)

n	SW Ct	SR Ct	WW Ct	WR Ct	SW Delay	SR Delay	SW Stdev Delay	SR Stdev Delay
1	1239	1277	1198	1233	0.00083	0.00077	0.00894	0.00814
2	929	911	916	952	0.00278	0.00254	0.01696	0.01874
3	1051	1100	1070	1068	0.00285	0.00322	0.01988	0.01864

4	572	639	551	588	0.00414	0.00765	0.02217	0.03389
5	902	880	924	861	0.00436	0.00648	0.02691	0.03504
6	568	523	566	572	0.01048	0.01147	0.04363	0.0438
7	1660	1561	1730	1669	0.00556	0.00618	0.03694	0.03951
8	382	439	425	370	0.01351	0.02293	0.04618	0.0604
9	579	632	598	646	0.01103	0.01612	0.04447	0.05672
10	945	989	930	952	0.01056	0.01385	0.04911	0.05675
11	594	587	557	533	0.01713	0.02203	0.06198	0.06828
12	740	708	703	698	0.02355	0.02524	0.06842	0.0754
13	745	741	747	764	0.02088	0.02588	0.07299	0.08035
14	657	624	634	629	0.02669	0.03142	0.08505	0.08794
15	658	632	613	607	0.02231	0.02818	0.07706	0.08502
16	477	457	461	455	0.02721	0.04766	0.08494	0.1089
17	848	883	794	859	0.02647	0.02911	0.08538	0.08815
18	643	654	631	651	0.02842	0.03983	0.08921	0.1081
19	539	567	515	508	0.03565	0.04683	0.09426	0.10361
20	625	609	640	701	0.0378	0.04256	0.1076	0.10911
21	1148	1253	1151	1195	0.01926	0.03594	0.08205	0.11463
22	440	400	420	425	0.05362	0.07352	0.1166	0.14155
23	908	884	914	890	0.03538	0.04531	0.1168	0.14542
24	964	879	965	1049	0.03878	0.04736	0.12528	0.14482
25	605	646	636	672	0.04546	0.06113	0.11559	0.14174
26	748	774	754	794	0.04145	0.0585	0.10849	0.13008
27	538	553	549	610	0.05621	0.07281	0.12733	0.14678
28	766	727	727	709	0.05731	0.06686	0.1501	0.16901
29	1027	1015	1056	999	0.04677	0.04597	0.14095	0.13901
30	725	669	736	695	0.06368	0.0824	0.16151	0.17988

31	627	582	620	618	0.05998	0.08768	0.15355	0.18505
32	606	637	627	634	0.07652	0.09303	0.17464	0.19342
33	810	845	806	838	0.06232	0.07601	0.15391	0.18916
34	762	791	771	777	0.0557	0.08159	0.14509	0.17972
35	797	789	816	865	0.06411	0.07033	0.1675	0.18037
36	842	777	900	879	0.04934	0.09134	0.14589	0.20383
37	689	710	710	726	0.06567	0.10166	0.16659	0.21729
38	766	761	742	788	0.06693	0.09411	0.17128	0.20376
39	562	606	610	593	0.09471	0.12412	0.19871	0.22232
40	994	976	1017	970	0.0717	0.10487	0.17474	0.22646
41	695	672	662	695	0.07355	0.10972	0.17823	0.22209
42	876	888	935	960	0.08286	0.10296	0.18787	0.21716
43	686	701	698	757	0.09386	0.11878	0.19466	0.23514
44	564	563	595	591	0.11112	0.1357	0.20968	0.23766
45	509	538	529	544	0.13396	0.14937	0.21834	0.23734
46	629	656	655	659	0.11399	0.12919	0.21303	0.23512
47	689	701	705	725	0.10708	0.13018	0.21098	0.23054
48	748	782	759	775	0.10018	0.12477	0.20228	0.24499
49	651	628	620	620	0.11576	0.16257	0.22021	0.26901
50	584	527	579	602	0.13898	0.1827	0.23344	0.28645

Hybrid consistency (Strong/Weak=2)

n	SW Ct	SR Ct	WW Ct	WR Ct	SW Delay	SR Delay	SW Stdev Delay	SR Stdev Delay
1	2417	2375	1216	1266	0.00073	0.00081	0.00776	0.00854
2	965	965	452	455	0.00335	0.00338	0.01903	0.01817
3	658	670	360	329	0.00463	0.00508	0.02196	0.02344
4	601	571	301	286	0.00727	0.01057	0.02838	0.03434

5	382	375	192	184	0.01182	0.01202	0.03469	0.03293
6	778	746	369	377	0.00967	0.01074	0.03509	0.03624
7	441	458	249	195	0.01498	0.01492	0.04408	0.04277
8	1533	1499	711	745	0.00717	0.00831	0.03352	0.03452
9	579	614	290	290	0.01603	0.02303	0.04805	0.05989
10	655	634	297	302	0.01945	0.02609	0.05115	0.0615
11	610	617	317	290	0.01985	0.02401	0.05433	0.06349
12	714	718	360	378	0.02115	0.02833	0.05941	0.06822
13	707	725	379	392	0.02377	0.02759	0.06513	0.06934
14	486	508	261	251	0.03407	0.03878	0.07578	0.08267
15	870	871	474	448	0.02003	0.03174	0.0645	0.08344
16	608	563	269	291	0.04041	0.03973	0.08287	0.08435
17	544	604	305	315	0.03933	0.04446	0.08562	0.09372
18	576	657	320	319	0.03556	0.03516	0.07893	0.07907
19	746	726	380	367	0.03958	0.04215	0.08578	0.08751
20	645	587	275	310	0.0452	0.05462	0.09012	0.0989
21	557	573	269	262	0.05783	0.06074	0.10415	0.10883
22	750	781	410	394	0.04223	0.06231	0.09825	0.12372
23	910	953	453	468	0.04687	0.05218	0.10752	0.11559
24	575	521	286	300	0.05799	0.07179	0.11943	0.13395
25	769	753	383	371	0.05642	0.06654	0.11254	0.12888
26	740	750	374	390	0.06361	0.07228	0.12191	0.13665
27	981	964	496	491	0.05071	0.06316	0.11326	0.12433
28	774	794	413	395	0.06131	0.07075	0.12277	0.13485
29	939	887	456	460	0.07052	0.07543	0.13359	0.14305
30	506	592	287	295	0.07138	0.07689	0.13009	0.1361
31	617	572	304	319	0.08125	0.08807	0.13817	0.14692

32	734	704	380	369	0.084	0.09591	0.14354	0.15727
33	663	689	336	310	0.0714	0.08342	0.13428	0.14709
34	853	796	419	442	0.07271	0.0866	0.13575	0.15567
35	851	861	410	410	0.0659	0.08384	0.13146	0.14919
36	663	658	330	337	0.09899	0.1126	0.15242	0.16572
37	602	613	328	335	0.08822	0.1135	0.14472	0.16825
38	539	577	308	301	0.09361	0.1235	0.14257	0.16716
39	714	714	367	397	0.1085	0.10473	0.16135	0.1695
40	687	718	353	367	0.11251	0.11323	0.15987	0.17292
41	656	586	282	290	0.11169	0.12985	0.15784	0.19161
42	1305	1376	646	659	0.08286	0.09013	0.18177	0.19654
43	1063	1148	553	507	0.09052	0.10534	0.18316	0.1983
44	801	748	382	426	0.10064	0.12321	0.16083	0.18464
45	949	959	489	460	0.12524	0.12943	0.21259	0.22239
46	1313	1275	651	657	0.09029	0.09918	0.18701	0.20367
47	1121	1169	601	545	0.0972	0.12117	0.1887	0.21604
48	1339	1294	663	747	0.09107	0.10388	0.18577	0.20198
49	728	751	361	337	0.15406	0.14515	0.22231	0.23056
50	1179	1162	574	582	0.1166	0.1359	0.20953	0.23017

Hybrid consistency (Strong/Weak=0.5)

n	SW Ct	SR Ct	WW Ct	WR Ct	SW Delay	SR Delay	SW Stdev Delay	SR Stdev Delay
1	1353	1349	2720	2753	0.00065	0.00066	0.0091	0.00772
2	1180	1212	2501	2429	0.00133	0.00158	0.01361	0.01664
3	980	997	1904	2041	0.00127	0.0037	0.01269	0.02995
4	735	761	1467	1423	0.00283	0.00593	0.02464	0.03865
5	712	659	1425	1459	0.00523	0.0077	0.0377	0.04687

6	1461	1497	2966	2934	0.00418	0.00499	0.03619	0.04033
7	995	1011	2034	1949	0.00626	0.00773	0.04443	0.05251
8	1676	1685	3306	3405	0.0042	0.00663	0.03765	0.05187
9	1597	1598	3161	3373	0.00649	0.00818	0.05118	0.05976
10	662	699	1429	1422	0.01401	0.01958	0.07637	0.09198
11	799	753	1586	1459	0.01444	0.02182	0.07873	0.09816
12	1762	1723	3504	3465	0.00979	0.01269	0.0659	0.0801
13	640	640	1306	1237	0.01876	0.02483	0.09113	0.10691
14	751	709	1473	1513	0.02261	0.0269	0.10056	0.11572
15	861	849	1706	1711	0.02239	0.01961	0.10415	0.08943
16	521	484	988	1076	0.0344	0.04326	0.12595	0.15176
17	1013	1045	2056	1981	0.02249	0.02853	0.10526	0.12455
18	572	601	1243	1185	0.03456	0.03984	0.12819	0.14088
19	410	465	789	830	0.04007	0.04846	0.12644	0.13945
20	627	594	1325	1294	0.03142	0.05058	0.12326	0.15509
21	575	554	1167	1111	0.03623	0.04392	0.12683	0.13785
22	1143	1106	2255	2326	0.02433	0.03703	0.11382	0.14773
23	938	943	1970	1905	0.04083	0.05404	0.17172	0.20391
24	600	615	1250	1161	0.04501	0.06307	0.15337	0.18365
25	760	783	1586	1635	0.03881	0.05836	0.14603	0.18642
26	985	1032	2046	2140	0.04009	0.05176	0.17092	0.19342
27	500	508	1093	1110	0.04649	0.08016	0.15795	0.20566
28	795	759	1523	1463	0.0435	0.0678	0.17148	0.21027
29	592	586	1132	1157	0.05973	0.06549	0.17229	0.18888
30	642	635	1246	1273	0.05581	0.09614	0.17308	0.23524
31	887	833	1688	1707	0.05931	0.05801	0.21054	0.2085
32	1360	1390	2927	2815	0.04588	0.06974	0.20057	0.25241

33	716	742	1424	1363	0.06055	0.08254	0.20165	0.21848
34	679	694	1356	1308	0.06857	0.09081	0.23671	0.25595
35	950	965	2044	2023	0.07991	0.09085	0.27562	0.29381
36	1003	971	2020	2028	0.06061	0.09496	0.21817	0.31525
37	707	755	1464	1495	0.07156	0.10423	0.20814	0.28902
38	802	859	1725	1763	0.10324	0.09794	0.3064	0.3068
39	629	622	1385	1322	0.08602	0.10451	0.25141	0.27384
40	573	535	1125	1095	0.10092	0.13213	0.27389	0.30568
41	840	850	1823	1726	0.09253	0.14497	0.31199	0.40899
42	666	631	1357	1381	0.08888	0.14927	0.25575	0.34378
43	783	773	1564	1688	0.08957	0.11201	0.26735	0.30528
44	631	595	1380	1225	0.09325	0.12073	0.26883	0.31232
45	590	569	1342	1256	0.11678	0.12266	0.30184	0.31864
46	895	884	1690	1780	0.09553	0.13395	0.29077	0.36244
47	760	732	1539	1556	0.12419	0.18603	0.34676	0.43386
48	668	640	1331	1283	0.11522	0.14582	0.28573	0.32889
49	660	696	1319	1417	0.1497	0.18161	0.36886	0.42358
50	403	475	930	941	0.14342	0.20953	0.31815	0.38475

Linearizability

n	Write Ct	Read Ct	W Delay	R Delay	W Stdev Delay	R Stdev Delay
1	195	177	0.08057	0.08056	0.00011	0.00011
2	361	373	0.08175	0.08168	0.00029	0.00032
3	557	520	0.08357	0.08346	0.00049	0.00053
4	684	716	0.08572	0.08547	0.00088	0.00101
5	872	821	0.08856	0.08834	0.00113	0.00121
6	953	1021	0.09119	0.09097	0.00146	0.00153

7	1121	1091	0.09488	0.09465	0.00201	0.00173
8	1265	1164	0.09878	0.09852	0.00233	0.00222
9	1333	1304	0.10225	0.10196	0.00275	0.00274
10	1406	1424	0.10612	0.10546	0.00303	0.00375
11	1438	1576	0.10951	0.10901	0.00354	0.00395
12	1594	1538	0.11478	0.11454	0.0041	0.00401
13	1635	1645	0.11876	0.11848	0.00476	0.00395
14	1705	1697	0.12349	0.1228	0.00536	0.00541
15	1764	1752	0.12808	0.12728	0.00567	0.00627
16	1796	1836	0.13231	0.13147	0.00605	0.00643
17	1878	1828	0.1375	0.13683	0.00651	0.00639
18	1877	1942	0.14127	0.14056	0.00701	0.00731
19	1951	1925	0.14692	0.14607	0.00819	0.00749
20	2030	1892	0.1526	0.15227	0.0085	0.00711
21	1991	2047	0.15589	0.15513	0.00895	0.00845
22	2033	2064	0.16108	0.15994	0.00926	0.01028
23	2100	2038	0.16678	0.16611	0.00983	0.00962
24	2109	2091	0.17149	0.17049	0.01063	0.01032
25	2114	2159	0.17596	0.17446	0.01106	0.01208
26	2162	2128	0.18141	0.18066	0.01174	0.011
27	2150	2222	0.18523	0.18463	0.01194	0.01128
28	2225	2143	0.19237	0.1911	0.01248	0.01356
29	2232	2176	0.19705	0.19618	0.01334	0.01295
30	2199	2298	0.20061	0.19888	0.0144	0.01508
31	2247	2248	0.20675	0.20558	0.01439	0.01399
32	2272	2240	0.21227	0.21123	0.01479	0.01438
33	2270	2284	0.21718	0.2153	0.01573	0.01655

34	2242	2382	0.22035	0.2187	0.01629	0.01578
35	2305	2315	0.22743	0.22561	0.01737	0.01671
36	2322	2315	0.23316	0.23143	0.01739	0.01765
37	2380	2245	0.24018	0.23829	0.01857	0.01991
38	2327	2355	0.24276	0.24152	0.01832	0.01753
39	2330	2389	0.24778	0.24624	0.01932	0.01825
40	2362	2358	0.25391	0.2527	0.01989	0.01956
41	2371	2381	0.25894	0.25735	0.02032	0.01993
42	2322	2474	0.26213	0.26039	0.01991	0.02033
43	2359	2424	0.26911	0.26737	0.02121	0.02106
44	2405	2359	0.27625	0.27449	0.02159	0.02277
45	2393	2422	0.28056	0.27842	0.02177	0.02476
46	2364	2481	0.28425	0.28214	0.02348	0.02368
47	2414	2427	0.2911	0.28972	0.02367	0.02218
48	2416	2432	0.29677	0.29469	0.0238	0.02555
49	2403	2468	0.30112	0.29902	0.02445	0.02721
50	2445	2405	0.30884	0.30683	0.02529	0.0266

Sequential consistency

n	Write Count	Read Count	Write Delay	Write Stdev Delay
1	370	337	0.08098	0.0001
2	722	628	0.08302	0.00031
3	1047	993	0.08588	0.00061
4	1339	1472	0.0895	0.00098
5	1600	1593	0.09366	0.00141
6	1829	1781	0.09829	0.00188
7	2030	2169	0.10334	0.0024

8	2208	2087	0.10857	0.00294
9	2359	2491	0.11414	0.00352
10	2499	2522	0.11989	0.00411
11	2618	2565	0.12575	0.00471
12	2724	2711	0.13178	0.00534
13	2821	2965	0.13785	0.00597
14	2908	3042	0.14412	0.00661
15	2985	2969	0.15041	0.00727
16	3056	3125	0.15666	0.00791
17	3115	2993	0.16314	0.00858
18	3172	3327	0.16959	0.00926
19	3228	3271	0.17624	0.00992
20	3272	3292	0.18286	0.01064
21	3318	3329	0.18944	0.0113
22	3354	3296	0.19606	0.01197
23	3387	3432	0.20282	0.01269
24	3424	3362	0.20964	0.01335
25	3450	3480	0.21646	0.01411
26	3484	3508	0.2232	0.01481
27	3510	3511	0.23005	0.01547
28	3528	3657	0.23708	0.01621
29	3553	3551	0.24382	0.01691
30	3570	3556	0.25089	0.01759
31	3596	3650	0.25795	0.0183
32	3614	3785	0.26491	0.01909
33	3630	3774	0.27201	0.01989
34	3638	3662	0.27915	0.02062

35	3647	3586	0.28611	0.02135
36	3672	3740	0.29335	0.02194
37	3669	3755	0.30048	0.02268
38	3686	3626	0.30758	0.02338
39	3705	3867	0.31483	0.02402
40	3710	3783	0.32209	0.02475
41	3720	3854	0.3292	0.02557
42	3724	3895	0.33667	0.02645
43	3739	3869	0.34389	0.02733
44	3740	3919	0.3513	0.0281
45	3735	3752	0.35874	0.02893
46	3743	3796	0.36612	0.02968
47	3760	3992	0.37365	0.03032
48	3744	3801	0.38115	0.03091
49	3772	3843	0.38882	0.03147
50	3750	3996	0.39652	0.03331