CONSISTENCY CONDITIONS
FOR DISTRIBUTED
SHARED MEMORIES

Roy Friedman
CONSISTENCY CONDITIONS
FOR DISTRIBUTED
SHARED MEMORIES

RESEARCH THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Science

by
Roy Friedman

Submitted to the Senate of the Technion - Israel Institute of Technology
Tamuz 5754 Haifa June 1994
This research was carried out in the faculty of Computer Science under the supervision of Dr. Hagit Attiya.

I wish to thank Dr. Hagit Attiya for her invaluable help and her professional guidance throughout this research. Soma Chaudhuri and Jennifer Welch have collaborated with us on part of this work. In particular, Jennifer Welch has provided many useful comments on previous versions of this work. Thanks also to Shmuel Katz and Bilha Mendelson.
# Contents

Abstract ................................. 1
List of symbols and abbreviations ................. 3

1 Introduction .......................... 4

2 Related Work .................... 12
   2.1 Overview .............................. 12
   2.2 Comparison with Our Work ........... 14
      2.2.1 The Interface and Definitions ..... 14
      2.2.2 Support for Non-Sequential Execution of Operations ..... 15
      2.2.3 Programming Techniques for Hybrid Conditions .. 16
      2.2.4 Describing Real Architectures .......... 17

3 The Basic Framework ................. 18
   3.1 The System ............................ 18
   3.2 Defining Consistency Conditions ........ 23
      3.2.1 Examples .......................... 23

4 Weak Conditions ................... 26
   4.1 Weak Consistency .................... 26
      4.1.1 Examples .......................... 27
   4.2 Mutual Exclusion Using Weak Consistency .. 28
   4.3 Other Weak Conditions ............... 33
## Contents (Cont.)

4.3.1 Causal Memory ........................................... 33 
4.3.2 Processor Consistency ................................. 34 

5 Hybrid Consistency ........................................ 40 
  5.1 Definition of Hybrid Consistency ........................ 40 
  5.1.1 Examples ............................................ 42 
  5.2 Programming Example .................................... 43 
  5.3 Compositionality of Hybrid Consistency ............... 47 

6 Implementing Hybrid Consistency ............................ 49 
  6.1 A Generic Implementation for Hybrid Consistency .... 50 
  6.1.1 The Algorithm ...................................... 51 
  6.1.2 Proof of Correctness ................................ 53 
  6.1.3 Accessing Strong Objects with Weak Reads ....... 60 
  6.1.4 Complexity Analysis ................................ 62 
  6.2 Providing Hybrid Consistency with Read/Write Objects .. 62 
  6.2.1 The Algorithm ...................................... 63 
  6.2.2 Proof of Correctness ................................ 65 
  6.2.3 Complexity Analysis ................................ 81 
  6.3 Atomic Broadcast ....................................... 82 

7 Lower Bounds for Implementing Hybrid Consistency ........... 84 
  7.1 Reads and Writes ....................................... 84 
  7.2 Read-Modify-Write .................................... 86 
  7.3 Dequeue and Pop ....................................... 87 

8 A Framework for Non-Sequential Executions .................... 90
## Contents (Cont.)

8.1 Overview ................................................................. 90
8.2 System Components .................................................. 92
8.3 Sequential Executions ............................................... 93
8.4 Definitions of Non-Sequential Consistency Conditions ........ 95
  8.4.1 Discussion of the Framework ............................... 97

9 Programming Methods for Hybrid Consistency .................... 99
  9.1 Static Approaches .................................................. 99
    9.1.1 Strong Writes ............................................... 100
    9.1.2 Strong Reads ............................................... 100
  9.2 Programming with Critical Sections ............................ 103
  9.3 Running Data-Race-Free Programs ............................... 110
    9.3.1 Definition of Data-Race-Free Programs .................. 110
    9.3.2 The Influence Relation .................................... 110
    9.3.3 Running Data-Race-Free Programs on Hybrid Consistent Memory .......................... 112
    9.3.4 Reordering Operations of the Same Process .............. 116
  9.4 Programmability vs. Hardware Optimizations .................. 117

10 Alpha Consistency ..................................................... 119
  10.1 The Alpha Shared Memory Model ............................... 119
    10.1.1 Informal Description .................................... 119
    10.1.2 Defining Alpha Consistency ............................. 121
    10.1.3 Examples .................................................. 122
    10.1.4 Relation to the Specification of Alpha ................ 123
    10.1.5 Control Order ............................................. 125
  10.2 Programming with Critical Sections ........................... 126
    10.2.1 The General Method ..................................... 126
Contents (Cont.)

10.2.2 Lamport’s Bakery Algorithm .......................... 128
10.2.3 Peterson’s Algorithm ............................... 131
10.3 Running Generalized Data-Race Free Programs ............ 131
   10.3.1 Generalized Data-Race Free Programs ............... 132
   10.3.2 MBV and MBP Sequences ............................ 133
   10.3.3 Running GDRF Programs on Alpha Consistent Memories .. 134

11 Discussion ................................................................ 139
   11.1 Review ...................................................... 139
   11.2 Further Research ........................................... 140
      11.2.1 Understanding Consistency Conditions ............ 140
      11.2.2 Implementations ........................................ 141
      11.2.3 Programming Techniques ............................. 143
      11.2.4 Extending the Framework ............................ 144

References .................................................................... 145
List of Figures

1.1 Example of out-of-order execution of operations .................. 7
3.1 System architecture ................................................. 19
3.2 Examples of executions ............................................. 24
4.1 Execution $\sigma_{\varepsilon}$, not weakly consistent .................. 28
4.2 Executions used in the proof of Theorem 4.1 ......................... 32
4.3 Execution $\sigma$ used in the proof of Theorem 4.3 .................. 39
5.1 Mutual exclusion using hybrid consistency – code for process $p_i$, $i = 0, 1, \ldots$ 44
6.1 The generic algorithm — code for process $p_i$ ....................... 54
6.2 Proving the legality of $\tau_j$ in the generic algorithm ............... 58
6.3 The algorithm — code for process $p_i$ ................................ 66
6.4 The algorithm — code for handling message receive events at process $p_i$ ................. 67
6.5 Partial/potential execution intervals ................................. 68
6.6 Atomic broadcast — code for process $p_i$ ............................ 83
8.1 Non-sequential system architecture ................................... 91
8.2 Demonstrating the importance of having control operations in the framework ........ 96
8.3 A possible execution $\sigma$ without control operations ............... 97
8.4 Limitations of the framework ....................................... 98
9.1 A possible execution $\sigma$ without the unique writes assumption ........ 102
List of Figures (Cont.)

9.2 A possible execution $\sigma$ if not every write is read ............... 103
9.3 A cooperative solution for the mutual exclusion problem – code for $p_i$ .... 108
9.4 Reordering the operations of the same process to the same object ............ 117
9.5 A possible execution $\sigma$ if operations of the same process to the same object can be reordered ........................................ 117
9.6 The ordering of weak reads .................................................. 118
10.1 Example execution $\sigma'_{3}$ .................................................. 123
10.2 Litmus test 3 for alpha consistency ....................................... 124
10.3 Litmus test 4 for alpha consistency ....................................... 124
10.4 Litmus test 7 for alpha consistency ....................................... 125
10.5 Lamport’s bakery algorithm using alpha consistency – code for processor $p_i$, $i = 1, \ldots, n$. .................................................. 128
10.6 Peterson’s algorithm using alpha consistency – code for processor $p_i$, $i = 0, 1$. . 132
Abstract

This thesis addresses theoretical aspects of defining, implementing and programming with consistency conditions for distributed shared memories. We start by presenting a framework for defining consistency conditions from the programmer's point of view. We then show that weak conditions are too weak to solve some of the fundamental problems of concurrent programming. This provides a motivation for hybrid conditions.

Following this, we present a formal definition of hybrid consistency, capturing the main features of several hybrid conditions. In these conditions, operations are classified as either weak or strong. Weak operations provide a semantics similar to weak conditions and can be implemented efficiently. Strong operations, on the other hand, provide a semantics similar to strong conditions which is necessary to maintain the expressiveness of real shared memories in distributed shared memories.

Next, we develop two algorithms for implementing hybrid consistency in a message based system. In these algorithms, weak operations are executed instantaneously while the time required to execute strong operations is $O(d)$, where $d$ is the network delay. The first algorithm is a generic implementation and assumes that weak and strong operations do not access the same objects. However, when restricted to read, write and read-modify-write operations, or to FIFO queues and stacks, it is still correct even if weak reads (but not weak writes) are allowed to access the same objects as strong operations. The second algorithm supports only reads and writes, but allow every object to be accessed by every operation (weak or strong).

We also show lower bounds on the time required to execute operations by implementations of hybrid consistency. Specifically, for read/write objects, we show that if weak operations are executed faster than the network delay, then the execution time for strong operations must be at least $\Omega(d)$. We also show that read-modify-write, dequeue and pop operations cannot be executed faster than $\Omega(d)$, even if they are labeled as weak. These results indicate that the algorithms we developed for implementing hybrid consistency are optimal, up to a constant multiplicative factor.

Next, we generalize the basic framework for defining consistency condition to incorporate non-sequential executions, i.e., executions that allow to invoke operations in parallel and in
pipeline and to execute them out-of-order and even speculatively. As before, in this generalized framework, consistency conditions are defined from the programmer’s point of view. We use the generalized framework to generalize the definitions of sequential consistency, weak consistency and hybrid consistency to allow non-sequential executions.

We use the generalized definition of hybrid consistency to develop several programming techniques for it. The first technique is to run only data-race free programs. We show that these programs run on hybrid consistent hardware as if it was sequentially consistent. The second technique is to statically label operations as weak or strong, based on their type (read or write) and the objects they access. The third technique is to protect shared data within critical sections. For this, we show how to transform any given non-cooperative solution to the mutual exclusion problem based on sequential consistency into a solution based on hybrid consistency.

Finally, we define *alpha consistency*, a formal definition of the shared memory model of DEC Alpha based multiprocessors, in our generalized framework. We use this definition to develop several programming techniques for alpha consistency, similar to the ones developed for hybrid consistency. We show that every generalized data-race free program runs on alpha consistent hardware as if it was sequentially consistent. We also present a general method that transforms any given non-cooperative solution to the mutual exclusion problem based on sequential consistency into a solution based on alpha consistency. Finally, we show that this method is not optimal; in some cases, a specific analysis of the code yields a more efficient solution.
List of symbols and abbreviations

\(d\) the network delay.

\(op_i\) an operation by process \(p_i\).

\(sop_i\) a strong operation by process \(p_i\).

\(r_i(x, v)\) a read by process \(p_i\) that returns the value \(v\) from object \(x\).

\(sr_i(x, v)\) a strong read by process \(p_i\) that returns the value \(v\) from object \(x\).

\(w_i(x, v)\) a write by process \(p_i\) that writes the value \(v\) to object \(x\).

\(sw_i(x, v)\) a strong write by process \(p_i\) that writes the value \(v\) to object \(x\).

\(rmw_i(x, f, v)\) a read-modify-write operation by process \(p_i\) that uses the function \(f\) to access object \(x\) and returns the value \(v\).

\(\tau\) a sequence of operations.

\(\tau_i\) the view of process \(p_i\).

\(f\leq s\) flow control sequence for process \(p_i\).

\(c\Rightarrow\) control order for process \(p_i\).

MCS memory consistency system.

DRF data-race free.

GDRF generalized data-race free.
Chapter 1

Introduction

Highly parallel multiprocessors offer a potential for high-speed computing at a relatively low cost. This potential and the growing demand for computing power have increased the interest in multiprocessors. In order to fully utilize multiprocessors, simple and efficient paradigms for communication between processes must be developed. Shared memory is a convenient paradigm, since it is more high level than message passing and it is a natural extension of serial programming. This has led to a wide study of concurrent programming with shared memory, and many problems have already been solved using shared memory. Thus, supporting shared memory on distributed machines is a desired goal.

Unfortunately, implementing shared memory on a multiprocessor is much more complicated than it is on a uniprocessor. This is due to the higher degree of parallelism and the lack of synchronization that is inherent in the distributed architecture. Thus, the exact semantics of the shared memory must be explicitly defined. A consistency condition is the definition of the memory’s semantics. It describes the guarantees that hardware should present to software about the way memory operations may be seen at different processes.

Consistency conditions can be roughly classified into three categories: strong conditions, weak conditions and hybrid conditions. Strong conditions, e.g., sequential consistency [43] and linearizability [38], require the distributed shared memory to behave like real shared memory. Programming a distributed shared memory machine that support a strong consistency condition is as simple as programming a real shared memory machine. However, strong conditions cannot be implemented efficiently [19, 44].
Weak conditions, e.g., Pipelined RAM [44] and causal memory [10], provide very little guarantee about the order in which operations may be executed and therefore allow aggressive hardware implementations. However, as we show in this thesis, the semantics provided by these conditions is too weak to solve some of the fundamental problems in concurrent programming.

Hybrid conditions, e.g., weak ordering [31], release consistency [33], DRF0 [3], DRF1 [5] and hybrid consistency [17], classify the memory operations into strong operations and weak operations. Hybrid conditions seem to offer a good compromise between strong conditions and weak conditions: The weak operations can be implemented efficiently and therefore programs that mainly use weak operations result in high performance; strong operations can be used whenever the semantics of strong conditions is required. Hence, hybrid conditions are as expressive as strong conditions. Indeed, the shared memory model of many of the recently developed architectures support some form of hybrid conditions (e.g., DEC-Alpha [1, 57] and IBM/Apple/Motorola PowerPC [26]).

The main problem with most definitions of hybrid conditions is that they are given from the architect's point of view. That is, they describe the way operations are supposed to be executed rather than the order in which operations should be viewed. This makes these conditions uncomfortable for programmers and for theoreticians wanting to investigate them.

We have tried to overcome this problem by presenting a formal definition of hybrid consistency. This definition was developed from the programmer's point of view. That is, after realizing that strong conditions are impractical for large systems, we have tried to point out what are the minimal guarantees about the ordering of operations that a programmer would not want to give up. Then, we have tried to formalize these guarantees in a programmer oriented manner, i.e., by describing the possible ordering of operations. This is different from the usual approach for defining consistency conditions, i.e., finding out what can be implemented efficiently and then describing it in terms of what hardware optimizations are allowed or how operations should be executed.

The next step in this research is to investigate whether it is possible to implement hybrid consistency in such a way that (at least) the weak operations are executed faster than the network delay. Recall that the basic motivation for using hybrid conditions, instead of strong conditions, is the realization that strong conditions cannot be implemented efficiently. In this line of research, we develop algorithms for providing hybrid consistency in message based systems. The algorithms share the following properties:
- every process holds a complete copy of the entire shared memory,
- messages are sent using an atomic broadcast mechanism, and
- weak operations are executed immediately while the execution time for strong operations is $O(d)$, which is optimal (up to a constant multiplication factor).

The algorithms differ one from another by the types of operations they support and whether weak and strong operations are allowed to access the same objects or not.

The assumption that every process holds a complete copy of the entire memory is necessary to guarantee that every weak operation can be executed immediately (on the local copy of the memory). Admittedly, for real systems in which the shared memory is quite large, this assumption is unrealistic. Therefore, to implement these algorithms on real machines, they should be modified so that processes hold only a partial copy of the memory. Similarly, other optimizations, e.g., packing several messages into one message, may be also required in order to achieve good performance. As this thesis concentrates on the more theoretical aspects of consistency conditions (and their implementations), we do not address these optimizations any further.

The first algorithm, is a generic algorithm that supports most types of strong operations as well as weak reads and weak writes. It assumes that weak and strong operations never access the same objects. However, this algorithm is still correct for objects that support read, write and read-modify-write operations, as well as for FIFO queues and stacks, even if weak reads (but not weak writes) access the same objects as strong operations. The second algorithm allows every object to be accessed by every type of operation. In that sense, it is much more flexible. However, this algorithm only supports reads and writes.

In a third, intermediate, algorithm, objects that are accessed by strong operations may be also accessed by either weak reads or weak writes (but not by both). However, this algorithm only supports read, write and read-modify-write operations. We have decided not to present this algorithm in this thesis due to its similarity to the second algorithm. The reader is referred to [32] for more details.

An important problem that should be investigated in order to evaluate the usefulness of hybrid consistency and the algorithms we develop for it, is to achieve lower bounds on possible implementations of hybrid consistency. For read/write objects, we show that if weak operations
are executed faster than the network delay, then strong operations must take at least $\Omega(d)$ time. We also prove that read-modify-write operations as well as dequeue and pop operations must take at least $\Omega(d)$ time, even if they are labeled as weak. These lower bounds imply that our algorithms are optimal up to a multiplicative constant.

The work described so far assumes that processes execute memory operations one at a time and in the order they appear in the program. However, in recent years, we have witnessed the development of new techniques and architectures that attempt to decrease the effective execution time of each instruction by executing several instructions in parallel, by using a pipeline and by allowing the computer to execute instructions out of order and even speculatively. Since all these techniques deviate from the sequential order of memory accesses specified by the program, we call them “non-sequential”.

To demonstrate the potential performance gains in non-sequential executions, consider the example program in Figure 1.1 and assume that the access time to $x$ is very long. This can happen, for example, if $x$ is located on a distant node in the network. Note, that the write to $z$ is invoked regardless of the outcome of the read from $x$. Hence, if out-of-order execution is allowed, we can start the write to $z$ before the read from $x$ is finished and before the write to $y$ has started and reduce the overall latency for memory accesses. Also, if we allow speculative executions, then the run-time system could predict that the read from $x$ will indeed return $v$ and start executing the write to $y$. If this prediction turns out to be correct, then again the overall latency for memory accesses is reduced. If the prediction turns out to be false, actions should be taken to eliminate the effects of the write to $y$; this could require a large amount of time. Hence, if we really want to gain performance from speculative executions, we must guarantee that most predictions will turn out to be correct. How to guarantee that most predictions turn out to be successful is a very interesting question, outside the scope of this

\begin{verbatim}
pl's program
tmp := read(x);
if tmp = v then
    write(y, u);
write(z, w);
\end{verbatim}

Figure 1.1: Example of out-of-order execution of operations
thesis.

Following the discussion above, it is clear that generalizing the definitions of known consistency conditions to support non-sequential executions is quite desirable. However, it turns out that this task is not trivial. Most of the previously defined consistency conditions consider only memory operations. Yet, programs include control operations, e.g., if statements, as well. Also, in order to allow branch predictions and speculative executions, some of the operations that result from incorrect branch predictions should not be considered when the views of the processes are constructed, even though these operations were executed.

We develop a general framework for defining consistency conditions that allow non-sequential executions. This framework combines the following two features: (1) The interface at which conditions are specified in our model is between the program and the system instead of specifying the internal behavior of the system; this is the interface used in the first part of this thesis. (2) The framework allows for arbitrary optimizations by the system, including especially speculative execution of memory accesses. Recent experiments have shown that if the program has complex control flow then only moderate speedup can be achieved by parallelism without speculative execution [40, 58].

Given this framework, we generalize three known consistency conditions—sequential consistency, weak consistency and hybrid consistency. Our extensions have two pleasing properties: (1) The conditions are defined for all programs, not just programs that satisfy certain properties. (2) We give a formal, yet intuitive, treatment of explicit control instructions, which are crucial for expressing the flow of control in a program and in analyzing its correctness on non-sequential implementations.

Unfortunately, it is more difficult to program memories that support hybrid consistency than to program memories that support sequential consistency, since the guarantees provided by the former are weaker than those provided by the latter. A way to cope with this problem is to develop rules and transformations for executing programs that were written for sequentially consistent memories on hybrid consistent memories. The benefit is that sequentially consistent executions are easier to reason about while hybrid consistency can be implemented more efficiently. In this thesis, we consider several techniques for turning programs written for sequential consistency into programs that work for hybrid consistency.

The first approach is based on statically labeling specific accesses as strong, according to
their type (read or write) and the object they access. We derive two specific techniques from this approach. First, we prove that programs in which all writes are strong run on hybrid consistent shared memory implementations as if they were sequentially consistent. Symmetrically, we show that programs in which all reads are strong run on hybrid consistent shared memory implementations as if they were sequentially consistent. (The last result depends on certain assumptions which might make it impractical.)

The second approach is based on the mutual exclusion paradigm and uses the first approach as a tool. One proposed way to program with hybrid-like consistency conditions [30, 33] is to protect accesses to shared data with critical sections and then to use strong operations in the mutual exclusion code and weak operations inside the critical section. When the critical sections are significant in size, the extra cost to execute the strong operations in the mutual exclusion algorithm is more than compensated for by the efficiency of the weak operations in the critical sections. We take a careful look at this paradigm for hybrid consistency and show that it is applicable, although care must be taken. Specifically, we show that many mutual exclusion algorithms designed to work on a sequentially consistent memory can be modified to work correctly on a hybrid consistent memory. The modification is to label all writes in the entry and exit sections as strong and all other operations as weak and to insert a dummy write in an appropriate place. However, this transformation only works for non-cooperative algorithms, in which processes do not participate in the mutual exclusion protocol unless they are actively vying for entry to the critical section.

The third approach for programming with hybrid consistency is to run data-race free programs. This approach was pioneered by [3, 4, 33, 34, 35]. (See Chapter 2 for a more detailed discussion.) A data race (also called access anomaly) occurs when two accesses to the same location occur, at least one is a write, and there is no synchronization between them. Data races in a program are considered bad practice: They add to the uncertainty of concurrent programs, beyond what is already implied by the fact that different processes may run at different rates and memory accesses may have variable duration. (Some debuggers even regard data races as bugs in the program.) Methods have been developed to detect and report data races, e.g., [6, 25, 28, 29, 47, 51, 52]. It is reasonable to assume that data-race free programs account for a substantial portion of all concurrent programs. We formally prove that data-race free programs run on hybrid consistent shared memory implementations as if they were sequentially consistent; this result is shown in our programmer-oriented framework.
Although many parallel programs are expected to be data-race free, we cannot ignore the drawbacks of these programs. Proving that a program is data-race free, even for restricted cases, is co-NP-hard [50]. Also, it is sometimes difficult to find the exact location of the data race in the program [51]. Our static methods provide an alternative and show that it is not necessary to make a program data-race free in order to guarantee correct operation. These methods are especially well-suited to applications in which reads greatly outnumber writes (or vice versa).

A natural question is whether our framework can be applied to describe existing systems and to investigate them. As a test case, we have chosen Alpha\(^1\), a novel 64-bit load/store RISC architecture, introduced by Digital Equipment Corporation in 1992 [1, 57]. Alpha was designed with particular emphasis on multiple instruction issue and support for multiprocessing. It is one of the first commercial microprocessors planned to serve as a building block for multiprocessors. Therefore, its architectural specification includes a detailed description of the shared memory behavior of multiprocessors based on Alpha chips. Although this specification is rigorous, it requires an understanding of many parts of the architecture and employs notions of ordering accesses, etc. As such it is very useful for hardware designers, but is not very convenient for programmers wishing to exploit the improved efficiency of Alpha.

Using our framework for non-sequential consistency conditions, we define alpha consistency, a formalization of the shared memory model of Alpha based multiprocessors. We use this definition to develop several programming techniques that allow the programmer to write programs assuming sequentially consistent hardware and run them on an Alpha based multiprocessors. Specifically, we show that every data-race free program runs on an alpha consistent hardware as if it was sequentially consistent. We also develop a general method that transforms any given non-cooperative algorithm for the mutual exclusion problem based on sequential consistency into a correct algorithm based on alpha consistency. Many of the proofs for alpha consistency were almost the same as the proofs for hybrid consistency. This suggests that the theoretical lessons learned in earlier parts of this thesis are relevant for practical systems.

The rest of this thesis is organized as follow: We discuss related work in Chapter 2. The basic framework is presented in Chapter 3. Weak conditions are defined and investigated in Chapter 4. Hybrid consistency is defined in Chapter 5. We present the algorithms for providing hybrid consistency in Chapter 6, while the lower bounds for implementing hybrid consistency

---

\(^1\)Alpha is a trademark of Digital Equipment Corporation.
are given in Chapter 7. The framework for defining consistency conditions that allow non-
sequential executions is presented in Chapter 8 where we also generalize the definitions of
sequential consistency, weak consistency and hybrid consistency. The programming techniques
for hybrid consistency are discussed in Chapter 9. Alpha consistency and the programming
techniques for it appear in Chapter 10. We conclude with a summary and discussion in
Chapter 11.
Chapter 2

Related Work

2.1 Overview

Traditionally, distributed shared memories were expected to behave, from the programmer's point of view, as real shared memories. That is, they were assumed to support strong conditions like sequential consistency [43] or linearizability [38].

The first theoretical work to note a fundamental difference between sequential consistency and linearizability was carried out by Herlihy and Wing [38]. They showed that it is possible to implement a linearizable memory as a collection of smaller independent linearizable implementations while it is not possible to implement sequential consistency in a similar manner. This result suggests that linearizability might be easier to implement, especially when different optimizations should be applied to different objects, e.g., in systems where each object may exhibit a different access pattern [46].

On the other hand, Attiya and Welch have shown that in asynchronous systems, assuming that access durations and patterns are roughly the same for all processes and objects, sequential consistency may be implemented more efficiently than linearizability [19]. These results were extended by Mavronicolas and Roth [46] and later on by Kosa [39].

Lipton and Sandberg were the first to explicitly show that even sequential consistency cannot be implemented efficiently enough so highly parallel multiprocessors can support it [44], although this has been considered “common knowledge” among architects even earlier.
This result was later formalized and extended in [19, 39, 46]. This inherent inefficiency in strong conditions has led to the definition of several weak conditions, such as Pipelined RAM [44], processor consistency [36] and causal memory [10]. Alas, many designers felt that the programming model provided by weak conditions is too complicated to work with. Instead, they started developing and implementing hybrid conditions. These conditions distinguish between weak or data operations and strong or synchronization operations. Weak operations provide weak semantics and can be implemented efficiently while strong operations provide the strong semantics that is sometimes necessary to write correct programs.

Dubois, Scheurich and Briggs started the research on hybrid conditions when they presented the definition of weak ordering [31]. Gibbons, Merritt and Gharachorloo continued with this approach by defining release consistency, a formal definition of the Stanford DASH multiprocessor [34, 35] using an I/O automaton. This definition makes a further classification of strong (or synchronization) operations into release and acquire; a release operation orders only the previous operations while an acquire operation orders only the following operations. To ease programming with release consistency, Gibbons, Merritt and Gharachorloo suggested a programming model called PL-programs and showed that every PL-program runs on a release consistent hardware if it was sequentially consistent [34, 35]. This result allows programmers to continue writing for sequentially consistent memories even when the programs run on release consistent hardware.

Adve and Hill used the approach that programmers always want to work assuming sequentially consistent memories in a formalism called SCNF [2, 5]. By this approach, a consistency condition is a contract between software and hardware. That is, hardware must behave as if it was sequentially consistent for all programs that obey certain properties; the difference between various consistency conditions is the required properties from programs. Examples of such conditions include, e.g., DRF0 [3], DRF1 [4], PL_{pc1} and PL_{pc2} [2]. For each of these programming models, Adve and Hill introduced sufficient conditions for hardware; hardware that follows these conditions executes programs that obey the corresponding programming model as if it was sequentially consistent.

Singh developed a framework for defining consistency conditions using partial orders [56]. He also proved sufficient conditions for executions generated by optimized hardware, e.g., Pipelined RAM, causal memory and hybrid consistency, to be sequentially consistent. Unlike the approach taken by Gibbons, Merritt and Gharachorloo and by Adve and Hill, the approach
taken by Singh examines the executions generated by the optimized hardware and not the programs that should be run on it.

Herlihy and Wing developed a model for describing memory models based on local histories in [38]. That is, each process is associated with a local history which is a sequence of (complete) memory operations. An execution in this model is a collection of local histories, one for each process. This model was used by Ahamad, Neiger, Kohli, Burns and Hutto to define causal memory in [11]. This definition is based on the assumption that no value is written more than once to the same memory location. Using this definition of causal memory, Ahamad et al. developed several programming techniques for it. Some of these techniques depend on the addition of await statements and/or semaphore operations to the basic definition of causal memory.

Recently, Corella, Stone and Barton provided an axiomatic definition of the memory model of PowerPC based multiprocessors [26]. They showed that if all accesses to shared data on a PowerPC based multiprocessor are protected with lock/unlock sequences, then the result is as if the hardware was sequentially consistent.

2.2 Comparison with Our Work

2.2.1 The Interface and Definitions

Most previous formalisms for defining consistency conditions were developed from the architect's point of view. This includes the work of Dubois, Schurich and Briggs [31] and the work of Gibbons, Merritt and Gharachorloo [34, 35]. Hence, definitions developed in these frameworks are not natural for programmers and theoreticians. The framework of Adve and Hill [2, 3, 4] is programmer oriented. However, it does not give a crisp guarantee about the behavior of hardware for programs that do not obey certain conditions, even if these programs make sense and can provide better performance; examples of such programs are in Sections 9.2 and 10.2 of this thesis. The framework of Singh is programmer oriented and allows defining consistency conditions for all programs [56]. However, since it is based on partial orders, it is harder to guarantee liveness in this framework, i.e., that an operation will eventually be seen by processes other than its invoking process.
Our work complements previous work on hybrid conditions by presenting an abstract and formal definition at an interface that is more comfortable for programmers without restricting the programming style. Also, our framework is based on sequences of operations and not partial orders and, hence, it already encompasses certain liveness properties that need special treatment in frameworks that are based on partial orders. Our approach is driven by two basic assumptions: (a) Programmers prefer to write programs and prove their correctness when the exact behavior of hardware does not have to be taken into account. (b) Programmers should be free to write and optimize their programs in any way they see fit. For those wishing to continue working assuming sequentially consistent memories, we develop programming techniques that allow it. On the other hand, those wishing to develop other programs are provided with a comfortable interface and definitions that allow them to do so.

Admittedly, our work is quite theoretical. As a result, our definitions may occasionally disallow some hardware optimizations and they are not very natural for architects. In particular, it is not always clear how to implement a consistency condition that is defined in our framework. However, our results have practical implications for software, as demonstrated in Chapter 10, where we develop programming techniques for DEC-Alpha based multiprocessors.

2.2.2 Support for Non-Sequential Execution of Operations

Several previous formalisms for defining consistency conditions have addressed the issue of non-sequential executions. For example, the work of Adve and Hill deals formally with non-sequential executions of memory operations. However, while the sufficient conditions for DRF1 [4, 5] includes a formal treatment of control, this condition is based on the notion of a read operation controlling a write operation by the same processor. This is a semantic notion and it is not proven that it captures all the possible ways one operation can control another. In contrast, our approach is syntactic, based on using the actual control instructions in the program, and thus is safer. Other previous work on specifying consistency conditions has either totally ignored control instructions [7, 17, 19, 38, 43] or has merely made the intuitive informal requirement that uniprocessor control dependences are preserved [3, 33, 34, 35].

The formal definition of release consistency introduced in [35] does not allow non-sequential executions. A later definition by Gibbons and Merritt [34] deals formally and explicitly with pipelining of memory operations, and presents definitions for sequential consistency and release
consistency that allow pipelining of memory operations. However, the program is not explicitly modeled in their definitions, and it is not clear how the intended semantics of the program is preserved. In contrast, our framework explicitly models the program and the run-time system executing it. Unlike our work, their results do not encompass arbitrary out-of-order or speculative execution of operations.

2.2.3 Programming Techniques for Hybrid Conditions

The approach taken by Adve and Hill for running data-race free programs or programs that obey other similar programming models, is to develop the programming model first and then to show sufficient conditions for hardware on which these programs will run [2, 3, 4]. Our approach and the one taken by Gibbons, Merritt and Gharachorloo [34, 35] is to take existing consistency conditions and show that programs that obey some variant of the data-race free model runs on hardware that implements the consistency condition as if it was sequentially consistent. However, when compared to the work of Gibbons, Merritt and Gharachorloo [34, 35], our framework yields much simpler proofs and a more comprehensive support for non-sequential executions.

Several papers on relaxed consistency conditions refer to the method of programming with critical sections as a justification for the separation of strong and weak operations. In [33], it is argued (page 19):

“For example, a large class of programs are written such that accesses to shared data are protected within critical sections. Such programs are called synchronized programs, whereby writes to shared locations are done in a mutually exclusive manner (no other reads or writes can occur simultaneously). In a synchronized program, all accesses (except accesses that are part of the synchronization constructs) can be labelled as ordinary.”

In Section 9.2 we show a more efficient way to utilize synchronized programs. If the mutual exclusion algorithm is non-cooperative, then it is not necessary to label reads that are part of the synchronization constructs (i.e., entry and exit sections) as strong (provided that an extra write is added).


2.2.4 Describing Real Architectures

Our work on alpha consistency joins two previous works in attempting to provide a formal memory model for an existing architecture and using this model to develop programming techniques for systems based on this architecture. The first is for the Stanford DASH multiprocessor [34, 35] and the second is for PowerPC based multiprocessors [26], both mentioned above.

The PL-programs used for the DASH multiprocessor are quite similar to generalized data-race free programs, described in Chapter 10. There is a slight difference, however, between the way release and acquire operations and the interaction between them are defined in alpha consistency and in release consistency. In alpha consistency, release and acquire are sequences of operations while in release consistency release and acquire are assumed to be indivisible atomic operations. In addition, PL-programs require a pairing to be defined between release and acquire operations; alpha consistency does not impose a similar requirement.

Protecting all accesses with lock/unlock sequences, as suggested by Corella, Stone and Barton [26], is a special case of generalized data-race free programs. In Chapter 10, we specify MBV and MBP sequences, implementing Dijkstra’s V and P operations on semaphores [27]. As defined in [26], a lock sequence is a special case of an MBP sequence and an unlock sequence is a special case of an MBV sequence.
Chapter 3

The Basic Framework

In this chapter we present a framework for defining consistency conditions in a programmer oriented manner.

3.1 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 \( P = \{p_1, \ldots, p_n\} \) connected via a communication network; each application program runs on a different node. The shared memory abstraction is implemented by a memory consistency system (MCS), which uses local memory and some protocol executed by the MCS processes (one at each node). A correctness condition is defined at the interface between the application programs (written by the user) and the MCS processes (supplied by the system). Thus, the MCS must provide the proper semantics when the values of the responses to calls are considered, throughout the network. An illustration of the system architecture is given in Figure 3.1.

The following \textit{events} may occur at the MCS process on node \( i \):

1. \textit{Call events}: The initiation of operations by the application program according to their specification. For example, in the case of objects that support read, write and read-modify-write operations, the call events are \( \text{Read}_i(X) \), \( \text{Write}_i(X, v) \) and \( \text{RMW}_i(X, f) \) for all objects \( X \), values \( v \) and functions \( f \).
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 objects that support read, write and read-modify-write operations, the response events are $\text{Return}_i(X, v)$, $\text{Ack}_i(X)$ and $\text{Return} & \text{Ack}_i(X, v)$ for all objects $X$ and values $v$.

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

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

The call and message-receive events are **interrupt events**.

An **MCS process** is an automaton with a (possibly infinite) set of states, including an initial state, and a transition function. (We sometime use *process* or *MCS* for short.) 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 (possibly empty) sequences of steps such that

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

An **execution** of an MCS is a set of histories, one for each process in $P$, in which there is a one-to-one correspondence from the messages received by $p_i$ from $p_j$ onto the messages sent by $p_i$ to $p_k$, for any pair of 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.)

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

Given a partial execution $\sigma'$, denote by $\#_{\text{call}}(\sigma')$ the number of call events invoked at node $i$ during $\sigma'$; denote by $\#_{\text{response}}(\sigma')$ the number of response events generated at node $i$ during $\sigma'$. An execution $\sigma$ is *admissible* if the following conditions hold:
1. For every $i$ and $j$, every message in $\sigma$ from $p_i$ to $p_j$ has its delay in the range $[0, d]$, for a fixed nonnegative integer $d$. (This is a restriction on the network.)

2. For every partial execution $\sigma'$ of $\sigma$ and every $i$, $0 \leq \#_{\text{call}}(\sigma') - \#_{\text{response}}(\sigma') \leq 1$. (This is a restriction on the application program.)

From now on, we assume all executions are admissible. Each pair of a call event and a subsequent matching response event forms an operation. The call event marks the start of the operation, while the response event marks the end of that operation. An operation $op$ is invoked when the application process issues the appropriate call for $op$; $op$ terminates when the MCS process issues the appropriate response for $op$. An operation is pending if it is invoked and is not terminated. Note that since we assumed that all executions are admissible, there can be at most one pending operation per process, i.e., an application program does not invoke a new operation before the previous one has terminated.

Every object is assumed to have a sequential specification (cf. [38]) defining a set of operations, which are ordered pairs of call and response events, and a set of operation sequences, which are the allowable sequences of operations on that object. For example, in the case of a read/write object, the ordered pair of events $[\text{Read}_i(x), \text{Return}_i(x, v)]$ forms an operation for any $p_i$, $x$, and $v$, as does $[\text{Write}_i(x, v), \text{Ack}_i(x)]$. The set of operation sequences consists of all sequences in which every read operation returns the value of the latest preceding write operation (the usual read/write semantics). Note that sometimes the sequential specification of an object may depend on the number of processes in the system.

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

Given an execution $\sigma$, let $\text{ops}(\sigma)$ 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 $\sigma$ induces a partial order, $\preceq$, on the operations that appear in $\sigma$; $op_1 \preceq op_3$ if the response event of $op_1$ appears in $\text{ops}(\sigma)$ before the call event of $op_3$.

Given an execution $\sigma$, a sequence of operations $\tau$ is a serialization of $\sigma$ if it is a permutation of $\text{ops}(\sigma)$. A serialization $\tau$ of $\sigma$ is a linearization if it extends $\preceq$; that is, if $op_1 \preceq op_3$ then $op_1 \tau op_3$. 

21
Let $\sigma$ be an execution. Denote by $\sigma|j$ the restriction of $\sigma$ to call and response events of $p_j$; denote by $\sigma|(j + op)$ the restriction of $\sigma$ to call and response events for operations of type $op$ (by any process) or for operations by $p_j$; denote by $\sigma|x$ the restriction of $\sigma$ to call and response events to and from object $x$. Similarly, let $\tau$ be a sequence of operations. Denote by $\gamma j$ the restriction of $\tau$ to operations invoked by $p_j$; denote by $\gamma x$ the restriction of $\tau$ to operations that access object $x$.

The semantics of an object can affect its possible implementations. In particular, some operations may not be able to terminate until the object they access holds a certain value or until some other operations are invoked. For example, the operation $\text{await}(x = 1)$ can not terminate until the value of $x$ is 1. Hence, if the value of $x$ is different than 1 when the $\text{await}$ is invoked, then it cannot terminate until another write updates $x$ with 1. On the other hand, the termination of operations like read, write, read-modify-write, push and pop does not depend on an object having a certain value, or another operation being executed.

Formally, we assume that each call event is associated with a (possibly empty) list of input variables and every response event is associated with a (possibly empty) list of output variables. A typical input list would usually consist of an object to be accessed and a value or a function to be applied to the object; a typical output list would usually consist of the object that was accessed and a value returned from this object.

**Definition 3.1** An object $x$ is responsive if for any initial value for $x$, any operation type $op$ supported by $x$, any number of processes in the system and any input list in $\text{in\_list}$, there exists an output list out $\text{\_list}$ such that $op(\text{in\_list}, \text{out\_list})$ is in the sequential specification of $x$.

The following notation is used in the rest of this thesis. We denote by $op_i$ an operation invoked by $p_i$. We use superscripts, e.g., $op_1^1, op_1^2, \ldots$, to distinguish between operations invoked by the same process. We sometimes use a shorthand notation for read, write and read-modify-write operations and denote by $r_i(x,v)$ a read operation invoked by process $p_i$ returning $v$ from $x$; we denote by $w_i(x,v)$ a write operation invoked by process $p_i$ writing $v$ to $x$; $rmw_i(x,f,v)$ is a read-modify-write invoked by process $p_i$ returning $v$ from $x$ and writing $f(v)$ to $x$.  

22
3.2 Defining Consistency Conditions

A consistency condition is defined in our framework by describing the executions that are allowed by the condition. An allowed execution is defined by describing for each process $p_i$ a view $\tau_i$ of the operations in this execution. An implementation obeys a certain consistency condition if and only if every execution generated by it is allowed by the consistency condition. This means that consistency conditions differ one from another only in the requirements they impose on the possible views of the processes. Therefore, programmers need only care about the possible orderings of the operations and not the exact hardware optimizations.

Our definition style is very flexible and it can be applied to many different consistency conditions. To demonstrate our definition style, we now define two well known consistency conditions; sequential consistency and linearizability. Other consistency conditions are defined later in the thesis.

Definition 3.2 (Sequential consistency) An execution $\sigma$ is sequentially consistent if there exists a legal serialization $\tau$ of $\sigma$, such that for each process $p_j$, $\sigma \upharpoonright j = \tau \upharpoonright j$.

Definition 3.3 (Linearizability) An execution $\sigma$ is linearizable if there exists a legal linearization $\tau$ of $\sigma$, such that for each process $p_j$, $\sigma \upharpoonright j = \tau \upharpoonright j$.

Note that in the above definitions, $\tau$ serves as the view of every process, meaning that all processes must view all operations in the same order. This can be seen as if there is an implicit requirement that for every process $p_j$, there exists a legal sequence of operations $\tau_j$ such that $\tau_j = \tau$. However, the requirement that the views of all the processes are the same does not always exist. For example, weak consistency and hybrid consistency, defined later in this thesis, allow different processes to view at least some of the operations in a different order.

3.2.1 Examples

Consider the executions in Figure 3.2. (In these executions we assume that $x$ and $y$ are initialized to 0.)

Execution $\sigma_1$: $\sigma_1$ is linearizable since we can take $\tau = w_1^1(x,1), w_2^1(y,1), r_1^2(y,1), r_2^2(x,1)$. Clearly, $\tau$ is a legal linearization. Note that this implies that $\sigma_1$ is sequentially consistent, too.
(a) Execution $\sigma_1$

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$p_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1^1(x, 1)$</td>
<td>$w_3^3(y, 1)$</td>
</tr>
<tr>
<td>$r_1^2(y, 1)$</td>
<td>$r_2^2(x, 1)$</td>
</tr>
</tbody>
</table>

(b) Execution $\sigma_2$

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$p_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1^1(x, 1)$</td>
<td>$r_2^2(x, 0)$</td>
</tr>
</tbody>
</table>

(c) Execution $\sigma_3$

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$p_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1^1(x, 1)$</td>
<td>$w_3^3(y, 1)$</td>
</tr>
<tr>
<td>$r_1^2(y, 0)$</td>
<td>$r_2^2(x, 0)$</td>
</tr>
</tbody>
</table>

(d) Execution $\sigma_4$

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$p_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1^1(x, 1)$</td>
<td>$w_3^3(x, 2)$</td>
</tr>
<tr>
<td>$r_1^2(x, 2)$</td>
<td>$r_2^2(x, 1)$</td>
</tr>
</tbody>
</table>

(e) Execution $\sigma_5$

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1^1(y, 1)$</td>
<td>$w_3^3(y, 1)$</td>
<td>$r_3^3(x, 2)$</td>
</tr>
<tr>
<td>$w_2^2(x, 2)$</td>
<td>$w_3^3(x, 2)$</td>
<td>$r_3^3(y, 0)$</td>
</tr>
</tbody>
</table>

Figure 3.2: Examples of executions
Execution $\sigma_2$: $\sigma_2$ is not linearizable since in every linearization of $\sigma_2$, $r_2^t(x,0)$ must be ordered after $w_1^t(x,1)$, meaning that the linearization is not legal. In order to show that $\sigma_2$ is sequentially consistent, we can take $\tau = r_2^t(x,0), w_1^t(x,1)$. Clearly, $\tau$ is a legal serialization of $\sigma_2$ and for every $i$, $\sigma_2[i] = \tau[i]$.

Execution $\sigma_3$: $\sigma_3$ is not sequentially consistent since in every legal serialization $\tau$ of $\sigma_3$, $r_2^t(x,0)$ must be ordered before $w_1^t(x,1)$ and $r_1^t(y,0)$ must be ordered before $w_1^t(y,1)$. Hence, either $r_2^t(x,0)$ should be ordered before $w_1^t(y,1)$ or $r_1^t(y,0)$ should be ordered before $w_1^t(x,1)$. Note that this implies that $\sigma_3$ is not linearizable, too.

Execution $\sigma_4$: $\sigma_4$ is not sequentially consistent since in every legal serialization $\tau$ of $\sigma_4$, $r_2^t(x,1)$ must be ordered immediately after $w_1^t(x,1)$ and $r_1^t(x,2)$ must be ordered immediately after $w_1^t(x,2)$. Hence, either $r_2^t(x,1)$ should be ordered before $w_1^t(x,2)$ or $r_1^t(x,2)$ should be ordered before $w_1^t(x,1)$.

Execution $\sigma_5$: $\sigma_5$ is not sequentially consistent since in every legal serialization $\tau$ of $\sigma_5$, $r_2^t(y,0)$ must be ordered before $w_1^t(y,1)$ and $w_1^t(y,1)$ while $r_3^t(x,2)$ must be ordered after $w_1^t(x,2)$ or $w_2^t(x,2)$. Hence, either $r_2^t(y,1)$ should be ordered before $r_3^t(x,2)$, $w_1^t(x,2)$ should be ordered before $w_1^t(x,1)$ or $w_2^t(x,2)$ should be ordered before $w_1^t(x,1)$.

These examples are used later in this thesis to explain other consistency conditions.
Chapter 4

Weak Conditions

In this chapter we define *weak consistency*, capturing a large class of weak conditions. These conditions provide very little guarantee on the order in which operations of one process are viewed by other processes. We show that weak conditions are, in some sense, too weak to solve some of the fundamental problems in concurrent programming.

4.1 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 in which they were issued. Formally:

**Definition 4.1 (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 $\tau_j | j = \sigma | j$.

Note that $\tau_j$ represents the ordering of operations that is consistent with the view of $p_j$. An implementation of weak consistency is provided by the Pipelined RAM of Lipton and Sandberg [44]. In this implementation, each process holds a complete copy of the entire

---

1In fact, the Pipelined RAM of Lipton and Sandberg supports a slightly stronger consistency condition than weak consistency. However, this only strengthen the impossibility result of this chapter.
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.

4.1.1 Examples

It is clear from Definition 4.1 that every sequentially consistent execution is also weakly consistent. Hence, Executions $\sigma_1$ and $\sigma_2$ in Figure 3.2 are weakly consistent. We now show that Executions $\sigma_3$, $\sigma_4$ and $\sigma_5$ in Figure 3.2 are weakly consistent, too.

**Execution $\sigma_3$:** For this execution, we can take

$$\tau_1 = r_2^1(x, 0), w_1^1(x, 1), r_1^1(y, 0), w_2^1(y, 1)$$

and

$$\tau_2 = r_1^1(y, 0), w_2^1(y, 1), r_2^2(x, 0), w_1^1(x, 1).$$

**Execution $\sigma_4$:** For this execution, we can take

$$\tau_1 = w_1^1(x, 1), r_2^1(x, 1), w_2^1(x, 2), r_2^2(x, 2)$$

and

$$\tau_2 = w_2^1(x, 2), r_1^2(x, 2), w_1^1(x, 1), r_2^2(x, 1).$$

**Execution $\sigma_5$:** For this execution, we can take

$$\tau_1 = w_2^2(x, 2), r_3^2(x, 2), r_2^3(y, 0), w_1^1(y, 1), w_2^2(y, 1), w_1^2(x, 2),$$

$$\tau_2 = w_1^2(x, 2), r_2^1(x, 2), r_3^2(y, 0), w_1^1(y, 1), w_2^1(y, 1), w_2^2(x, 2)$$

and

$$\tau_3 = w_2^1(x, 2), r_2^1(x, 2), r_2^3(y, 0), w_1^1(y, 1), w_2^1(y, 1), w_2^2(x, 2).$$

**Execution $\sigma_6$ (Figure 4.1):** This execution is not weakly consistent, even if we assume that the initial value of $x$ is 0. In $\sigma_6$, $p_1$ executes $w_1^1(x, 1)$ while $p_2$ executes an infinite number of reads, $r_2^i(x, 0)$, $i = 1, 2, \ldots$. We claim that there does not exist a legal sequence of operations $\tau_1$ as in the definition of weak consistency. This is because in every such sequence, $w_1^1(x, 1)$ must
be ordered after some finite number of reads by $p_2$. Hence, there will be an infinite number of reads of the form $r_i^j(x, 0)$ ordered after $w_i^1(x, 1)$, implying that the sequence is not legal. This example also demonstrates the liveness requirement included in weak consistency, i.e., that every operation is eventually seen by every process.

4.2 Mutual Exclusion Using Weak Consistency

Lipton and Sandberg ([44]) 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.

Formally, an algorithm for mutual exclusion consists of four disjoint sections—entry, critical, exit and remainder (cf. [54]). 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. Roughly speaking, 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.

The above definition of mutual exclusion assumes that the order in which operations are executed in “real-time” reflects the order in which they are viewed by the processes. However,
our formalism for defining consistency conditions puts restrictions only on the order in which operations are viewed, which may not correspond directly to the order in which they are executed. Hence, it is unclear that an algorithm which violates mutual exclusion (according to the above definition) on weakly consistent hardware also violates logical mutual exclusion. In this section we formally define logical mutual exclusion and show that there does not exist a non-cooperative algorithm that provides logical mutual exclusion based on weak consistency.

Formally, given a mutual exclusion algorithm (program) $A$, consider an execution $\sigma$ and let $CS_i^k$ be the set of operations invoked by some process $p_i$ during the $k$th time that $p_i$ executes the critical section in $\sigma[i]$. Recall that a consistency condition is defined by describing the executions it allows and that an allowed execution is defined by describing for each process a view of the operations in the execution.

$A$ guarantees logical mutual exclusion based on a consistency condition $C$ provided that the following holds. Let $\sigma$ be an execution of $A$ that is allowed by $C$ and let $\{\tau_q\}_{q=1}^n$ be a set of sequences as required in the definition of $C$ (the views of the different processes). Consider the $CS_i^k$'s induced by $\sigma[i]$.

**Logical mutual exclusion:** For any four operations $op_i^1, op_i^2 \in CS_i^k$ and $op_j^1, op_j^2 \in CS_j^l$, $op_i^1 \xrightarrow{\tau_q} op_i^2$ if and only if $op_j^1 \xrightarrow{\tau_q} op_j^2$ for all $\tau_q$. As before, this implies that there is a total order on all critical section executions; furthermore, this ordering is the same in all $\tau_q$'s.

**Deadlock freedom:** For every $q$, if $\tau_q$ is infinite and there exists a process $p_i$ that is in its entry section from some point on in $\tau_q$, then there is another process $p_j$ that enters (and leaves) its critical section infinitely often in $\tau_q$.

Definition 3.2 of sequential consistency and Definition 3.3 of linearizability define only one sequence of operations $\tau$. However, we remind the reader that these definitions can be phrased such that for every process $p_j$, there exists a sequence of operations $\tau_j = \tau$.

To prohibit solutions in which some processes must participate in the mutual exclusion algorithm, even if these processes do not wish to enter the critical section, e.g., 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. Formally, the exclusion set of a mutual exclusion algorithm $A$ is the set of shared variables read inside the entry or exit sections of $A$; this set is denoted $exc(A)$. 

29
Definition 4.2  A mutual exclusion algorithm $A$ is non-cooperative if every process which executes the critical section or the remainder section of $A$ does not write any variable in $exc(A)$; otherwise, the algorithm is cooperative.

In particular, all solutions in which a process wishing to enter the critical section must communicate with a fixed subset of the processes, even if these processes are not trying to enter (or are inside) the critical section, are cooperative. In particular, centralized solutions (like the one presented in [44]) are cooperative.

Finally, we assume that a solution to the mutual exclusion problem must operate correctly regardless of the code of the critical section. The main result of this chapter is:

Theorem 4.1 Any algorithm that provides logical mutual exclusion based on weak consistency must be cooperative.

Proof: Consider a system with two processes $p_1$ and $p_2$, and assume, by way of contradiction, that there exists a non-cooperative algorithm $A$, based on weak consistency, which solves the mutual exclusion problem. $A$ must operate correctly under any implementation of weak consistency, in particular under the Pipelined RAM implementation of weak consistency [44]. In the proof, we assume that all operations can be executed as fast as we want. Thus, we assume there is an implementation of weak consistency with this property; the Pipelined RAM is an example of such an implementation. The Pipelined RAM assumes that every process has a complete copy of the entire memory; operations are executed on the local copy of the memory and return immediately.

Consider the following Pipelined RAM execution $\sigma'_1$ of an instance of $A$ in which that first operation in the critical section is a write. In $\sigma'_1$, $p_1$ and $p_2$ take steps at each real time $r$, for any integer $r \geq 0$. 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 of its own steps, $p_1$ makes the transition from the entry section to the critical section. Hence, there exists a finite step, $t_1$, in which $p_1$ executes the first (write) operation in the critical section. Denote this write by $w_1$. Since $p_2$ is in the remainder section, and since $A$ is non-cooperative, $p_2$ does not write to any variable in $exc(A)$ during $\sigma'_1$. Let $\sigma_1$ be the steps in $\sigma'_1$ up to and including
time \( t_1 \), when all steps are rescheduled to occur at time \( < d \). This can be done since each process takes only a finite number of steps in a finite time. Since in \( \sigma'_1 \), \( p_1 \) does not write any variable in \( \text{exc}(A) \) while \( p_2 \) reads only variables from \( \text{exc}(A) \), it follows that \( p_1 \)'s history in \( \sigma'_1 \) is equal to \( p_2 \)'s history in \( \sigma_1 \), until time \( t_1 \), except for the timing of events. (See Figure 4.2(a).)

Now consider the symmetric Pipelined RAM execution \( \sigma'_2 \) of the same instance of \( A \). In \( \sigma'_2 \), \( p_2 \) and \( p_1 \) take steps at each real time \( r \), for any integer \( r \geq 0 \). 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 of its own steps, \( p_2 \) makes the transition from the entry section to the critical section. Hence, there exists a finite step, \( t_2 \), in which \( p_2 \) executes the first (write) operation in the critical section. Denote this write by \( w_2 \). Since \( p_1 \) is in the remainder section, and since \( A \) is non-cooperative, \( p_1 \) does not write to any variable in \( \text{exc}(A) \) during \( \sigma'_2 \). Let \( \sigma_2 \) be the steps in \( \sigma'_2 \) up to and including time \( t_2 \), when all steps are rescheduled to occur at time \( < d \). This can be done since each process takes only a finite number of steps in a finite time. Since in \( \sigma'_2 \), \( p_1 \) does not write any variable in \( \text{exc}(A) \) while \( p_2 \) reads only variables from \( \text{exc}(A) \), it follows that \( p_1 \)'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. (See Figure 4.2(b).)

Note that no message is ever received in \( \sigma_1 \) and \( \sigma_2 \). Hence, the partial execution \( \sigma \) obtained from \( \sigma_1 \) by replacing \( p_1 \)'s history with \( p_2 \)'s history in \( \sigma_2 \) can be extended into a finite weakly consistent execution \( \sigma' \) such that at the end of \( \sigma' \) both \( p_1 \) and \( p_2 \) are inside the critical section. (See Figure 4.2(c).) Since \( A \) is non-cooperative, neither \( w_1 \) nor \( w_2 \) write to an object that is being read from during \( \sigma' \). Hence, there exists a pair of sequences \( \tau_1 \) and \( \tau_2 \) for \( \sigma' \), from the definition of weak consistency, such that \( w_1 \) and \( w_2 \) are the last two operations in \( \tau_1 \) and \( \tau_2 \). Moreover, since both \( w_1 \) and \( w_2 \) are writes, and since these are the last two operations in \( \tau_1 \) and \( \tau_2 \), we may assume that \( w_1 \xrightarrow{\tau_1} w_2 \) and \( w_2 \xrightarrow{\tau_2} w_1 \). Since \( w_1 \) and \( w_2 \) are executed inside two different critical sections, this is a contradiction to the assumption that \( A \) guarantees logical mutual exclusion based on weak consistency.

This proof relies on the fact that there is no lower bound on the step time of the processes. A similar proof can be used 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...
Figure 4.2: Executions used in the proof of Theorem 4.1
(The delay of each message is exactly $d$)
propagation time, then the claim does not hold; that is, one can design a non-cooperative algorithm for the mutual exclusion problem. For example, it is possible to give time slots to each process. Then a process is allowed to execute critical sections only during its own time slots. The problem with such algorithms is that they must rely on timing information to provide a safety property, a behavior which is better avoided. Note also that our proof holds for both deterministic and randomized algorithms; this only makes the impossibility result stronger.

**Remark:** In a sense, some sort of cooperation must exist at some level. If a consistency condition is weak enough to be implemented in such a way that the hardware is “non-cooperative”, then the software must compensate for that and use cooperative algorithms to achieve synchronization. On the other hand, if the consistency condition is strong enough to support non-cooperative synchronization algorithms, then its implementation must be cooperative (cf. the atomic broadcast used in the algorithms of Chapter 6). Although both alternatives are acceptable, designers should be aware of this tradeoff.

### 4.3 Other Weak Conditions

In this section we discuss other weak conditions. We show that either the impossibility result holds for them, or they cannot be implemented efficiently, i.e., in every implementation of these conditions, operations cannot be executed faster than the network delay.

#### 4.3.1 Causal Memory

A consistency condition called causal memory was suggested in [10, 11]. For the definition of causal memory, we assume that no value is written more than once to the same memory location. Given an execution \( \sigma \), an operation \( op_i^1 \) precedes \( op_j^2 \) (denoted \( op_i^1 \prec \sigma \rightarrow op_j^2 \)) if either \( i = j \) and \( op_i^1 \rightarrow op_j^2 \), or \( op_i^1 = w_i(x, v) \) for some memory location \( x \) and value \( v \) and \( op_j^2 = r_i(x, v) \). Let \( \prec \sigma \rightarrow \) denote the transitive closure of \( \prec \sigma \rightarrow \).

**Definition 4.3 (Causal memory)** An execution \( \sigma \) is causal if for every process \( p_i \) there exists a legal serialization \( \tau_i \) of \( \sigma | (i + \text{writes}) \) such that \( \tau_i \) is consistent with \( \prec \sigma \rightarrow \).
An implementation of causal memory for message based systems is presented in [11]. Like in the implementation of Pipelined RAM discussed above, every process holds a complete copy of the entire memory and local operations are executed instantaneously on the local copy of the memory. Whenever a write operation is invoked, an update message is sent to all other processes. The only difference between the implementation in [11] and the implementation of Pipelined RAM is the contents of the update messages and the rules for updating objects when such a message is received. However, the proof of Theorem 4.1 does not consider the contents of update messages or the rules for updating objects when these messages are received. Thus, Theorem 4.1 holds for the implementation of causal memory given in [11] and, hence, for causal memory, too.

### 4.3.2 Processor Consistency

Informally, processor consistency requires that the operations of each process will appear to be executed by all the processes in the order they were invoked [36]. The following is a formal definition which attempts to capture this intuitive semantics.

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

**Definition 4.4 (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 [44] is processor consistent, according to Definition 4.4. Thus, Theorem 4.1 holds for Definition 4.4 of processor consistency.

Recently, two other formal definitions, PCG and PCD, that have a slightly different interpretation of processor consistency, were suggested [9]. Both PCG and PCD are stronger than Definition 4.4 and the impossibility result does not hold for them. In particular, Peterson’s

---

2PCD is supported by the Stanford DASH multiprocessor [33]. The D in PCD stands for DASH and the G in PCG stands for Goodman.
non-cooperative algorithm [53] guarantees mutual exclusion based on these conditions [9]. Below we formalize PCG and PCD in our framework. We show that for any implementation of PCG, the time required to execute two write operations and one read operation is at least $d$, where $d$ is the network delay. Similarly, we show that for any implementation of PCD, the time required to execute two write operations and two read operations is at least $d$.

Intuitively, PCG requires that each process will view its own operations in the order they were invoked and that all processes will agree on the order of write operations to the same object. Formally:

**Definition 4.5 (PCG)** An execution $\sigma$ is PCG if for every process $p_j$ there exists a legal serialization $\tau_j$ of $\sigma[j + \text{writes}]$ such that

1. $\tau_j[j = \sigma[j$.
2. For every two write operations $w_1^j(x, v)$ and $w_2^j(x, u)$, $w_1^j(x, v) \xrightarrow{\tau_j} w_2^j(x, u)$ if and only if $w_1^j(x, v) \xrightarrow{\tau_j} w_2^j(x, u)$ for every process $p_j$, object $x$ and values $v$ and $u$.

Given a particular MCS, an object $x$ implemented by it, and an operation type $op$ on $x$, we denote by $|op(x)|$ the maximum time taken from the invocation to the termination of an operation of type $op$ on $x$ in any execution. We denote by $|op|$ the maximum of $|op(x)|$ over all objects $x$ implemented by the MCS.

We use the proof technique of [19, 44, 46] to prove:

**Theorem 4.2** For any implementation of PCG, $|\text{write}| + |\text{write}| + |\text{read}| \geq d$.

**Proof:** Assume, by way of contradiction, that there exists an implementation of PCG for which $|\text{write}| + |\text{write}| + |\text{read}| < d$. Let $p_1$ and $p_2$ be two processes that access $x$, $y$ and $z$; assume that $x$, $y$ and $z$ are initially 0.

By the specification of $x$, $y$ and $z$, there exists a PCG execution $\sigma_1$ such that $\text{ops}(\sigma_1)$ is

$[\text{Write}_1(x, 1), \text{Ack}_1(x)], [\text{Write}_1(z, 1), \text{Ack}_1(z)], [\text{Read}_1(y), \text{Return}_1(y, 0)]$,
Write$_1(x, 1)$ occurs at real time 0, Write$_1(z, 1)$ occurs immediately after Ack$_1(x)$ and Read$_1(y)$ occurs immediately after Ack$_1(z)$. 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 $x$, $y$ and $z$, there exists a PCG execution $\sigma_2$ such that $\text{ops}(\sigma_2)$ is

$$[\text{Write}_2(y, 1), \text{Ack}_2(y)], [\text{Write}_2(z, 2), \text{Ack}_2(z)], [\text{Read}_2(x), \text{Return}_2(x, 0)]$$

Write$_2(y, 1)$ occurs at real time 0, Write$_2(z, 2)$ occurs immediately after Ack$_2(y)$ and Read$_2(x)$ occurs immediately after Ack$_2(z)$. The delay of all messages in $\sigma_2$ is exactly $d$. By assumption, the real time at the end of $\sigma_2$ is less than $d$. Hence, no message is received at any node during $\sigma_2$.

Since no message is ever received in $\sigma_1$ and $\sigma_2$, the partial execution obtained from $\sigma_1$ by replacing $p_1$'s history with $p_2$'s history in $\sigma_2$ can be extended to a PCG execution $\sigma$. Then $\text{ops}(\sigma)$ consists of the operations

$$[\text{Write}_1(x, 1), \text{Ack}_1(x)], [\text{Write}_1(z, 1), \text{Ack}_1(z)], [\text{Read}_1(y), \text{Return}_1(y, 0)]$$

by $p_1$ and

$$[\text{Write}_2(y, 1), \text{Ack}_2(y)], [\text{Write}_2(z, 2), \text{Ack}_2(z)], [\text{Read}_2(x), \text{Return}_2(x, 0)]$$

by $p_2$. Since $\sigma$ is PCG, for every $j \in \{1, 2\}$, there exists a legal serialization $\tau_j$ of $\sigma|_{\{j+\text{writes}\}}$ such that the following hold:

1. $\tau_j|j = \sigma|j$.
2. For every two write operations $w^1(w, v)$ and $w^2(w, u)$, $w^1(w, v) \xrightarrow{\tau_j} w^2(w, u)$ if and only if $w^1(w, v) \xrightarrow{\gamma} w^2(w, u)$ for every object $w$ and values $v$ and $u$.

Hence, either [Write$_1(z, 1)$, Ack$_1(z)$] precedes [Write$_2(z, 2)$, Ack$_2(z)$] in both $\tau_1$ and $\tau_2$ or [Write$_2(z, 2)$, Ack$_2(z)$] precedes [Write$_1(z, 1)$, Ack$_1(z)$] in both $\tau_1$ and $\tau_2$.

If [Write$_1(z, 1)$, Ack$_1(z)$] precedes [Write$_2(z, 2)$, Ack$_2(z)$], then consider $\tau_3$. Since $\tau_3$ is legal, each read should precede the write to the same object (by the other process) in $\tau_3$. 

36
Thus, in \( \tau_3 \), \([\text{Read}_2(x), \text{Return}_2(x, 0)]\) precedes \([\text{Write}_1(x, 1), \text{Ack}_1(x)]\), which, in turn, precedes \([\text{Write}_1(z, 1), \text{Ack}_1(y)]\). However, by definition, \([\text{Write}_2(z, 2), \text{Ack}_2(z)]\) precedes \([\text{Read}_2(x), \text{Return}_2(x, 0)]\). Hence, \([\text{Write}_2(z, 2), \text{Ack}_2(z)]\) precedes \([\text{Write}_1(z, 1), \text{Ack}_1(z)]\). A contradiction to the assumption that \([\text{Write}_1(z, 1), \text{Ack}_1(z)]\) precedes \([\text{Write}_2(z, 2), \text{Ack}_2(z)]\).

If \([\text{Write}_2(z, 2), \text{Ack}_2(z)]\) precedes \([\text{Write}_1(z, 1), \text{Ack}_1(z)]\), then we can reach a contradiction for \(\tau_1\) using the symmetric arguments. ■

We now discuss PCD. The definition of PCD requires the notion of semi-causality, which is slightly weaker than the notion of causality discussed above. However, before we can define semi-causality, we must introduce three other partial orders. These relations are defined as follows: Given an execution \(\sigma\), for every object \(x\) we assume a legal serialization \(\tau_x\) of \(\text{ops}(\sigma)|x\). For every two operations \(op^1_j\) and \(op^2_k\), we define:

- **Weak program order:** \(op^1_j \xrightarrow{wp_0} op^2_k\) if \(k = j\), \(op^1_j \xrightarrow{w} op^2_k\) and either \(op^1_j\) and \(op^2_k\) access the same object, \(op^1_j\) and \(op^2_k\) are both reads or both writes or \(op^1_j\) is a read and \(op^2_k\) is a write. Unlike the definition of program order, it does not require to order two operations by the same process if the first one is a write and the second one is a read to a different object.

- **Weak writes before:** \(op^1_j \xrightarrow{ww^b} op^2_k\) if \(op^1_j = r^1_j(x, v)\), \(op^2_k = r^2_k(y, u)\) and there exists another operation \(w^3_k(y, u)\) such that \(op^1_j \xrightarrow{wp_0} w^3_k(y, u)\). Note that this is a weaker notion than the usual definition of writes before, since it does not require to order \(w^3_k(y, u)\) before \(r^1_k(y, u)\).

- **Weak reads before:** \(op^1_j \xrightarrow{wr^b} op^2_k\) if \(op^1_j = r^1_j(x, v)\), \(op^2_k = w^2_k(y, u)\) and there exists another operation \(w^3_k(x, v')\) such that \(op^1_j \xrightarrow{w} w^3_k(x, v')\) and \(w^3_k(x, v') \xrightarrow{wp_0} op^2_k\).

The semi-causality relation, denoted by \(\sim\), is defined as be the transitive closure of \(\xrightarrow{wp_0}, \xrightarrow{ww^b}\) and \(\xrightarrow{wr^b}\).

Intuitively, PCD requires that each process will view its own operations in the order they were invoked and that all processes agree on the order of writes to the same location and the order of operations that are ordered by the semi-causality relation. Formally,

**Definition 4.6 (PCD)** An execution \(\sigma\) is PCD if the following hold:
1. for every object \( x \) there exists a serialization \( \tau_x \) of \( \text{ops}(\sigma)\|x \) such that for every process \( p_j \), \( \tau_x\|j = \sigma\|j \).

2. for every process \( p_j \) there exists a legal serialization \( \tau_j \) of \( \sigma\|(j + \text{writes}) \) such that
   - For every two operations \( \text{op}_1^j \) and \( \text{op}_2^j \) in \( \tau_j \), if \( \text{op}_1^j \sim \text{op}_2^j \) then \( \text{op}_1^j \xrightarrow{\tau_j} \text{op}_2^j \).
   - For every object \( x \), \( \tau_j\|x = \tau_x\|(j + \text{writes}) \).

Theorem 4.3 For any implementation of PCD, \( |\text{write}| + |\text{write}| + |\text{read}| + |\text{read}| \geq d \).

Proof: Assume, by way of contradiction, that there exists an implementation of PCD for which \( |\text{write}| + |\text{write}| + |\text{read}| + |\text{read}| < d \). Let \( p_1 \) and \( p_2 \) be two processes that access \( x, y \) and \( z \); assume that \( x, y \) and \( z \) are initially \( 0 \).

By the specification of \( x, y \) and \( z \), there exists a PCD execution \( \sigma_1 \) such that \( \text{ops}(\sigma_1) \) is

\[
\begin{align*}
\text{Write}_1(x,1), \text{Ack}_1(x), \text{Write}_1(z,1), \text{Ack}_1(z), \text{Read}_1(z), \text{Return}_1(z,1), \\
\text{Read}_1(y), \text{Return}_1(y,0)
\end{align*}
\]

\( \text{Write}_1(x,1) \) occurs at real time \( 0 \), \( \text{Write}_1(z,1) \) occurs immediately after \( \text{Ack}_1(x) \), \( \text{Read}_1(z) \) occurs immediately after \( \text{Ack}_1(z) \) and \( \text{Read}_1(y) \) occurs immediately after \( \text{Return}_1(z) \). 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 \( x, y \) and \( z \), there exists a PCD execution \( \sigma_2 \) such that \( \text{ops}(\sigma_2) \) is

\[
\begin{align*}
\text{Write}_2(y,1), \text{Ack}_2(y), \text{Write}_2(z,2), \text{Ack}_2(z), \text{Read}_2(z), \text{Return}_2(z,2), \\
\text{Read}_2(x), \text{Return}_2(x,0).
\end{align*}
\]

\( \text{Write}_2(y,1) \) occurs at real time \( 0 \), \( \text{Write}_2(z,2) \) occurs immediately after \( \text{Ack}_2(y) \), \( \text{Read}_2(z) \) occurs immediately after \( \text{Ack}_2(z) \) and \( \text{Read}_2(x) \) occurs immediately after \( \text{Return}_2(z) \). 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 a PCD execution \( \sigma \) (Figure 4.3).
Since $\sigma$ is PCD, for every $j \in \{1, 2\}$, there exists a legal sequence $\tau_j$ that obeys the requirements in the definition of PCD. Hence, either $w_1(z, 1) \xrightarrow{\tau_j} r_1(z, 1) \xrightarrow{\tau_j} w_2(z, 2) \xrightarrow{\tau_j} r_2(z, 2)$ or $w_2(z, 2) \xrightarrow{\tau_j} r_2(z, 2) \xrightarrow{\tau_j} w_1(z, 1) \xrightarrow{\tau_j} r_1(z, 1)$.

Note that PCD requires that for every $j \in \{1, 2\}$ $w_1(x, 1) \xrightarrow{\tau_j} w_1(z, 1), r_1(z, 1) \xrightarrow{\tau_j} r_1(y, 0), w_2(y, 1) \xrightarrow{\tau_j} w_2(z, 2)$ and $r_2(z, 2) \xrightarrow{\tau_j} r_2(x, 0)$ (this is due to the partial order). Hence, if $w_1(z, 1) \xrightarrow{\tau_j} r_1(z, 1) \xrightarrow{\tau_j} w_2(z, 2) \xrightarrow{\tau_j} r_2(z, 2)$, then we have $w_1(x, 1) \xrightarrow{\tau_j} w_1(z, 1) \xrightarrow{\tau_j} r_1(z, 1) \xrightarrow{\tau_j} w_2(z, 2) \xrightarrow{\tau_j} r_2(z, 2) \xrightarrow{\tau_j} r_2(x, 0)$. A contradiction to the assumption that $\tau_2$ is legal. On the other hand, if $w_2(z, 2) \xrightarrow{\tau_j} r_2(z, 2) \xrightarrow{\tau_j} w_1(z, 1) \xrightarrow{\tau_j} r_1(z, 1)$, then we have $w_2(y, 1) \xrightarrow{\tau_j} w_2(z, 2) \xrightarrow{\tau_j} r_2(z, 2) \xrightarrow{\tau_j} w_1(z, 1) \xrightarrow{\tau_j} r_1(z, 1) \xrightarrow{\tau_j} r_1(y, 0)$. A contradiction to the assumption that $\tau_1$ is legal.
Chapter 5

Hybrid Consistency

In this chapter we present a formal definition of hybrid consistency in the framework developed in Chapter 3, in two flavors. Following this, we present a programming example that demonstrates how to use the formal definition of hybrid consistency in order to develop programs and prove their correctness. Finally, we study the compositionality of hybrid consistency.

5.1 Definition of Hybrid Consistency

In order to define hybrid consistency, we introduce the notion of *strong* and *weak* operations. Given a program, it is possible to mark some operations as strong; all other operations are weak. In the case of read/write objects this means that it is possible to use *strong reads* and *strong writes*. We denote the call events for strong operations by SWrite($x, v$) and SRead($x$) and the respective response events by SAck($x$) and SReturn($x, v$). In order to emphasize that a certain operation is strong, we use the notation $sop$. We use subscripts to denote strong operations of a certain process and superscripts to distinguish between operations of the same process, e.g., $sop^1_i$, $sop^2_i$ etc. In particular, $sr_i(x, v)$ is a strong read operation invoked by process $p_i$ returning $v$ from $x$ and $sw_i(x, v)$ is a strong write operation invoked by process $p_i$ writing $v$ to $x$.

We now turn to the formal definition of hybrid consistency. This definition requires that (a) strong operations appear to occur in the same order at all processes, and (b) if two operations
are invoked by the same process and one of them is strong, then they appear to occur at all processes in the order they were invoked.

**Definition 5.1 (Hybrid consistency based on linearizability)** An execution $\sigma$ is hybrid if there exists a linearization $\rho$ of $\sigma$ such that for each process $p_i$, there exists a legal sequence of operations $\tau_i$ with the following properties:

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

Definition 5.1 defines the view $\tau_i$ for each process $p_i$. This view must include all the memory operations in the execution (this is the first requirement). Any two operations invoked by the same process $p_i$, where at least one of them is strong, must be viewed in their order of invocation in $p_i$ by $p_j$ (this is the second requirement). The views of all processes must agree on the order of the strong operations (this is the third requirement). Finally, process $p_j$ must view its own operations in their order of invocation (this is the fourth requirement).

The definition requires the existence of a linearization $\rho$ of $\sigma$. However, $\rho$ is used only to force a global order on the strong operations. Therefore, we can require the existence of a linearization only of the strong operations in $\sigma$, obtaining an equivalent definition.

When restricted to strong operations, this definition is equivalent to linearizability. However, in certain systems, sequential consistency can be implemented more efficiently than linearizability [19]. Hence, it is interesting to examine hybrid models in which strong operations are only sequentially consistent. This leads to the following variant of hybrid consistency in which $\rho$ is a serialization of $\sigma$ rather than a linearization: (The only difference between Definition 5.1 and Definition 5.2 is in the requirement from $\rho$. We repeat the entire definition for completeness.)

**Definition 5.2 (Hybrid consistency based on sequential consistency)** An execution $\sigma$ is hybrid if there exists a serialization $\rho$ of $\sigma$ such that for each process $p_i$, there exists a legal sequence of operations $\tau_i$ with the following properties:
1. $\tau_i$ is a permutation of $\text{ops}(\sigma)$.

2. if $\text{op}_1^i \xrightarrow{\sigma} \text{op}_2^i$ and at least one of $\text{op}_1^i$ and $\text{op}_2^i$ is strong, then $\text{op}_1^i \xrightarrow{\tau_i} \text{op}_2^i$, for any $i$.

3. if $\text{op}_1^i \xrightarrow{\sigma} \text{op}_2^i$ and $\text{op}_1^i$ and $\text{op}_2^i$ are strong, then $\text{op}_1^i \xrightarrow{\tau_i} \text{op}_2^i$, for any $i$ and $k$.

4. $\tau_i \mid j = \sigma \mid j$.

### 5.1.1 Examples

If we use only strong operations, then hybrid consistency is equivalent to linearizability, according to Definition 5.1, and to sequential consistency, according to Definition 5.2. Similarly, if we use only weak operations, then hybrid consistency is equivalent to weak consistency. However, things start to be more interesting if we mix weak and strong operations. We demonstrate this with the executions of Figure 3.2.

**Execution $\sigma_1$:** This execution is linearizable (Section 3.2.1). Hence, $\sigma_1$ is hybrid consistent according to both definitions (Definitions 5.1 and 5.2), regardless of the labeling of the operations (weak or strong).

**Execution $\sigma_2$:** This execution is weakly consistent (Section 4.1.1). Hence, if we label all operations in $\sigma_2$ as weak, then it is hybrid consistent according to both definitions. On the other hand, this execution is sequentially consistent but not linearizable (Section 3.2.1). Hence, if we label all operations as strong, then this execution is hybrid consistent only according to Definition 5.2.

**Execution $\sigma_3$:** This execution is weakly consistent (Section 4.1.1). Hence, if we label all operations in $\sigma_3$ as weak, then it is hybrid consistent according to both definitions. However, if we label at least one operation in Execution $\sigma_3$ as strong, then it is no longer hybrid consistent. For example, if we label $r_2^3(x, 0)$ as strong, then there does not exist a sequence $\tau_1$ as required in the definition of hybrid consistency. This is because in any legal serialization of $\sigma_3$, the order of either the operations of $p_1$ or the operations of $p_2$ must be changed w.r.t. the order in which they were invoked. However, according to both definitions of hybrid consistency, all operations of $p_1$ must appear in $\tau_1$ in the order they were invoked. Also, since $r_2^3(x, 0)$ is strong,
the two operations of \( p_2 \) must be ordered in \( \tau_1 \) in the order they were invoked and we have a contradiction.

**Execution \( \sigma_4 \):** This execution is weakly consistent (Section 4.1.1). Hence, if we label all operations in \( \sigma_4 \) as weak, then it is hybrid consistent according to both definitions. However, it can be shown that if we label only one operation in Execution \( \sigma_4 \) as strong, then it is no longer hybrid consistent. (The arguments here are the same as for \( \sigma_3 \).)

**Execution \( \sigma_5 \):** This execution is not sequentially consistent (Section 3.2.1). Hence, if we label all operations in \( \sigma_5 \) as strong, then it is not hybrid consistent according to both definitions. On the other hand, if we only label the reads in Execution \( \sigma_5 \) as strong (and the writes as weak), then this execution is hybrid consistent according to both definitions. To show this, we take:

\[
\tau_1 = \begin{array}{c}
w_2^3(x, 2), r_3^1(x, 2), r_3^2(y, 0), w_1^1(y, 1), w_2^3(y, 1), w_1^2(x, 2), \\
\end{array}
\]

\[
\tau_2 = \begin{array}{c}
w_1^2(x, 2), r_2^3(x, 2), r_3^2(y, 0), w_1^1(y, 1), w_1^3(x, 1), w_2^3(x, 2) \quad \text{and} \\
\end{array}
\]

\[
\tau_3 = \begin{array}{c}
w_2^3(x, 2), r_3^1(x, 2), r_3^2(y, 0), w_1^1(y, 1), w_2^1(y, 1), w_1^2(x, 2) \\
\end{array}
\]

(as in Section 4.1.1). Clearly, these sequences obey all requirements in both definitions of hybrid consistency (even if the two reads are labeled as strong).

### 5.2 Programming Example

As mentioned before, 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 [44]. Unfortunately, this method turns out to be “too weak” to solve some of the fundamental problems in concurrent programming, as was shown in Chapter 4.

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

As before, we limit our discussion to logical mutual exclusion. A simpler discussion of physical mutual exclusion with hybrid consistency, using the same programming example, appears in [17]. Hence, in the rest of this section, we mean logical mutual exclusion whenever we refer to mutual exclusion. We remind the reader that $CS^k_i$ represents the set of operations that are invoked during the $k$th time that $p_i$ executes the critical section in $\sigma[j]$. We denote by $CS^k_i \rightarrow CS^l_j$ the fact that all operations of a critical section $CS^k_i$ appear in a sequence of operations $\tau$ before all operations of another critical section $CS^l_j$.

Figure 5.1 presents a solution for two processes which is fair; it is a simple modification of Peterson's algorithm [53]. Lines 1-4 are the entry section; Line 5 is the exit section. In the code we use the notation $r(x) = v$ to denote a weak read of $x$ returning the value $v$ and $sw(x, v)$ to denote a strong write of $v$ to $x$, for any read/write object $x$ and value $v$.

We assume that dummy is a special object that is accessed only by the strong write in Line 4. This strong write is not part of the original algorithm and is added in order to ensure that all operations of the critical section are viewed by every process after all operations of the corresponding entry section.

**Lemma 5.1** The algorithm in Figure 5.1 guarantees mutual exclusion under hybrid consistency.

**Proof:** In order to prove mutual exclusion we have to show that operations that belong to critical sections appear in both $\tau_0$ and $\tau_1$ in a non-overlapping manner and that the order
in which all critical sections are ordered in \( \tau_0 \) is the same as in \( \tau_1 \). Note that there is a strong write immediately before entering the critical section and immediately after leaving the critical section. This guarantees that all operations invoked inside a particular execution of the critical section will be viewed by every process after every operation that is invoked by the same process before entering the critical section. Similarly, all operations invoked inside a particular execution of the critical section will be viewed by every process before every operation by the same process that is invoked after leaving the critical section. Hence, operations that belong to one critical section do not overlap with operations that belong to another critical section by the same process.

Assume, by way of contradiction, that operations belonging to a critical section of \( p_0 \) overlap with operations belonging to a critical section of \( p_1 \) in \( \tau_i \), for some \( i \in \{0, 1\} \). Denote the last strong writes to \( \text{need}_0 \) and \( \text{turn} \) by \( p_0 \) before entering this critical section by \( \text{sw}_0(\text{need}_0, \text{true}) \) and \( \text{sw}_0(\text{turn}, 1) \). Similarly, denote the last strong writes to \( \text{need}_1 \) and \( \text{turn} \) by \( p_1 \) before entering its corresponding critical section by \( \text{sw}_1(\text{need}_1, \text{true}) \) and \( \text{sw}_1(\text{turn}, 0) \). Assume, without loss of generality, that \( \text{sw}_0(\text{turn}, 1) \xrightarrow{\tau_i} \text{sw}_1(\text{turn}, 0) \). However, by the assumption that \( p_1 \) started executing its critical section and the requirements from \( \tau_i \) in the definition of hybrid consistency, either \( \text{sw}_0(\text{turn}, 1) \xrightarrow{\tau_i} \text{sw}_1(\text{turn}, 0) \xrightarrow{\tau_i} \text{r}_1(\text{turn}, 1) \) or \( \text{sw}_0(\text{need}_0, \text{true}) \xrightarrow{\tau_i} \text{r}_1(\text{need}_0, \text{false}) \). Hence, either \( \text{sw}_1(\text{turn}, 0) \) is the last write to \( \text{turn} \) before a read that returns 1 or \( \text{sw}_0(\text{need}_0, \text{true}) \) is the last write to \( \text{need}_0 \) before a read that returns false. A contradiction to the assumption that \( \tau_i \) is legal. Hence, we have shown that operations that belong to critical sections appear in a non-overlapping manner in both \( \tau_0 \) and \( \tau_1 \).

Assume, by way of contradiction, that there exist two critical sections \( \text{CS}_k^0 \) and \( \text{CS}_l^1 \) such that \( \text{CS}_k^0 \xrightarrow{\tau_i} \text{CS}_l^1 \) but \( \text{CS}_l^1 \xrightarrow{\tau_i} \text{CS}_k^0 \), for some \( i, \tau \in \{0, 1\} \). If there are several such pairs, then assume that \( \text{CS}_k^0 \) and \( \text{CS}_l^1 \) are the first one of them in \( \tau_i \). (Note that this is well defined.) Since we already showed that all critical sections are ordered in a non-overlapping manner, we have:

\[
\tau_i : \ldots, \text{CS}_k^0, \ldots, \text{CS}_l^1, \ldots
\]

and

\[
\tau_i : \ldots, \text{CS}_l^1, \ldots, \text{CS}_k^0, \ldots
\]

That is, the critical sections are ordered in the same order in \( \tau_i \) and \( \tau_i \) up to \( \text{CS}_k^0 \) and \( \text{CS}_l^1 \); \( \text{CS}_l^1 \) is the first critical section by \( p_1 \) that is ordered in \( \tau_i \) after \( \text{CS}_k^0 \) and \( \text{CS}_0^k \) is the first critical section by \( p_0 \) that is ordered in \( \tau_i \) after \( \text{CS}_l^1 \). Denote the last read from \( \text{turn} \) by \( p_0 \).
before entering $CS_k^0$ by $r_0(t_{\text{turn}}, u_0)$ and the last read from $t_{\text{turn}}$ by $p_1$ before entering $CS_1^1$ by $r_1(t_{\text{turn}}, u_1)$. Since $CS_k^0$ and $CS_1^1$ are the first pair of critical sections that appear in a different order in $\tau_i$ and $\tau_1$, and since $CS_k^0 \xrightarrow{\tau_i} CS_1^1$, $u_1 = 1$. Similarly, by the minimality of $CS_k^0$ and $CS_1^1$ and since $CS_1^1 \xrightarrow{\tau_1} CS_k^0$, $u_0 = 0$. Hence, either the last critical section executed in $\tau_i$ is by $p_i$ or the initial value of $t_{\text{turn}}$ is 0 and $CS_k^0$ is the first critical section in $\tau_i$. Therefore, by the minimality of $CS_k^0$ and $CS_1^1$, either the last critical section executed in $\tau_i$ is by $p_i$ or the initial value of $t_{\text{turn}}$ is 0 and $CS_1^1$ is the first critical section in $\tau_i$. Since we assumed that $\tau_i$ is legal, $u_1 = 0$. A contradiction. Hence, we have shown that the order in which all critical sections are ordered in $\tau_0$ is the same as in $\tau_1$, which completes the proof. 

**Lemma 5.2** The algorithm in Figure 5.1 is free of deadlocks under hybrid consistency.

**Proof:** Assume, by way of contradiction, that there exists an infinite execution $\sigma$ that has a deadlock and consider the sequences of operations $\tau_0$ and $\tau_1$ as guaranteed by the definition of hybrid consistency. Assume, without loss of generality, that $p_0$ executes the entry section from some point on in $\tau_0$, for some $i \in \{0, 1\}$, but $p_1$ executes its critical section only a finite number of times in $\tau_i$. 

Denote by $sw_1(\text{need}[1], \text{false})$ the strong write in the last exit section executed by $p_1$ in $\tau_i$. Since we assumed that $p_0$ executes the entry section from some point on in $\tau_0$, $\tau_0$ includes an infinite number of operations of the form $r_0^k(\text{need}[1], u)$. Thus, there exists a read $r_0^k(\text{need}[1], u_0)$ such that $sw_1(\text{need}[1], \text{false}) \xrightarrow{\tau_i} r_0^k(\text{need}[1], u_0)$. Since $\tau_i$ is legal and since $sw_1(\text{need}[1], \text{false})$ is the last strong write to $\text{need}[1]$ in $\tau_i$, $u_0 = \text{false}$. Hence, $p_0$ should have entered its critical section following $r_0^k(\text{need}[1], u_0)$ and the corresponding $r_0(t_{\text{turn}}, v)$. A contradiction. 

An additional property we would like to provide is:

**Starvation freedom:** If $\tau_i$ is infinite and there exists a process $p_i$ which executes the entry section at some point in $\tau_i$, then $p_i$ executes the corresponding critical section at some later point in $\tau_i$, for every process $p_j$.

**Lemma 5.3** The algorithm in Figure 5.1 is starvation free under hybrid consistency.

**Proof:** Assume, by way of contradiction, that there exists an infinite execution $\sigma$ that has starvation. Assume, without loss of generality, that $p_0$ is executing the entry section from some
point on in $\tau_j$, for some $j \in \{0, 1\}$, but never executes the corresponding critical section. By Lemma 5.2, $p_i$ enters and leaves its critical section infinitely often in $\tau_j$. Thus, there exists a strong write $sw_1^j(\text{turn}, 0)$ such that $sw_0\text{(need[0], true)} \xrightarrow{\tau_j} sw_1\text{(turn, 1)} \xrightarrow{\tau_j} sw_2^j(\text{turn}, 0)$. However, since we assumed that $p_1$ enters its critical section, either the following read from $\text{turn}$ by $p_1$ returns 1, or the following read from $\text{need[0]}$ returns false. Thus, either $sw_1(\text{turn}, 0)$ is the last strong write to $\text{turn}$ that appears in $\tau_j$ before a read from $\text{turn}$ that returns 1, or $sw_0(\text{need[0]}, \text{true})$ is the last strong write to $\text{need[0]}$ that appears in $\tau_j$ before a read from $\text{need[0]}$ that returns false. A contradiction to $\tau_j$ being legal.

The algorithm uses only weak reads in the entry section; hence, when running on an efficient implementation of hybrid consistency (like the ones presented in Chapter 6), the entry section is executed faster and with less message overhead.\footnote{Note that by efficient implementations we refer to implementations that execute weak operations faster than the network delay. By the lower bounds presented in [19] and in Chapter 7, strong operations cannot be implemented faster than the network delay.}

### 5.3 Compositionality of Hybrid Consistency

Compositionality is a property of consistency conditions defined by Herlihy and Wing ([38]), and called there locality. If $C$ is a compositional consistency condition then a shared memory supporting $C$ can be composed of independent implementations of each object (or group of objects), each supporting $C$. Hence, compositionality allows easy combination of various implementations, with different optimizations for different objects. This gives a flexibility which is especially important for systems in which different objects exhibit different access patterns, as demonstrated in [46]. More formally:

**Definition 5.3** A correctness condition $C$ is compositional if for every execution $\sigma$, $\sigma$ satisfies $C$ if and only if $\sigma | X$ satisfies $C$ for every object $X$.

It is shown in [38] that linearizability is compositional, while sequential consistency is not compositional. We show:

**Theorem 5.4** Hybrid consistency is not compositional.
Proof: Consider the execution $\sigma$ such that

$$\sigma = r_1(x, 1), sr_1(y, 0), sw_2(y, 1), w_2(x, 1)$$

where both $x$ and $y$ are initialized to 0. Note that

$$\sigma|_{x} = r_1(x, 1), w_2(x, 1) \quad \text{and} \quad \sigma|_{y} = sr_1(y, 0), sw_2(y, 1).$$

Clearly, $\sigma$ is not a hybrid execution, while both $\sigma|_{x}$ and $\sigma|_{y}$ are hybrid executions. $\blacksquare$

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

**Theorem 5.5** *Weak consistency is compositional.*

Proof: We need to show that $\sigma$ is a weakly consistent execution, if and only if $\sigma|_{x}$ 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 $\tau_j|_{j} = \sigma|_{j}$. Since $\tau_j$ is legal, for each object $x$, $\tau_j|_{x}$ is in the specification of $x$. The execution $\tau_j|_{x}$ is a permutation of $\sigma|_{x}$ and $(\tau_j|_{x})|_{j} = (\sigma|_{x})|_{j}$.

The execution $\sigma|_{x}$ is weakly consistent if, for every process $p_j$, there exists a legal serialization $\tau_j^x$ of $\sigma|_{x}$ such that $\tau_j^x|_{j} = (\sigma|_{x})|_{j}$. Thus, any merging (i.e., any interleaving of the operations of all the sequences that obeys the per sequence orderings) of the sequences $\tau_j^x$, for all objects $x$, forms a legal sequence $\tau_j$ for process $p_j$. The execution $\tau_j$ is a serialization of $\sigma$ and $\tau_j|_{j} = \sigma|_{j}$. $\blacksquare$

Thus, hybrid consistency (based on linearizability) is not compositional although it is composed of two types of operations, weak and strong, each of which is compositional. This implies that compositionality is not a “monotonic” property.
Chapter 6

Implementing Hybrid Consistency

In this chapter we present two asynchronous algorithms for implementing hybrid consistency in a message based system. Both algorithms share three basic properties:

- every process holds a complete copy of the entire memory,
- messages are sent using an atomic broadcast mechanism, and
- week operations are executed immediately while the execution time for strong operations is $O(d)$, which is optimal (up to a constant multiplicative factor).

These algorithms allow weak operations to be executed instantaneously, while keeping the response time for strong operations linearly proportional to the network delay. Being able to execute weak operations fast is vital for the usefulness of these algorithms, since it is the major advantage hybrid consistency has over sequential consistency. By allowing each process to hold a local copy of the entire memory, we can guarantee instantaneous execution of weak operations, provided that write operations will eventually update all copies. (In Chapter 11, we discuss how to extend our algorithms for the case where processes do not hold a local copy of the entire memory.)

The algorithms use an atomic broadcast mechanism to send messages. Having an atomic broadcast as an underlying tool in the development of distributed systems is becoming quite

\[\text{That is, processes run at different rates and the delay of messages is unknown.}\]
common [20, 21, 45, 55]. Our decision to use atomic broadcast follows this trend and is justified by the immediate benefits that this mechanism provides. Assuming an atomic broadcast mechanism is more modular than implementing the communication protocol directly. Moreover, the atomic broadcast mechanism creates a *global logical time*, which simplifies the code of the algorithms and their proof of correctness. Several atomic broadcast algorithms with various degrees of fault tolerance and efficiency have been developed [12, 22, 23, 24]. Thus, the exact choice of an atomic broadcast algorithm can help reach the design goals of the implementation in terms of fault tolerance, message complexity and time complexity.

Our algorithms differ one from the other mainly by the types of operations they support and whether weak and strong operations are allowed to access the same objects or not. These assumptions affect the specific code of each of the algorithms and its proof of correctness.

The first algorithm is a generic implementation that supports weak read and weak write operations and any type of strong operations that access responsive objects (Definition 3.1). Recall that this includes, for example, read, write, read-modify-write, enqueue, dequeue, push and pop. This algorithm assumes that the set of objects accessed by weak operations is disjoint from the set of objects accessed by strong operations. However, when restricted to objects that support read, write, read-modify-write, enqueue, dequeue, push and pop, weak reads can access the same objects as strong operations.

In the second algorithm, every object can be accessed by every operation. However, the only types of strong operations supported by this algorithm are read and write. Also, the code for this algorithm is much more complicated than the code of the first algorithm.

The first algorithm is presented in Section 6.1 while the second algorithm is presented in Section 6.2. In Section 6.3, we discuss several efficient algorithms for implementing atomic broadcast and present one such algorithm.

### 6.1 A Generic Implementation for Hybrid Consistency

In this section, we present a generic algorithm for providing hybrid consistency that supports weak read/write operations and any type of strong operations that access responsive objects. This algorithm assumes that the set of objects accessed by weak operations is disjoint from the set of objects accessed by strong operations. Later, we show that the algorithm is still correct
even if weak read operations and strong operations are allowed to access the same objects, when restricted to read, write and read-modify-write operations and operations on queues and stacks.

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

The main problem in the design of the algorithm is that weak operations must return immediately and a weak read that immediately follows a weak write by the same process must return the value written by that write (the “newest” value). Note that due to the atomic broadcast mechanism, given two weak writes \( w_i(x, v) \) and \( w_j(x, u) \), if the update message of \( w_j(x, u) \) is delivered between the invocation of \( w_i(x, v) \) and the delivery of its update message, then \( w_i(x, v) \) is “newer” than \( w_j(y, u) \). Hence, the problem described above is solved using the following two rules: (a) a weak write updates the local copy of the memory immediately following its invocation, and (b) a process that invokes a weak write ignores every message that is delivered between the invocation of the weak write and the delivery of its corresponding message and contains a request to update the same object. This guarantees that each weak read returns the value of the “newest” weak write (according to the logical time) to the same object that is known to its invoking process when the read is invoked.

A more detailed description of the algorithm and the pseudocode can be found in Section 6.1.1 followed by the proof of correctness for the generic case in Section 6.1.2. In Section 6.1.3 we show that the algorithm is correct even if we allow weak read operations and strong operations to access the same objects, when restricted to objects that support read, write and read-modify-write operations. Finally, we analyze the complexity of the algorithm in Section 6.1.4.

### 6.1.1 The Algorithm

We assume a system of \( n \) processes, connected by some interconnection network, each holds a local copy of the entire memory. We assume an atomic broadcast mechanism capable of supporting two primitives: \( \texttt{abc} \) and \( \texttt{send.abc} \). The first primitive, \( \texttt{abc} \), accepts a message as
a parameter, and broadcasts this message to all processes atomically. The second primitive, \texttt{send\_abc}, accepts two parameters: a message to send and a process id to whom the message is designated. A process may use \texttt{send\_abc} to send a message to a single process when the interleaving of this message with other messages sent by it, using either \texttt{abc} or \texttt{send\_abc}, should reflect their relative order of invocation.

More precisely, denote by $\text{deliver}(j)$ the sequence of all messages delivered at $p_j$ (either from \texttt{abc} or \texttt{send\_abc}), in the order they were delivered at $p_j$; denote by $\text{abc\_deliver}(j)$ the restriction of $\text{deliver}(j)$ to messages sent with \texttt{abc}. For every pair of processes $p_i$ and $p_j$, denote by $\text{sent}(i, j)$ the sequence of messages that are sent from $p_i$ to $p_j$ (either with \texttt{abc} or \texttt{send\_abc}), ordered in the order they were sent by $p_i$; denote by $\text{deliver}(j)|i$ the restriction of $\text{deliver}(j)$ to messages sent from $p_i$ to $p_j$. We require that

1. $\text{sent}(i, j) = \text{deliver}(j)|i$, and
2. $\text{abc\_deliver}(i) = \text{abc\_deliver}(j)$.

We use \texttt{send\_abc} instead of a regular send since it guarantees that the order of delivery of all messages at each process reflects the order in which they were sent. We use \texttt{send\_abc} instead of \texttt{abc} because in some implementations of atomic broadcast, e.g., the one presented later in the chapter, it has a lower message complexity.

Every process holds a local copy of the entire memory \texttt{mem}, an array of integers \texttt{local\_pending}, initially 0, an integer counter \texttt{missing\_acks} that counts the number of expected (missing) acknowledges, initially 0, and an additional strong object \texttt{val} that is used to temporarily store a value that a pending strong operation should return, as described below.

Weak operations are executed on the local copy of the memory and return immediately. A weak write increments the corresponding entry of \texttt{local\_pending} and causes an \texttt{update} message to be broadcast to all processes. This message contains the object to be updated and the new value for the object. Upon receiving an \texttt{update} message, the invoking process decrements the corresponding entry of \texttt{local\_pending}. Other processes execute the write on the local copy of the memory after receiving its \texttt{update} message if they have no pending weak writes to the same object, i.e., if the value of the corresponding entry of \texttt{local\_pending} is 0. This check guarantees that the local copy of every object always holds the “newest” value, w.r.t. the global logical

52
time, known to the local process. Hence, all writes to the same object are *logically* executed by all processes in the same order (the order in which their update messages are delivered).

A strong operation broadcasts a **strong-op** message to every process. The operation is executed by every process whenever the appropriate message is delivered. A strong operation that has write semantics is executed by updating the local copy of the object. A strong operation that has read semantics is executed at the invoking process by storing the value it should return in `val`, and by doing nothing otherwise. (Note, that some operations, e.g., read-modify-write and dequeue, have both read and write semantics. On the other hand, other operations, e.g., DEC-alpha’s MB operations, might not have a read or a write semantics at all; in this case, the operation is executed by doing nothing.) After executing a strong operation, an **ack** message is sent back to the initiator of the operation. A strong operation does not return until all **ack** messages have been received. Note that we can always execute a strong operation when its appropriate message is delivered since we assume that strong operations only access responsive objects. (Note that this is not always possible with objects that are not responsive. For example, a process may not be able to execute an **await** statement until another process issues an appropriate write.)

The code for process $p_i$ appears in Figure 6.1.

### 6.1.2 Proof of Correctness

We prove the correctness of the algorithm by showing that for every execution generated by the algorithm, we can build a set of sequences of operations $\{\tau_i\}_{i=1}^n$ as required by the definition of hybrid consistency.

We start with several definitions and some notation that will be used later.

A read operation $r_i(x, v)$ **reads from a write operation** $w_k(x, v)$ if $w_k(x, v)$ is the last write to $x$ that updates $p_i$'s copy of $x$ before $r_i(x, v)$ reads $x$. We say that a read $r_i(x, v)$ is **legal** in a sequence of operations $\tau$ if either there exists a write $w_k(x, v)$ such that $w_k(x, v) \xrightarrow{\tau} r_i(x, v)$ and there does not exist another write $w_l(x, u)$, $u \neq v$, such that $w_k(x, v) \xrightarrow{\tau} w_l(x, u) \xrightarrow{\tau} r_i(x, v)$, or $v$ is the initial value of the object and there does not exist another write $w_l(x, u)$ such that $w_l(x, u) \xrightarrow{\tau} r_i(x, v)$; otherwise, the read is **illegal** in $\tau$.

For the rest of the proof, fix some execution $\sigma$ of the algorithm.
Read($x$):
    generate Return($x, mem[x]$) /* This is the response event */

Write($x, v$):
    $mem[x] := v$
    $local_{pending}[x] := local_{pending}[x] + 1$
    $abc < update, x, v >$
    generate Ack($x$) /* This is the response event */

Strong-Op(type, var_list):
    $abc < strong-op, type, variables_list >$
    $missing_acks := missing_acks + n$

received $< update, x, v >$ from $p_j$:
    if ($j = i$) then
        $local_{pending}[x] = local_{pending}[x] - 1$
    else if ($local_{pending}[x] = 0$) then
        $mem[x] := v$
    endif

received $< strong-op, t, v >$ from $p_j$:
    $send_{abc} < ack, j >$
    update all objects that needs to be updated according to $t$'s
    semantics and the variables list $v$
    if $t$ was invoked by $p_i$ and has to return a value, then
        store this value in $val$
    endif

received $< ack, k >$ from $p_j$:
    $missing_acks := missing_acks - 1$
    if ($missing_acks = 0$) then
        if the pending strong operation must return a value then
            generate SReturn($t, val$) /* This is the response event */
        else
            generate SAck($t$) /* This is the response event */
        endif
    endif

Figure 6.1: The generic algorithm — code for process $p_i$. 
A partial execution interval of a weak write $w_k$ in $\sigma$ is the interval of events in $p_k$’s history between the invocation of $w_k$ and the delivery of the update message of $w_k$ by the atomic broadcast at $p_k$.

A weak write $w_k(x,v)$ influences process $p_j$ if it updates $p_j$’s copy of $x$.

Next, define the notion of an operation being executed by a process as follows: A strong operation $sop_k$ is executed by $p_j$ when the corresponding strong-op message is delivered at $p_j$. A weak operation $op_j$ is executed by $p_j$ when the appropriate call event occurs. A weak write $w_k$ that influences $p_j$, $k \neq j$, is executed by $p_j$ when the corresponding update message is delivered at $p_j$. Note that this means that a strong operation is always executed by $p_j$.

We now turn to the details of the proof.

For each process $p_j$, let $\tau_j'$ be the sequence of all operations in $\sigma$ that are executed by $p_j$ (all strong operations, all operations by $p_j$ and some of the weak write operations), in the order they are executed in $p_j$. Note that if a weak write $w_k$ is not executed by $p_j$, then by the code of the algorithm, the update message of $w_k$ is delivered inside a partial execution interval of another weak write by $p_j$ to the same object. For each $\tau_j'$, add every weak write $w_k$ that is not executed by $p_j$ immediately before the weak write by $p_j$ to the same object such that the update message of $w_k$ is delivered inside the partial execution interval of $w_j$. (If the update message of $w_k$ is delivered inside the potential execution intervals of more than one weak write to the same object, then let $w_j$ be the first one of them.) Finally, for every weak read $r_k$, $k \neq j$, insert $r_k$ in the first place that is after the write it reads from (if exists) and the previous strong operation by $p_k$ (if exists), but not after the next strong operation by $p_k$ (if exists). (If there exists a next strong operation $sop_k$ and the write $r_k$ reads from is ordered after $sop_k$, then $r_k$ is ordered immediately before $sop_k$.) Denote the resulting sequence by $\tau_j$.

**Lemma 6.1** There exists a linearization $\rho$ of all strong operations in $\sigma$ such that for every pair of strong operations $sop_k^1$ and $sop_k^2$, $sop_k^1 \xrightarrow{\rho} sop_k^2$ if and only if $sop_k^1 \xrightarrow{\tau_j} sop_k^2$, for every $\tau_j$.

**Proof:** Recall that atomic broadcast delivers all messages to all processes in the same order. Since the strong operations are ordered in each $\tau_j$ according to their order of delivery, they appear in every $\tau_j$ in the same order. Since a strong operation does not return before all its ack messages return, the delivery of its strong-op message is always between its invocation...
Lemma 6.2 Every sequence $\tau_j$ is legal.

Proof: By Lemma 6.1, the strong operations are ordered according to the order in which they are executed by every process. Thus, for every strong object $x$, $\tau_j|x$ is in the serial specification of $x$. By the construction of $\tau_j$, every weak read by $p_j$ is ordered in $\tau_j$ after the write it reads from, with no other write to the same object in between them. Thus, it is left to be shown that weak reads by other processes are legal.

By construction of $\tau_j$, a weak read $r_k(x, v), k \neq j$, may not be legal only if one of the following holds:

- there exists a previous strong operation by $p_k$ and the write $r_k(x, v)$ reads from is ordered before the previous strong operation by $p_k$, or
- there exists a next strong operation by $p_k$ and the write $r_k(x, v)$ reads from is ordered after the next strong operation by $p_k$.

Assume, by way of contradiction, that there exists an illegal weak read $r_k(x, v)$, and assume that $r_k(x, v)$ reads from some weak write $w_l(x, v)$. If there exists a next strong operation $sop_k^2$ and $w_l(x, v)$ is ordered in $\tau_j$ after $sop_k^2$, then the update message of $w_l(x, v)$ is delivered after the execution of $r_k(x, v)$ in $p_k$. A contradiction to the assumption that $r_k(x, v)$ reads from $w_l(x, v)$.

Hence, we may assume that there exists a previous strong operation $sop_k^1$ and that there exists another weak write $w_i(x, u), u \neq v$, such that $w_l(x, v) \xrightarrow{\tau_j} w_i(x, u) \xrightarrow{\tau_j} sop_k^1 \xrightarrow{\tau_j} r_k(x, v)$. We complete the proof by examining the following four possible cases.

1. (Figure 6.2(a)) If both $w_l(x, v)$ and $w_i(x, u)$ are executed by $p_j$, then the update messages of $w_l(x, v)$ and $w_i(x, u)$ are delivered in $p_k$ in the order they appear in $\tau_j$, and before the strong-op message of $sop_k^1$. Thus, either $w_i(x, u)$ or some other write to $x$ is executed by $p_k$ between the delivery of the update message of $w_l(x, v)$ and the invocation of $r_k(x, v)$. A contradiction to the assumption that $r_k(x, v)$ reads from $w_l(x, v)$.

and termination. Thus, the order in which all strong operations appear in every $\tau_j$ extends the partial order implied by $\sigma$ and is therefore a linearization. 

2. (Figure 6.2(b).) If only \( w_l(x, v) \) is executed by \( p_j \), then since \( w_l(x, u) \) is ordered before \( sop_k^1 \), there exists another write \( w_j(x, w) \) that is invoked between the delivery of the update message of \( w_l(x, v) \) and the strong-op message of \( sop_k^1 \). Thus, either \( w_j(x, w) \) or some other write to \( x \) is executed by \( p_k \) between the delivery of the update message of \( w_l(x, v) \) and the invocation of \( r_k(x, v) \). A contradiction to the assumption that \( r_k(x, v) \) reads from \( w_l(x, v) \).

3. (Figure 6.2(c).) If only \( w_l(x, u) \) is executed by \( p_j \), then there exists a write \( w_j(x, w) \) whose update message is delivered at \( p_j \) between the delivery of the update message of \( w_l(x, v) \) and the delivery of the update message of \( w_l(x, u) \). (It is possible that \( w_j(x, w) = w_l(x, u) \).) Hence, the update message of \( w_j(x, w) \) is delivered in \( p_k \) between the delivery of the update message of \( w_l(x, v) \) and the delivery of the ack message for \( sop_k^1 \). Thus, either \( w_j(x, w) \) or some other write to \( x \) is executed by \( p_k \) between the delivery of the update message of \( w_l(x, v) \) and the invocation of \( r_k(x, v) \). A contradiction to the assumption that \( r_k(x, v) \) reads from \( w_l(x, v) \).

4. (Figure 6.2(d).) If neither \( w_l(x, v) \) nor \( w_l(x, u) \) are executed by \( p_j \), then there exists a write \( w_j(x, w) \) whose update message is delivered in \( p_j \) after the update messages of \( w_l(x, v) \) and \( w_l(x, u) \), and is invoked before the delivery of the strong-op of \( sop_k^1 \). Hence, the update message of \( w_j(x, w) \) is delivered in \( p_k \) between the delivery of the update messages of \( w_l(x, v) \) and \( w_l(x, u) \) and the ack message of \( sop_k^1 \). Thus, either \( w_j(x, w) \) or some other write to \( x \) is executed by \( p_k \) between the delivery of the update message of \( w_l(x, v) \) and the invocation of \( r_k(x, v) \). A contradiction to the assumption that \( r_k(x, v) \) reads from \( w_l(x, v) \).

\[\Box\]

Lemma 6.3 For every sequence \( \tau_j, \tau_j|j = \sigma|j \).

Proof: Atomic broadcast delivers all messages sent by the same process in the order they are invoked. Also, strong operations do not return until all \texttt{ack} messages are delivered. Thus, since weak operations are ordered according to their invocation and strong operations are ordered according to the delivery of their corresponding messages, for every sequence \( \tau_j, \tau_j|j = \sigma|j \). \[\Box\]
Figure 6.2: Proving the legality of $\tau_j$ in the generic algorithm
Lemma 6.4 For each sequence of operations $\tau_j$ and every pair of operations $op_k^1$ and $op_k^2$ such that either $op_k^1$ or $op_k^2$ is strong, if $op_k^1 \xrightarrow{e} op_k^2$ then $op_k^1 \xrightarrow{\tau_j} op_k^2$.

Proof: By Lemma 6.3, the claim holds if $k = j$. Thus, for the rest of the proof, we may assume that $k \neq j$. Recall that weak reads are placed between the previous and next strong operations of their invoking process as part of the construction of $\tau_j$. Weak writes that influence $p_j$ are ordered according to the order their update messages are delivered at $p_j$ and are therefore ordered between the previous and next strong operations by the same process. A weak write that does not influence $p_j$ is ordered before any next strong operation by the same process as part of the construction of $\tau_j$ and due to the use of abc for both update and strong-op messages.

Therefore, we are left with the case that $op_k^2$ is a weak write that does not influence $p_j$ and $op_k^1$ is a strong operation. Recall that $op_k^2$ does not influence $p_j$ only if its update message is delivered inside the partial execution interval of some weak write $w_j$ to the same location. In this case, $op_k^2$ is ordered in $\tau_j$ immediately before the invocation of $w_j$. Since the update message of $op_k^2$ is delivered inside the partial execution interval of $w_j$, the update message of $w_j$ is delivered in $p_k$ after the ack message that $p_j$ sent for $op_k^1$. Hence, the strong-op message of $op_k^1$ is delivered in $p_j$ before the invocation of $w_j$. Therefore, $op_k^2$ is ordered in $\tau_j$ after $op_k^1$ and the claim follows.

Theorem 6.5 Every execution generated by the algorithm described in this section is hybrid consistent.

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

- By Lemma 6.2, each sequence $\tau_j$ is legal.

- The construction of $\{\tau_j\}_{j=1}^n$ guarantees that each $\tau_j$ is a permutation of $ops(\sigma)$, which satisfies Condition 1 in the definition of hybrid consistency.

- By Lemma 6.4, every operation in each sequence $\tau_j$ is ordered between its previous and next strong operations by the same process, which satisfies Condition 2 in the definition of hybrid consistency.
• By Lemma 6.1, there exists a linearization $\rho$ of all operations in $\sigma$ such that every sequence $\tau_j$ agrees with $\rho$ on the order of strong operations, which satisfies Condition 3 in the definition of hybrid consistency.

• By Lemma 6.3, for every sequence $\tau_j$, $\tau_j \mid j = \sigma \mid j$, which satisfies Condition 4 in the definition of hybrid consistency.

6.1.3 Accessing Strong Objects with Weak Reads

In this section we show that for objects that support read, write and read-modify-write operations, the algorithm in Figure 6.1 implements hybrid consistency even if weak read operations and strong operations are allowed to access the same objects. We start by generalizing the definitions of “reads from”, “legal read”, “illegal read”, and “obliterating write” to encompass read-modify-write operations in the natural way. That is, every read and every write in the original definitions of these terms can be also a read-modify-write in the generalized definitions.

For each process $p_j$, we build a sequence of operations $\tau_j$ as in Section 6.1.2, with a small modification in the way weak reads are inserted into $\tau_j$. Specifically, let $\tau_j$ be the sequence of all operations in $\sigma$ that are executed by $p_j$, ordered by the order they are executed in $p_j$. For each $\tau_j$, add every weak write $w_k$ that is not executed by $p_j$ immediately before the weak write by $p_j$ to the same object such that the update message of $w_k$ is delivered inside the partial execution interval of $w_j$. (If the update message of $w_k$ is delivered inside the potential execution intervals of more than one weak write to the same object, then let $w_j$ be one that is ordered first among them in $\tau_j$.) Next, insert every weak read $r_k$, $k \neq j$, in the first possible place which is after the operation it reads from (if exists) and after the previous strong operations by $p_k$ (if exists), but before the next strong operation by $p_k$ (if exists). (If there exists a next strong operation $sop_k$ and the write $r_k$ reads from is ordered after $sop_k$, then $r_k$ is ordered immediately before $sop_k$.) Note that if $r_k$ reads from an object which is accessed by strong operations, then since this object is not accessed by weak writes, the operation $r_k$ reads from is always ordered between the previous and next strong operations by $p_k$ (if they exist).

Note that the proofs of Lemmas 6.1 and 6.3 do not rely on the assumption that weak reads do not access the same objects as strong operations. Therefore, these lemmas are valid even
without this assumption. It is also clear from the construction of \( \{ \tau_j \}_{j=1}^n \) that each \( \tau_j \) is a permutation of \( \text{ops}(\sigma) \). Thus, we only need to show that each \( \tau_j \) is legal and that in each \( \tau_j \) each operation is ordered between the previous and next strong operations by the same process.

The next lemma replaces Lemma 6.2 of Section 6.1.2.

**Lemma 6.6** Every sequence \( \tau_j \) is legal.

**Proof:** Assume, by way of contradiction, that there exists a sequence \( \tau_j \) which is not legal. By Lemma 6.2 and the way weak reads are inserted, this can only happen if there exists a weak read \( r_i(x, v) \) that reads from a strong operation, \( i \neq j \), and \( r_i(x, v) \) is ordered in \( \tau_j \) after an obliterating write. (By the construction of \( \tau_j \), the other cases are already covered by Lemma 6.2.) Let \( swo_i(x, u) \) be the first obliterating write for \( r_i(x, v) \) in \( \tau_j \). Denote the previous strong operation of \( p_k \) by \( sop_k \) and the operation \( r_i(x, v) \) reads from by \( swo_i(x, v) \). Thus, \( swo_i(x, v) \xrightarrow{\tau_i} swo_k(x, u) \xrightarrow{\tau_j} sop_k \xrightarrow{\tau_j} r_i(x, v) \). Recall that strong operations are ordered in every sequence \( \tau_j \) according to the order their strong-op message is delivered and that this order is unique. Hence, the strong-op message of \( swo_k(x, u) \) is delivered in \( p_k \) after the strong-op message of \( swo_i(x, v) \) and before the strong-op message of \( sop_k \). Hence, \( swo_k(x, u) \) is executed by \( p_k \) between the execution of \( swo_i(x, v) \) and the execution of \( r_i(x, v) \). A contradiction to the assumption that \( r_i(x, v) \) reads from \( swo_i(x, v) \).

The next lemma replaces Lemma 6.4 of Section 6.1.2.

**Lemma 6.7** For each sequence of operations \( \tau_j \) and every pair of operations \( op_k^1 \) and \( op_k^2 \) such that either \( op_k^1 \) or \( op_k^2 \) is strong, if \( op_k^1 \xrightarrow{\tau_j} op_k^2 \), then \( op_k^1 \xrightarrow{\tau_j} op_k^2 \).

**Proof:** By Lemma 6.4 and the way weak read operations are inserted, we know that the claim holds if \( op_k^2 \) is not a strong operation or if \( op_k^1 \) is a weak read that reads from a strong operation and \( k \neq j \). Hence, we are left with the case that \( op_k^2 \) is a strong operation, \( op_k^1 \) is a weak read that reads from a strong operation and \( k \neq j \). Let \( swo_i \) be the operation \( op_k^1 \) reads from. Thus, \( swo_i \xrightarrow{\tau_i} op_k^1 \xrightarrow{\tau_j} op_k^2 \). By Lemma 6.1, \( swo_i \xrightarrow{\tau_j} op_k^2 \). Hence, by the rules for inserting weak reads, \( op_k^1 \xrightarrow{\tau_j} op_k^2 \).
We have shown that all the conditions in the definition of hybrid consistency are satisfied by any execution generated by the algorithm, even if weak read operations are allowed to access the same objects that are accessed by strong read, strong write and strong read-modify-write operations. Thus, the algorithm is a correct implementation for hybrid consistency under these assumptions.

Remark: It can be argued that this result holds for queue and stack operations as well. In this case, a read operation serves as a peek operation, i.e., it returns the head of the queue or the stack, respectively. The formal proof is omitted since it is almost the same as the proof for objects that support read, write and read-modify-write operations.

6.1.4 Complexity Analysis

Since the algorithm uses an atomic broadcast mechanism, the actual time and message complexity depends on the complexity of this mechanism. The complexity is analyzed with respect to \( t_{abc} \), the time required to broadcast a message using \( abc \), \( t_{send} \), the time required to send a message using \( \text{send}_abc \), \( n_{abc} \), the number of physical messages required to perform \( abc \) and \( n_{send} \), the number of physical messages required to perform \( \text{send}_abc \).

Weak operations are executed instantaneously. A strong operation broadcasts a strong-op message to all processes and then waits for the acknowledgements. Therefore, the time required to execute a strong operation is \( t_{abc} + t_{send} \).

The message complexity of the above algorithm is zero physical messages for weak reads, \( n_{abc} \) physical messages for weak writes and \( n_{abc} + n_{send} \) physical messages for strong operations.

In Section 6.3, we present an atomic broadcast that for which \( t_{abc} = t_{send} = 2d \), \( n_{abc} = 2n \) and \( n_{abc} = n+1 \). By using these figures, we get a message complexity of \( 3n+1 \) physical messages for strong operations and \( n + 1 \) physical messages for weak writes. The time complexity of strong operations is \( 4d \).

6.2 Providing Hybrid Consistency with Read/Write Objects

In this section we present an efficient asynchronous algorithm for providing hybrid consistency that supports read/write objects, accessible by either strong or weak operations. The algorithm
allows weak operations to be executed instantaneously, while keeping the response time for strong operations proportional to the network delay.

The basic idea behind the algorithm presented in this section is similar to the one in Section 6.1. However, the rules for invoking strong operations and for executing weak operations in this algorithm were modified in order to allow weak and strong operations to access the same objects. Strong operations and weak writes by the local process are executed, as before, when their corresponding message is delivered. Other weak writes, on the other hand, are executed under the following conditions: A weak write that is not concurrent\(^2\) with any other (weak or strong) write to the same object is always executed when its corresponding message is delivered. If there are two concurrent weak writes, then the first one is executed by every process and the second one is ignored by every process. Also, since weak and strong operations may access the same objects, when we build the views of the processes, it is sometimes required to reorder strong operations like we do with weak writes in Section 6.1. To guarantee that this reordering will not violate the requirements in the definition of hybrid consistency, processes send acknowledgements after executing weak writes and strong operations do not start until all acknowledgements for previous (weak) operations are delivered.

A more detailed description of the algorithm and its pseudo-code in Section 6.2.1 is followed by a correctness proof in Section 6.2.2 and a complexity analysis in Section 6.2.3.

### 6.2.1 The Algorithm

Every process maintains a local copy of the entire memory \(\text{mem}\), an array of integers \(\text{last\_mess}\), an array of ids \(\text{last\_id}\), an integer counter \(\text{lts}\) and another object \(\text{val}\), as described below.

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

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

\(^2\)Concurrency is with respect to the global logical time. In other words, two weak writes are concurrent if the same number of messages have been delivered at the invoking processes before they were invoked.
Finally, the variable `val` is used to temporarily store a value that a pending strong read should return, as described later.

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

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

In the proof of correctness that appears below, we explicitly build the view for each process, as required in the definition of hybrid consistency. During this construction, we sometimes need to reorder strong reads with respect to the delivery events of their corresponding strong-read
messages if these messages are delivered between the invocation and termination of a weak write. However, this can cause a strong read to be ordered before a previous weak read by the same process, which is a violation of hybrid consistency. The dummy message which is broadcast between the invocation of a weak read and the invocation of a strong read guarantees that the invocation of a strong read is separated by at least a certain amount of “logical time” from the invocation of the last previous weak operation by the same process. This allows us to reorder the strong reads in the construction of the views in the proof of correctness, without causing a strong read to be ordered before a previous weak read. Note that there is no need to send a dummy message after a weak write since weak writes broadcast an update message; this already guarantees that there is enough “logical time” between the invocation of the weak write and the strong operation.

The precise code appears in Figures 6.3 and 6.4.

6.2.2 Proof of Correctness

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

The construction goes as follows: For each process \( p_j \), we first build the sequence \( \tau_j' \), consisting of all (weak and strong) write operations, all local weak read operations, and all delivery events. The operations are ordered according to the order they occurred in \( p_j \). The delivery events serve as markers which are useful for the rest of the proof. They are ordered according to the order they occurred, such that each delivery event is ordered after the corresponding operation. We show that \( T' = \{\tau_j'\}_{j=1} \) obeys all the requirements in the definition of hybrid consistency, except for including all operations in \( ops(\sigma) \) (Lemmas 6.8, 6.9, 6.10 and 6.12). Following this, we insert the strong reads into \( T' \), creating a new set of sequences \( T'' = \{\tau_j''\}_{j=1} \), such that the strong reads are legal. \( T'' \) obeys all the requirements in the definition of hybrid consistency, except for including all operations in \( ops(\sigma) \) (Lemmas 6.14, 6.17, 6.18 and 6.19). Next, for each sequence \( \tau_j'' \), we insert all weak reads by other processes, creating a new set of sequences \( T''' = \{\tau_j'''\}_{j=1} \) such that all reads in each \( \tau_j''' \) are legal. \( T''' \) obeys all the requirements in the definition of hybrid consistency except for having the delivery events (Lemmas 6.20 and 6.21). Finally, we construct the desired set of sequences \( T \) by removing all delivery events from \( T''' \), and show that \( T \) obeys all the requirements in the definition of hybrid consistency.
The state of each process $p_i$ consists of the following components:

- $\text{mem}$: copy of every object, initially equal to its initial value
- $\text{last.mess}$: array of integers with one entry for each object, all initially 0
- $\text{last.id}$: array of ids with one entry for each object, all initially 0
- $\text{lts}$: integer, initially 0 (serial number of the last message delivered)
- $\text{missing.acks}$: integer, initially 0 (number of acknowledges that are expected)
- $\text{val}$: an object

Read($x$):

$\text{generate Return}(x, \text{mem}[x])$  /* This is the response event */

Write($x,v$):

$\text{mem}[x] := v$

$\text{abc < update, } x, v, \text{lts}>$

$\text{missing.acks} := \text{missing.acks} + n$

$\text{generate } \text{Ack}(x)$  /* This is the response event */

SRead($x$):

if the last previous operation by $p_i$ was a weak read then

$\text{abc < dummy}>$

$\text{missing.acks} := \text{missing.acks} + n$

endif

wait until $\text{missing.acks} = 0$  /* This line is not atomic */

$\text{abc < strong-read, } x>$

$\text{missing.acks} := \text{missing.acks} + n$

SWrite($x,v$):

wait until $\text{missing.acks} = 0$  /* This line is not atomic */

$\text{abc < strong-write, } x, v>$

$\text{missing.acks} := \text{missing.acks} + n$

Figure 6.3: The algorithm — code for process $p_i$. 
received <update,x,v,s> from p_j:
    send_{abc} < ack,j >
    lts := lts + 1
    if ((s ≥ last_mess[x]) or (last_id[x] = j)) then
        last_mess[x] := lts
        last_id[x] := j
        if i ≠ j then
            mem[x] := v
        endif
    endif

received <dummy> from p_j:
    send_{abc} < ack,j >
    lts := lts + 1

received <strong-write,x,v> from p_j:
    send_{abc} < ack,j >
    lts := lts + 1
    last_mess[x] := lts
    mem[x] := v
    last_id[x] := j

received <strong-read,x> from p_j:
    send_{abc} < ack,j >
    lts := lts + 1
    if (j = i) then val := mem[x] endif

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

Figure 6.4: The algorithm — code for handling message receive events at process p_i.
We start with some definitions and notations. For every operation that causes a message to be broadcast, i.e., weak write, strong write or strong read, we denote by \( \text{del}(op) \) the delivery event of the corresponding message (\text{update}, \text{strong-write} or \text{strong-read}, accordingly) in \( p_k \). Given a strong read \( sr_i \), we denote by \( \text{rel}(sr_i) \) the last (\text{strong-read}, \text{strong-write}, \text{update} or \text{dummy}) message broadcast by \( p_i \) before the \text{strong-read} message of \( sr_i \). We slightly abuse the notation and denote by \( \text{del}(\text{rel}(sr_i)) \) the delivery event of \( \text{rel}(sr_i) \) in \( p_j \).

Recall that a partial execution interval of a weak write \( w_k(x,v) \) is the interval of events in \( \sigma \) between the invocation of \( w_k \) and the delivery of the \text{update} message of \( w_k \) by the atomic broadcast at \( p_k \). A weak write \( w_k(x,v) \) \text{is overwritten} by another (weak or strong) write \( w_j(x,u) \) if \( w_j(x,u) \) is executed in \( p_k \) during the partial execution interval of \( w_k(x,v) \). A \text{potential execution interval} of a weak write \( w_k(x,v) \) is the largest interval inside the partial execution interval of \( w_k \), starting with the invocation of \( w_k(x,v) \), in which no other write \( w_j(x,u) \), \( j \neq k \), is executed by \( p_k \). Note that a weak write updates the local copy of the object and returns immediately. The potential execution interval is the interval after a weak write updates the local copy of the object, but may still be overwritten. For example, Figure 6.5 demonstrate the potential and partial execution intervals of a weak write \( w_i(x,v) \) that is overwritten by another write \( w_k(x,u) \).

We now define when an operation \text{influences} a process. A weak write \( w_k(x,v) \) \text{influences} process \( p_j \), \( j \neq k \), if it updates the copy of \( x \) in \( p_j \). A weak write \( w_k(x,v) \) \text{influences} process \( p_k \) if it is not \text{overwritten}. A strong write \text{influences} every process.

Next, define the notion of an operation being \text{executed} by a process as follows. A strong operation \( sop_k \) is executed by \( p_j \) when the corresponding \text{strong-write} or \text{strong-read} mes-
sage is delivered at \( p_j \). A weak operation \( op_j \) is executed by \( p_j \) when the appropriate call event occurs. A weak write \( w_k \) that influences \( p_j, k \neq j \), is executed by \( p_j \) when the corresponding update message is delivered at \( p_j \).

We now turn to the details of the proof; for the rest of this section, fix some execution \( \sigma \) of the algorithm. Let \textit{abcast order} be the order by which all messages sent with \texttt{abc} are delivered; note that this order is uniquely defined.

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

\[
\alpha_i(op_k^1) \rightarrow \alpha_i(op_l^2) \text{ if and only if } \alpha_i(op_k^1) \rightarrow \alpha_i(op_l^2), \text{ for every } i \text{ and } j.
\]

**Lemma 6.8** There exists a linearization \( \rho \) of all the operations in \( \sigma \) such that for every pair of strong operations \( sop_k \) and \( sop_l \) in \( \tau_j' \), \( sop_k \rightarrow sop_l \) if and only if \( sop_k \rightarrow sop_l \), for every \( \tau_j' \).

**Proof:** The strong operations appear in every \( \tau_j' \) in the abcast order. Since a strong operation does not return before all its \texttt{ack} messages return, the real time of its delivery is always between the real time of its invocation and the real time of its termination. Thus, the order in which all strong operations appear in every \( \tau_j' \) is a linearization.

**Lemma 6.9** For every sequence \( \tau_j' \), \( \tau_j'|j = \sigma|j \).

**Proof:** Recall that weak operations by \( p_j \) are ordered according to their invocation and strong operations by \( p_j \) are ordered according to their delivery events. Since a strong operation does not return until all \texttt{ack} messages are delivered, every strong operation by \( p_j \) is ordered
between the previous operation (if it exists) and next operations (if it exists) by the \( p_j \). Hence, \( \tau'_j[j = \sigma[j, \text{ for every sequence } \tau'_j. \]

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

**Lemma 6.10** For every sequence \( \tau'_j \), all the weak reads of \( p_j \) are legal in \( \tau'_j \).

**Lemma 6.11** If a write influences some process, then it influences every process.

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

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

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

**Lemma 6.12** For each sequence of operations \( \tau'_j \) and every pair of operations \( op^1_k \) and \( op^2_k \) in \( \tau'_j \) such that either \( op^1_k \) or \( op^2_k \) is strong, \( op^1_k \xrightarrow{\tau'_j} op^2_k \) if and only if \( op^1_k \xrightarrow{\sigma} op^2_k \).
**Proof:** By Lemma 6.9, the claim holds if \( k = j \). Thus, for the rest of the proof, we may assume that \( k \neq j \).

Assume, by way of contradiction, that for some sequence \( \tau'_j \) and two operations \( op^1_k \) and \( op^2_k \) such that either \( op^1_k \) or \( op^2_k \) is strong, \( op^1_k \overset{i}{\rightarrow} op^2_k \) but \( op^2_k \overset{\sigma}{\rightarrow} op^1_k \). We claim that \( op^1_k \) is a weak write that does not influence \( p_i \) and \( op^2_k \) is strong. Strong operations and weak writes that influence \( p_i \) are ordered immediately before their corresponding delivery events, and since weak writes are always ordered before their corresponding delivery event, a weak write that does not influence \( p_i \) may not be ordered after a later strong operation by the same process.

Assume, without loss of generality, that \( op^1_k \) writes to \( x \). Let \( w_i \) be the first write to \( x \) that influences \( p_i \) and is ordered after \( op^1_k \) in \( \tau'_j \). The existence of such a write is guaranteed by the assumption that \( op^1_k \) does not influence \( p_i \) and the construction of \( \tau'_j \); \( w_i \) is the write that caused \( op^1_k \) to be ordered before \( op^2_k \). The code of the algorithm also implies that \( w_i \) is delivered in \( p_i \) before \( op^2_k \).

Assume that \( i \neq j \), or if \( i = j \) and \( op^2_k \) is delivered after the completion of the partial execution interval of \( w_i \) in \( p_j \). By Lemma 6.11, \( w_i \) influences every process. Thus, \( w_i \) is delivered in every process before \( op^2_k \). In particular, \( w_i \) is delivered in \( p_k \) before \( op^2_k \). Since \( op^2_k \) is a strong operation and since \( op^1_k \) is invoked after \( op^2_k \), \( w_i \) is delivered in \( p_k \) before \( op^1_k \) starts. Therefore, there exists another write \( w_i(x,w) \) that influences \( p_i \) and is delivered in \( p_i \) between \( w_i \) and \( op^1_k \). Thus, \( w_i(x,w) \) is ordered in \( \tau'_j \) between \( w_i \) and \( op^1_k \). A contradiction to the assumption that \( w_i \) is the first write to \( x \) that influences \( p_i \) is ordered after \( op^1_k \) in \( \tau'_j \).

Hence, assume that \( i = j \) and \( op^2_k \) is delivered inside the partial execution interval of \( w_i \). Therefore, the \( \text{ack} \) message of \( op^2_k \) is delivered in \( p_k \) after the \( \text{update} \) message of \( w_i \). By Lemma 6.11, \( w_i \) influences \( p_k \) and is delivered in \( p_k \) before \( op^1_k \) starts. Therefore, there exists another write \( w_i(x,v) \) that influences \( p_j \) and is delivered in \( p_k \) before \( op^1_k \). Thus, \( w_i(x,v) \) is ordered in \( \tau'_j \) between \( w_i \) and \( op^1_k \). A contradiction to the assumption that \( w_i \) is the first write to \( x \) that influence \( p_j \) and is ordered after \( op^1_k \) in \( \tau'_j \).

Thus, we have shown that the sequences \( \{ \tau'_j \}_{j=1}^n \) obey all the requirements in the definition.
of hybrid consistency, except for including all the operations of $ops(\sigma)$. Next, we insert the missing strong reads to each $\tau'_j$ without violating the requirements of the definition of hybrid consistency.

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

$$\tau''_i = \ldots w_i(x,v) \ldots sop_q \ldots sr_k(y,u) \ldots del_j(sr_i(x,u)) \ldots del_j(w_i(x,v))$$

then $sr_i(x,v)$ must be inserted before $w_i(x,v)$ and $sop_q$ in $\tau''_i$. Therefore, $sr_i(x,v)$ must be inserted before $sop_q$ in $\tau''_i$, too. In addition if $sr_k(y,u)$ is a strong read that is dragged by the insertion of $sr_i(x,v)$ (we explain the rules for this below), then $sr_k(y,u)$ must be ordered before $sr_i(x,v)$ and $sop_q$ both in $\tau''_i$ and $\tau''_j$.

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

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

\[
\tau''_j = \ldots w_j(x,v), sw_q, sr_i(y,u), w_2(z,w), sr_i(z,w), del_j, (sr_i(x,u)), del_j, (w_j(x,v)), \ldots
\]

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

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

**Lemma 6.13** A strong read reads from a write that influences every process.

**Proof:** Let \( sr_i \) be a strong read that reads from some write \( w_k \). If \( w_k \) is strong then by definition, \( w_k \) influences every process. So, assume that \( w_k \) is weak. If \( k = i \), then since \( sr_i \) is a strong operation, it is not executed until the partial execution interval of \( w_k \) is completed. Therefore, \( w_k \) is not overwritten and by Lemma 6.11, \( w_k \) influences every process. If \( k \neq i \), then \( w_k \) influences \( p_i \) and by Lemma 6.11, \( w_k \) influences every process. ■

**Lemma 6.14** Every strong read is legal in every sequence \( \tau''_j \).
**Proof:** We prove that every strong read is ordered in every sequence $\tau''_i$ after the write it reads from. Since by definition, every strong read appears in every sequence $\tau''_i$ before any obliterating write operation, this will imply that the strong read is legal.

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

Assume that $sr_i(x, v)$ is ordered in $\tau''_i$ before the write it reads from, and denote this write by $w_k(x, v)$. In particular, $sr_i(x, v)$ is ordered before $del_i(w_k(x, v))$ in $\tau''_i$. Thus, $del_i(sr_i(x, v))$ is ordered in some sequence $\tau''_p$ inside a potential execution interval of some weak write $w_l(x, u)$ that includes a strong operation $sop_q$ which is ordered in $\tau''_i$ before $del_i(w_k(x, v))$. (If there is no such $sop_q$, then there is no reason to order $sr_i(x, v)$ in $\tau''_i$ before the write it reads from.) By the assumption that $sr_i(x, v)$ reads from $w_k(x, v)$ and by Lemma 6.13, $w_k(x, v)$ influences every process. Therefore, $del_i(w_k(x, v))$ is ordered in $\tau''_i$ before the potential execution interval of $w_l(x, u)$. Since we assumed that $sop_q$ is ordered in $\tau''_i$ before $del_i(w_k(x, v))$, and since $del_i(sop_q)$ is ordered after $del_i(w_k(x, v))$, it follows that $sop_q$ is a strong read and the write $sop_q$ reads from is ordered inside the potential execution interval of $w_l(x, u)$. Denote the write $sop_q$ reads from by $w_m$. By Lemma 6.13, both $w_m$ and $w_k(x, v)$ influence every process and therefore both $w_m$ and $w_k(x, v)$ are ordered immediately before their delivery events. Thus, $w_m$ is ordered after $w_k(x, v)$ in $\tau''_i$. By the minimality of $sr_i(x, v)$, $sop_q$ is ordered after $w_m$ in $\tau''_i$, and therefore after $w_k(x, v)$. A contradiction.

Assume that $sr_i(y, w)$ is dragged by $sr_i(x, v)$ before the write it reads from, and denote this write by $w_k(y, w)$. In particular, $sr_i(y, w)$ is dragged by $sr_i(x, v)$ before $del_i(sop_i(y, w))$ in $\tau''_i$. By the minimality of $sr_i(x, v)$, $sr_i(y, w)$ is ordered after $w_k(y, w)$ in every sequence $\tau''_p$ before the insertion of $sr_i(x, v)$. Thus, $del_i(sop_i(x, v))$ and $del_i(sop_i(y, w))$ are ordered inside a potential execution interval of some weak write $w_l(x, u)$ that includes a strong operation $sop_q$ which is ordered in $\tau''_i$ before $del_i(w_k(y, w))$. Since we assumed that $sop_q$ is ordered in $\tau''_i$ before $del_i(w_k(y, w))$, and since $del_i(sop_q)$ is ordered after $del_i(w_k(y, w))$, it follows that $sop_q$ is a strong read and the write $sop_q$ reads from is ordered inside the potential execution interval of $w_l(x, u)$. Denote the write $sop_q$ reads from by $w_m$. By Lemma 6.13, both $w_m$ and $w_k(x, v)$
influence every process and therefore both \( w_m \) and \( w_k(x,v) \) are ordered immediately before their delivery events. Thus, \( w_m \) is ordered after \( w_k(x,v) \) in \( \tau'' \). By the minimality of \( sr_i(x,v) \), \( sop_k \) is ordered after \( w_m \) in \( \tau'' \), and therefore after \( w_k(y,w) \). A contradiction.

Recall that given a strong operation \( sop_k \), we denote by \( \text{del}_j(\text{rel}(sop)) \) the delivery event of the previous message that was broadcast by \( p_k \) at \( p_j \). We show:

**Lemma 6.15** Every strong read \( sr_i(x,v) \) is ordered after \( \text{del}_j(\text{rel}(sr_i(x,v))) \) in every sequence \( \tau'' \).

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

Assume that \( sr_i(x,v) \) is ordered in \( \tau'' \) before \( \text{del}_j(\text{rel}(sr_i(x,v))) \). Thus, there exists a sequence \( \tau'' \) for which \( \text{del}_i(sr_i(x,v)) \) is ordered in \( \tau'' \) inside a potential execution interval of some weak write \( w_i(x,u) \) that includes a strong operation \( sop_k \) which is ordered in \( \tau'' \) before \( \text{del}_j(\text{rel}(sr_i(x,v))) \). Since \( sr_i(x,v) \) does not return until all \text{ack} messages of \( \text{rel}(sr_i(x,v)) \) are delivered, \( \text{del}_i(\text{rel}(sr_i(x,v))) \) is ordered in \( \tau'' \) before the potential execution interval of \( w_i(x,u) \).

Since \( sop_k \) is ordered before \( \text{del}_j(\text{rel}(sr_i(x,v))) \) in \( \tau'' \) and after \( \text{del}_i(\text{rel}(sr_i(x,v))) \) in \( \tau'' \), then \( sop_k \) is a strong read and the write \( sop_k \) reads from is ordered inside the potential execution interval of \( w_i(x,u) \). Denote the write \( sop_k \) reads from by \( w_m \). By Lemma 6.13, \( w_m \) influences every process and is therefore ordered in \( \tau'' \) immediately before \( \text{del}_j(w_m) \). Thus, \( w_m \) is ordered after \( \text{del}_i(\text{rel}(sr_i(x,v))) \) in \( \tau'' \). By Lemma 6.14, \( sop_k \) is ordered after \( w_m \) in \( \tau'' \), and therefore after \( \text{del}_j(\text{rel}(sr_i(x,v))) \). A contradiction.

Assume that \( sr_s(y,w) \) is dragged by \( sr_i(x,v) \) before \( \text{del}_j(\text{rel}(sr_s(y,w))) \) in \( \tau'' \). By the minimality of \( sr_i(x,v) \), \( sr_s(y,w) \) is ordered after \( \text{del}_i(\text{rel}(sr_s(y,w))) \) in every sequence \( \tau'' \) before the insertion of \( sr_i(x,v) \). Thus, there exists a sequence \( \tau'' \) for which \( \text{del}_i(sr_i(x,v)) \) and \( \text{del}_i(sr_s(y,w)) \) are ordered inside a potential execution interval of some weak write \( w_i(x,u) \) that includes a strong operation \( sop_k \) which is ordered in \( \tau'' \) before \( \text{del}_j(\text{rel}(sr_s(y,w))) \). Since \( sop_k \)
is ordered before \( \text{del}_j(\text{rel}(sr_i(y,w))) \) in \( \tau''_i \) and after \( \text{del}_i(\text{rel}(sr_i(y,w))) \) in \( \tau''_i \), it follows that \( sop_q \) is a strong read and the write \( sop_q \) reads from is ordered inside the potential execution interval of \( w_i(x,u) \). Denote the write \( sop_q \) reads from by \( w_m \). By Lemma 6.13, \( w_m \) influences every process and therefore is ordered in \( \tau''_i \) immediately before \( \text{del}_j(w_m) \). Thus, \( w_m \) is ordered after \( \text{del}_j(\text{rel}(sr_i(y,w))) \) in \( \tau''_i \). By Lemma 6.14, \( sop_q \) is ordered after \( w_m \) in \( \tau''_i \), and therefore after \( \text{del}_j(\text{rel}(sr_i(y,w))) \). A contradiction. 

**Lemma 6.16** For every pair of strong operations \( sop_k \) and \( sop_l \) and every pair of sequences \( \tau''_i \) and \( \tau''_i \), \( sop_k \xrightarrow{\tau''_i} sop_l \) if and only if \( sop_k \xrightarrow{\tau''_i} sop_l \).

**Proof:** Assume, by way of contradiction, that there exist two strong operations \( sop_k \) and \( sop_l \) and two sequences \( \tau''_i \) and \( \tau''_i \) such that \( sop_k \xrightarrow{\tau''_i} sop_l \) but \( sop_k \xrightarrow{\tau''_i} sop_l \). By Lemma 6.8 and since strong writes are not dragged by strong reads, this can only happen if at least one of \( sop_k \) or \( sop_l \) is a strong read.

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

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

**Lemma 6.17** There exists a linearization \( \rho \) of the operations in \( \sigma \) such that for every pair of strong operations \( sop_k \) and \( sop_l \), \( sop_k \xrightarrow{\tau''_i} sop_l \) if and only if \( sop_k \xrightarrow{\tau''_i} sop_l \) for every \( \tau''_i \).

**Proof:** In this proof, we show that there exists a linearization of all strong operations in \( \sigma \). This linearization can than be extended to a linearization that includes all operations in \( \sigma \).
Note that the definition of hybrid consistency requires a linearization of all the operations but does not impose any restriction on the weak operations that appear in this linearization.

In Lemma 6.16, we showed that all strong operations appear in the same order in all sequences $\tau''_j$. We now show that this order preserves the order implied by $\sigma$. Assume, by way of contradiction, that the order by which all strong operations appear in every sequence $\tau''_j$ is not a linearization. Note that strong reads are always inserted before their corresponding delivery event. By Lemma 6.8 and since strong writes are not dragged by strong reads, the total order is not a linearization only if there exist a strong read $sr_i(x,v)$ and another strong operation $sop_k$ such that $sop_k$ terminates before $sr_i(x,v)$ begins but $sr_i(x,v) \rightarrow \tau'_j$ for every sequence $\tau''_j$. Since, by definition, $sop_k \rightarrow del_j(sop_k)$ for every sequence $\tau''_j$, this can only happen in one of two cases:

**Case 1:** The delivery event of the strong-read message of $sr_i(x,v)$ is ordered inside the potential execution interval of some $w_i(x,v)$, and $sop_k$ is also ordered inside the same potential execution interval. Then the ack message for $sop_k$ is delivered in $p_k$ after the update message of $w_i(x,u)$, while the strong-read message of $sr_i(x,v)$ is delivered in all processes before the update message of $w_i(x,u)$. Hence, the strong-read message of $sr_i(x,v)$ is delivered in $p_k$ before $sop_k$ terminates. A contradiction to the assumption that $sop_k$ terminates before $sr_i(x,v)$ begins.

**Case 2:** $sr_i(x,v)$ is dragged by some other strong read $sr_q(y,w)$ before $sop_k$. Thus, the strong-read message of $sr_q(y,w)$ is delivered inside the potential execution interval of some weak write $w_i(y,z)$ that includes both $del_i(sop_k)$ and $del_i(sr_i(x,v))$. Therefore, the ack message for $sop_k$ is delivered in $p_k$ after the update message of $w_i(x,v)$, while the strong-read message of $sr_i(x,v)$ is delivered in all processes before the update message of $w_i(y,z)$. Hence, the strong-read message of $sr_i(x,v)$ is delivered in $p_k$ before $sop_k$ terminates. A contradiction to the assumption that $sop_k$ terminates before $sr_i(x,v)$ begins.

**Lemma 6.18** For every sequence $\tau''_j$, $\tau''_j|j = \sigma|j$.

**Proof:** Consider any two operations $op^1_j$ and $op^2_j$ such that $op^1_j \rightarrow op^2_j$. We show that $op^1_j \rightarrow op^2_j$ by a simple case analysis. If neither $op^1_j$ nor $op^2_j$ is a strong read, then the claim holds by Lemma 6.9 and since the insertion of strong reads does not change the order of other operations. If both $op^1_j$ and $op^2_j$ are strong reads, then the claim holds by Lemma...
6.17. If only \( op_i^1 \) is a strong read, then the claim holds since strong reads are inserted before their corresponding delivery event. If, on the other hand, only \( op_i^2 \) is a strong read, then by Lemma 6.15, \( del_j(rel(op_i^2)) \xrightarrow{\tau''_j} op_i^2 \). By construction of \( \tau''_j \), \( op_i^1 \xrightarrow{\tau''_j} del_j(rel(op_i^2)) \). Thus, \( op_i^1 \xrightarrow{\tau''_j} op_i^2 \) and the claim holds. (Note that we have covered all the cases since \( \tau''_j \) does not include weak reads by other processes.) 

Lemma 6.19 For each sequence of operations \( \tau''_j \) and every pair of operations \( op_k^1 \) and \( op_k^2 \) in \( \tau''_j \) such that either \( op_k^1 \) or \( op_k^2 \) is strong, \( op_k^1 \xrightarrow{\tau''_j} op_k^2 \) if and only if \( op_k^1 \xrightarrow{} op_k^2 \).

Proof: By Lemma 6.18, the claim holds for \( k = j \). Thus, we assume for the rest of the proof that \( k \neq j \). Assume, by way of contradiction, that for some sequence \( \tau''_j \) and two operations \( op_k^1 \) and \( op_k^2 \) such that either \( op_k^1 \) or \( op_k^2 \) is strong, \( op_k^1 \xrightarrow{\tau''_j} op_k^2 \) but \( op_k^1 \xrightarrow{} op_k^2 \). By Lemma 6.12 and Lemma 6.17 and the construction of \( \tau''_j \), this can only happen in one of the following cases:

Case 1: \( op_k^1 \) is a strong read and \( op_k^2 \) is a weak write. By the construction of \( \tau''_j \), \( op_k^2 \) is ordered before \( del_j(op_k^1) \). In particular, \( op_k^2 \) is ordered in \( \tau''_j \) before \( del_j(rel(op_k^1)) \). By Lemma 6.15, \( op_k^1 \) is ordered in \( \tau''_j \) after \( del_j(rel(op_k^1)) \). Thus, \( op_k^2 \xrightarrow{\tau''_j} op_k^1 \). A contradiction.

Case 2: \( op_k^1 \) is a strong read and \( op_k^2 \) is a weak write that does not influence \( p_j \). By definition, \( op_k^2 \) is ordered before \( del_j(op_k^2) \) in every sequence \( \tau''_j \). Assume, without loss of generality, that \( op_k^1 \) writes to \( x \), and let \( w_i(x, w) \) be the last write that influences \( p_j \) before the delivery of the update message of \( op_k^1 \). Thus, \( op_k^1 \) is invoked before the delivery of the update or strong-write message of \( w_i(x, w) \) in \( p_k \). Moreover, \( op_k^2 \) is ordered in \( \tau''_j \) before \( op_k^2 \) because \( w_i(x, v) \) is ordered in \( \tau''_j \) before \( op_k^2 \). Since \( op_k^2 \) does not return before all of its ack messages are delivered, and since every ack message is sent using \text{send
\_ack}, the ack message for \( op_k^2 \) is delivered in \( p_k \) after the update or strong-write message of \( w_i(x, w) \). Thus, \( op_k^1 \) is invoked after the delivery of the update or strong-write message of \( w_i(x, w) \). A contradiction.

We now insert the weak reads. A sequence \( \tau'''_j \) is constructed from each sequence \( \tau''_j \), by inserting into \( \tau''_j \) all the weak reads that are invoked by other processes than \( p_j \), one after the other as follows: Every weak read \( r_k(x, v), k \neq j \), is inserted in the first possible place such that it will be ordered after the previous strong operation by \( p_k \) (if there is one) and after the

78
write it reads from (if there is one). If there is no previous strong operation and the weak read returns the initial value of the object, then the weak read is inserted at the beginning of the sequence.

**Lemma 6.20** For each sequence of operations \( \tau''_j \) and every pair of operations \( op^1_k \) and \( op^2_k \) such that either \( op^1_k \) or \( op^2_k \) is strong, \( op^1_k \xrightarrow{\tau''_j} op^2_k \) if and only if \( op^1_k \xrightarrow{\tau''_j} op^2_k \).

**Proof:** Assume, by way of contradiction, that there exist a sequence \( \tau''_j \) and two operations \( op^1_k \) and \( op^2_k \) such that either \( op^1_k \) or \( op^2_k \) is strong, \( op^1_k \xrightarrow{\tau''_j} op^2_k \) but \( op^1_k \xrightarrow{\tau''_j} op^2_k \). By Lemma 6.18, the claim holds if \( k = j \). By Lemma 6.19 and since the insertion of weak reads does not change the order of other operations in \( \tau''_j \), the claim holds if neither \( op^1_k \) nor \( op^2_k \) is a weak read. Moreover, since weak reads are inserted after the previous strong operation by the same process, the only case in which \( op^1_k \) may be ordered in \( \tau''_j \) before \( op^1_k \) is when \( op^1_k \) is a weak read, \( op^1_k \) is a strong operation, \( k \neq j \), and the write \( op^2_k \) reads from is ordered after \( op^1_k \) in \( \tau''_j \).

Assume, without loss of generality, that \( op^2_k \) reads from \( w_l(x, v) \). Lemma 6.19 and the assumption that \( op^2_k \) reads from \( w_l(x, v) \) imply that \( l \neq k \). Thus, \( w_l(x, v) \) influences every process and is therefore ordered in \( \tau''_j \) immediately before \( del_j(w_l(x, v)) \). If \( op^1_k \) is a strong write, then it is ordered in \( \tau''_j \) immediately before \( del_j(op^1_k) \). Thus, \( del_j(op^1_k) \xrightarrow{\tau''_j} w_l(x, v) \xrightarrow{\tau''_j} del_j(w_l(x, v)) \). If, on the other hand, \( op^1_k \) is a strong read, then by Lemma 6.15, \( op^1_k \) is ordered in \( \tau''_j \) after \( del_j(rel(op^1_k)) \). Thus, \( del_j(rel(op^1_k)) \xrightarrow{\tau''_j} w_l(x, v) \xrightarrow{\tau''_j} del_j(w_l(x, v)) \). In either case, by the use of atomic broadcast, the **update** or **strong-write** message of \( w_l(x, v) \) is delivered in \( p_k \) after \( op^2_k \) is executed. A contradiction to the assumption that \( op^2_k \) reads from \( w_l(x, v) \). □

**Lemma 6.21** Every read is legal in every sequence \( \tau''_j \).

**Proof:** By Lemma 6.14, all strong reads are legal in any of the sequences \( \tau''_j \). Thus, assume, by way of contradiction, that there exist a sequence \( \tau''_j \) and a weak read \( r_i(x, v) \) such that \( r_i(x, v) \) is not legal in \( \tau''_j \). Denote by \( w_i(x, v) \) the write \( r_i(x, v) \) reads from and by \( sop_i \) the next strong operation by \( p_i \). By Lemma 6.10 and since the insertion of (strong or weak) read operations does not change the order of write operations (by any process) or the order of weak reads by \( p_i \), \( i \neq j \). Thus, by the rules for inserting weak reads, there exists an obliterating write \( w_k(x, u) \) such that \( w_l(x, v) \xrightarrow{\tau''_j} w_k(x, u) \xrightarrow{\tau''_j} sop_i \).
If \( k \neq j \), then the update or strong-write message of \( w_k(x, u) \) is delivered at \( p_i \) after the execution of \( w_l(x, v) \) and before the delivery of the ack message of \( sop_i \). Thus, either \( w_k(x, u) \) or another write to \( x \) is executed by \( p_i \) between the execution of \( w_l(x, v) \) and the invocation of \( r_i(x, v) \). A contradiction to the assumption that \( r_i(x, v) \) reads from \( w_l(x, v) \).

Thus, \( k \neq j \). We may assume, without loss of generality, that \( w_k(x, u) \) influences \( p_j \). Otherwise, there exists another write \( w_q(x, w) \) that influences \( p_j \) such that \( w_l(x, v) \stackrel{\tau_j^m}{\longrightarrow} w_k(x, u) \stackrel{\tau_j^m}{\longrightarrow} w_q(x, w) \stackrel{\tau_j^m}{\longrightarrow} sop_i \) and we could have chosen \( w_q(x, w) \). Thus, \( w_k(x, u) \) is ordered in \( \tau_j^m \) immediately before \( del_j(w_k(x, u)) \). By Lemma 6.11, \( w_k(x, u) \) influences every process. By the construction of \( \tau_j^m \), \( sop_i \) is ordered before \( del_j(sop_i) \). Thus, the update or strong-write message of \( w_k(x, u) \) is delivered at \( p_i \) after the execution of \( w_l(x, v) \) and before the delivery of the strong-write or strong-read message of \( sop_i \). Thus, \( w_k(x, u) \) is executed by \( p_i \) between the execution of \( w_l(x, v) \) and the invocation of \( r_i(x, v) \). A contradiction to the assumption that \( r_i(x, v) \) reads from \( w_l(x, v) \).

Finally, for each process \( p_j \), create a sequence of operations \( \tau_j \) by removing all delivery events from \( \tau_j^m \).

**Theorem 6.22** Every execution generated by the algorithm is hybrid.

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

- By Lemma 6.21, every sequence \( \tau_j \) is legal.
- By construction, every sequence \( \tau_j \) is a permutation of \( ops(\sigma) \), which satisfies Condition 1 in the definition of hybrid consistency.
- By Lemma 6.20, for every pair of operations \( op_1 \) and \( op_2 \) such that either \( op_1 \) or \( op_2 \) is strong, \( op_1 \stackrel{\tau_j}{\longrightarrow} op_2 \) if and only if \( op_1 \stackrel{\epsilon}{\longrightarrow} op_2 \), which satisfies Condition 2 in the definition of hybrid consistency.
- By Lemma 6.17, there exists a linearization \( \rho \) of all operations in \( ops(\sigma) \) such that for every pair of strong operations \( sop_k \) and \( sop_k \), \( sop_i \stackrel{\tau_j}{\longrightarrow} sop_k \) if and only if \( sop_i \stackrel{\rho}{\longrightarrow} sop_k \), which satisfies Condition 3 in the definition of hybrid consistency.
• By Lemma 6.18, for every sequence \( \tau_j, \tau_j[j = \text{ops}(\sigma)]j \), which satisfies Condition 4 in the definition of hybrid consistency.

6.2.3 Complexity Analysis

As in Section 6.1.4, the actual time and message complexity depends on the complexity of this mechanism. So, we shall make the analysis w.r.t. \( t_{abc}, t_{send}, n_{abc} \) and \( n_{send} \). Weak reads are executed locally and cause no messages to be sent. Weak writes are executed instantaneously. Each weak write requires \( n_{abc} \) physical messages in order to broadcast its update message and \( n \cdot n_{send} \) physical messages for the corresponding ack messages.

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

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

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

As before, we can use the algorithm described below in Section 6.3 for which \( n_{abc} = n + 1 \), \( n_{send} = 2 \) and \( t_{abc} = t_{send} = 2d \). By using these figures, we have a message complexity of \( 3n + 1 \) physical messages for both strong and weak write operations, and \( 6n + 2 \) physical messages for strong read operations. The time complexity of the strong operations is \( 8d \).
6.3 Atomic Broadcast

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

Much work has been done on atomic broadcast, and several atomic broadcast algorithms have been developed [12, 19, 22, 23, 24]. In particular, some of them deliver messages in $2d$ time, meeting our time complexity requirement.

In the rest of this section, we show an efficient centralized atomic broadcast algorithm, which has a better message complexity than any other algorithm known to us, i.e., $n + 1$ messages for $\text{abc}$ and 2 messages for $\text{send\_abc}$. The delivery time of this algorithm is $2d$ for both $\text{abc}$ and $\text{send\_abc}$.

Recall that we assume a system of $n$ processes connected by some interconnection network. We assume that the links are reliable and FIFO. A special process, $p_0$, is used to order the messages of all processes. Thus, $\text{abc}$ is implemented by sending the message to the special process which forwards this message to all processes. Similarly, $\text{send\_abc}$ is implemented by sending the message to the special process which forwards the message to the targeted process. Whenever a process receives a message from the special process, it delivers the message to the mcs. The precise code of the algorithm is given in Figure 6.6.

We are aware that on some specific architectures, this algorithm behaves worse than other distributed algorithms. On the other hand, in our abstract model, a virtual message sent from one process to the other is always counted as one physical message and its delivery time is not affected by the process ids. Our algorithm requires no bookkeeping and, in our model, it is as efficient as any other atomic broadcast algorithm we are aware of. The two other main problems of centralized algorithms, namely load balancing and fault tolerance, can be solved using round robin on the job of the special process and backoffs [24]. These issues, however, are beyond the scope of this work and are not discussed any further.
\textbf{abc} < m >:
\begin{itemize}
\item send < m,'abc'> to the special process (p_0)
\end{itemize}

\textbf{send\_abc} < m, j >:
\begin{itemize}
\item send < m,'send', j > to the special process (p_0)
\end{itemize}

received < m,'abc' > from p_j
\begin{itemize}
\item send < m,'ok'> to every process (including itself)
\end{itemize}

received < m,'send', k > from p_j
\begin{itemize}
\item send < m,'ok'> to p_k
\end{itemize}

received < m,'ok'> from p_j
\begin{itemize}
\item deliver < m >
\end{itemize}

Figure 6.6: Atomic broadcast — code for process p_i.
Chapter 7

Lower Bounds for Implementing Hybrid Consistency

In this chapter we show lower bounds for implementations of hybrid consistency. We show that in every implementation of hybrid consistency in which weak reads and weak writes are executed faster than the network delay, the time required to execute a strong read or a strong write is at least $\Omega(d)$. Following this, we show that the execution time of every read-modify-write operation, pop operation or dequeue operation is at least $\Omega(d)$, even if the operation is weak. These results indicate that the algorithms for implementing hybrid consistency, presented in Chapter 6, are optimal in their time complexity (up to a constant multiplication factor).

The proofs in Section 7.1 are slight modifications of the proofs for similar results that appeared previously in [19, 44, 46] and in Section 4.3.2. The proof in Section 7.2 is slightly different, but uses the same proof techniques. Finally, the proof in Section 7.3 is a slight modification of the proof for a similar result that appeared in [13].

7.1 Reads and Writes

**Theorem 7.1** For any implementation of hybrid consistency, $|read| + |write| \geq d$.

**Proof:** Assume, by way of contradiction, that there exists an implementation of hybrid
consistency for which \(|\text{read}| + |\text{write}| < d\). 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 exists a hybrid execution \(\sigma_1\) such that \(\text{ops}(\sigma_1)\) is

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

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

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

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

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

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

Since \(\sigma\) is hybrid, there exists a serialization \(\rho\) of \(\text{ops}(\sigma)\), such that for every \(i \in \{1, 2\}\), there exists a legal serialization \(\tau_i\) of \(\text{ops}(\sigma)\), that preserves: (a) the order of operations at \(p_i\), (b) the order between strong and weak operations at all processes, and (c) the order between strong operations in \(\rho\). (As formalized in Definition 5.2.) Without loss of generality, in \(\rho\), \([\text{SWrite}_1(x, 1), \text{SAck}_1(x)]\) precedes \([\text{SWrite}_2(y, 1), \text{SAck}_2(y)]\).

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

Note that this result is proved w.r.t. Definition 5.2. Since Definition 5.1 is strictly stronger than Definition 5.2, the lower bound is valid for both definitions. Note also that if weak
operations are not allowed to access the same objects as strong operations, then by using similar arguments, it can be shown that $|\text{read}| + |\text{sop}| + |\text{write}| \geq d$, where sop is any type of strong operation.

The same arguments as in the proof of Theorem 7.1, with the roles of reads and writes reversed, prove:

**Theorem 7.2** For any implementation of hybrid consistency, $|\text{write}| + |\text{read}| \geq d$.

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 7.3** For any fast read implementation of hybrid consistency, $|\text{write}| \geq d/2$.

**Corollary 7.4** For any fast write implementation of hybrid consistency, $|\text{read}| \geq d/2$.

### 7.2 Read-Modify-Write

**Theorem 7.5** For any implementation of weak consistency, $|\text{RMW}| \geq d$.

**Proof:** Assume, by way of contradiction, that there is an implementation of weak consistency for which $|\text{RMW}| < d$. Let $p_1$ and $p_2$ be two processes that access an object $x$, and let $f_1$ and $f_2$ be two functions such that $f_1(0) \neq 0$ and $f_2(0) \neq 0$. Without loss of generality, assume that $x$ is initially 0.

By the specification of $x$, there exists an execution $\sigma_1$ such that $\text{ops}(\sigma_1)$ is $[\text{RMW}_1(x, f_1), \text{Return}\&\text{Ack}_1(x, 0)]$ and $\text{RMW}_1(x, f_1)$ occurs at real time 0. 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 $\sigma_1$.

In a similar manner, by the specification of $x$, there exists an execution $\sigma_2$ such that $\text{ops}(\sigma_2)$ is $[\text{RMW}_2(x, f_2), \text{Return}\&\text{Ack}_2(x, 0)]$ and $\text{RMW}_2(x, f_2)$ occurs at real time 0. The delay of
all messages in \( \sigma_2 \) is exactly \( d \). By assumption, the real time at the end of \( \sigma_2 \) is less than \( d \). Hence, no message is received at any node during \( \sigma_2 \).

Since no message is ever received in \( \sigma_1 \) and \( \sigma_2 \), the partial execution obtained from \( \sigma_1 \) by replacing \( p_1 \)'s history with \( p_2 \)'s history in \( \sigma_2 \) can be extended to a weakly consistent execution \( \sigma \), having no other accesses to \( x \). Then \( \text{ops}(\sigma) \) consists of the operation \([\text{RMW}_1(x, f_1), \text{Return\&Ack}_1(x, 0)]\) and \([\text{RMW}_2(x, f_2), \text{Return\&Ack}_2(x, 0)]\).

Since \( \sigma \) is weakly consistent, there exists a legal serialization \( \tau \) of all the operations in \( \sigma \). Thus, either \( \text{rmw}_1(x, f_1, 0) \rightarrow \text{rmw}_2(x, f_2, 0) \) or \( \text{rmw}_2(x, f_2, 0) \rightarrow \text{rmw}_1(x, f_1, 0) \). In both cases, since we assumed that \( f_1(0) \neq 0 \) and \( f_2(0) \neq 0 \), \( \tau \) is not legal. A contradiction.

Since all hybrid consistent executions and all sequentially consistent executions are weakly consistent too, the next two theorems follow immediately:

**Theorem 7.6** For any implementation of sequential consistency, \( |\text{RMW}| \geq d \).

**Theorem 7.7** For any implementation of hybrid consistency, \( |\text{RMW}| \geq d \).

### 7.3 Dequeue and Pop

Attiya has shown that the time required to execute a dequeue or a pop operation in any implementation of sequential consistency is at least \( d \) [13]. These results rely on the requirement for a legal serialization of all operations in the definition of sequential consistency, but do not assume anything about the way operations are ordered in this serialization. Essentially, the same proofs apply to weak consistency and hybrid consistency as well. Thus, we have:

**Theorem 7.8** For any weakly consistent implementation of a FIFO queue \( Q \), \( |\text{deq}(Q)| \geq d \).

**Proof:** We use the following notations during the proof: The call events for enqueue and dequeue operations are Enq and Deq, respectively. The response events are Ack and Ret, respectively. A dequeue operation is denoted by \( \text{deq} \) and an enqueue operation by \( \text{enq} \).

Assume by way of contradiction that there exists a weakly consistent implementation of \( Q \) for which \( |\text{deq}(Q)| < d \). Let \( T = |\text{deq}(Q)| \) and let \( p_1 \) and \( p_2 \) be two processors that access
Q. By definition, the queue Q is initially empty. By the specification of Q, there exists an execution \( \sigma'_1 \) such that \( \text{ops}(\sigma'_1) \) is

\[
[\text{Enq}_1(Q, 1), \text{Ack}_1(Q)], [\text{Deq}_2(Q), \text{Ret}_2(Q, v_2^i)], \ldots, [\text{Deq}_2(Q), \text{Ret}_2(Q, v_2^i)], \ldots
\]

\( \text{Enq}_1(Q, 1) \) occurs at real time 0 and \( \text{Ack}_1(Q) \) occurs at time \( t \); \( \text{Deq}_2^1(Q) \) occurs at time \( t \) while \( \text{Deq}_2^j(Q) \) occurs at time \( t + (j - 1)T \). Since \( \sigma'_1 \) is weakly consistent, there exists a legal serialization of all the operations in \( \text{ops}(\sigma'_1) \). Thus, the enqueue by \( p_1 \) must appear in this serialization after a finite number of dequeue operations by \( p_2 \). Thus, there exists an \( i \) such that \( v_2^i = 1 \) and for all \( j, 1 \leq j \leq i, v_2^j = \perp \).

Fix this particular \( i \). Let \( \sigma_1 \) be the partial execution achieved by truncating \( \sigma'_1 \) after \( \text{deq}_2^i \). More precisely, \( \text{ops}(\sigma_1) \) is

\[
[\text{Enq}_1(Q, 1), \text{Ack}_1(Q)], [\text{Deq}_2(Q), \text{Ret}_2(Q, \perp)], \ldots, [\text{Deq}_2^{i-1}(Q), \text{Ret}_2^{i-1}(Q, \perp)], [\text{Deq}_2^i(Q), \text{Ret}_2^i(Q, 1)]
\]

\( \text{Enq}_1(Q, 1) \) occurs at real time 0 and \( \text{Ack}_1(Q) \) occurs at time \( t \); \( \text{Deq}_2^1(Q) \) occurs at time \( t \) while \( \text{Deq}_2^j(Q) \) occurs at time \( t + (i - 1)T \). By assumption, the real time at the end of \( \sigma_1 \) is less than \( t + (i - 1)T + d \). Thus, no message sent after \( t + (i - 1)T \) is received during \( \sigma_1 \).

We now consider the partial execution where \( \text{deq}_2^i \) is replaced with a dequeue operation by \( p_1 \). More precisely, by the specification of Q, there exists a partial execution \( \sigma_2 \) such that \( \text{ops}(\sigma_2) \) is

\[
[\text{Enq}_1(Q, 1), \text{Ack}_1(Q)], [\text{Deq}_2(Q), \text{Ret}_2(Q, \perp)], \ldots, [\text{Deq}_2^{i-1}(Q), \text{Ret}_2^{i-1}(Q, \perp)], [\text{Deq}_1(Q), \text{Ret}_1(Q, u)]
\]

\( \text{Enq}_1(Q, 1) \) occurs at real time 0 and \( \text{Ack}_1(Q) \) occurs at time \( t \); \( \text{Deq}_2^1(Q) \) occurs at time \( t \) while \( \text{Deq}_2^{i-1}(Q) \) occurs at time \( t + (i - 2)T \), and \( \text{Deq}_2^i(Q) \) occurs at time \( t + (i - 1)T \). Since \( \sigma_2 \) is weakly consistent, there exists a legal serialization of \( \text{ops}(\sigma_2) \) in which \( \text{deq}_2^i \) is ordered after \( \text{enq}_1 \). Thus, \( u = 1 \). By assumption, the real time at the end of \( \sigma_2 \) is less than \( t + (i - 1)T + d \). Thus, no message sent after \( t + (i - 1)T \) is received during \( \sigma_2 \).

\(^1\)If we allow queues to be initially non-empty, the proof of the lower bound becomes the same as the proof for read-modify-write operations; we leave the exact details to the interested reader.

88
Since no message sent after time $t + (i - 1)T$ is ever received in $\sigma_1$ and $\sigma_2$, and since $\sigma_1$ and $\sigma_2$ are identical until time $t + (i - 1)T$, the partial execution $\sigma$ obtained from $\sigma_1$ by replacing $p_i$’s history with $p_i$’s history in $\sigma_2$ can be extended to an admissible weakly consistent execution. Then $\text{ops}(\sigma)$ is

$$\text{ops}(\sigma) = \{\text{Enq}_1(Q, 1), \text{Ack}_1(Q), [\text{Deq}_2(Q), \text{Ret}_2(Q, \bot)], \ldots, [\text{Deq}_2(Q), \text{Ret}_2^{-1}(Q, \bot)], [\text{Deq}_1(Q), \text{Ret}_1(Q, 1)], [\text{Deq}_2(Q), \text{Ret}_2(Q, 1)]\}$$

Thus, there exists a legal serialization $\tau$ of the above operations. However, in $\tau$ the element “1” is enqueued once but dequeued twice. A contradiction.

The same proof holds for the following theorem, by replacing each enqueue operation with a push operation and each dequeue operation with a pop operation.

**Theorem 7.9** For any weakly consistent implementation of a stack $S$, $|\text{pop}(S)| \geq d$.

Since every hybrid consistent execution is also weakly consistent, the next two theorems follow immediately:

**Theorem 7.10** For any hybrid consistent implementation of a FIFO queue $Q$, $|\text{deq}(Q)| \geq d$.

**Theorem 7.11** For any hybrid consistent implementation of a stack $S$, $|\text{pop}(S)| \geq d$. 

89
Chapter 8

A Framework for Non-Sequential Executions

In this chapter, we present a framework for defining consistency conditions that allow non-sequential executions. We then demonstrate how this framework can be used to generalize the definitions of three consistency conditions, sequential consistency, weak consistency and hybrid consistency, to allow non-sequential executions. The framework developed in this chapter is also used later in this work to define and investigate alpha consistency.

8.1 Overview

We remind the reader that our original framework assumes a system consisting of a collection of nodes. At each node there is an application program and a memory consistency system (MCS) process.

In order to cope with non-sequential executions, we must extend our concept of the system architecture. In the extended model, an application program contains instructions to access shared memory and conditional branch instructions. Also, for each node, we add an interface called the run-time environment between the application program and the MCS. The run-time environment at each node executes the shared memory instructions by interacting with the local MCS process; its decisions as to which instructions to execute rely on the application program at that node. (We use the term run-time environment to refer to the combination
of the functionality of a compiler, which sees the whole program, and a conventional run-time system, which makes decisions dynamically based on partial knowledge.) An illustration of a node according to the generalized framework is given in Figure 8.1. In this framework, consistency conditions are specified as a guarantee on the run-time environments at all nodes with respect to the program at each node. It is the responsibility of both the MCS process and the run-time environment to provide this guarantee.

A straightforward run-time environment would simply submit operations to the MCS one at a time in the order specified by the program. In this case, the combination of the application program and the run-time environment degenerates to the previous concept of an application program, studied earlier in this thesis. In order to achieve various optimizations however, the run-time environment might submit operations out of order, might have multiple operations pending at a time, and might anticipate branches (sometimes incorrectly). We do not address the specific algorithm used by the run-time environment. (That is another very interesting problem, beyond the scope of this thesis.) Instead, our goal is to model the run-time environment sufficiently abstractly so that any of a large number of specific run-time environments can fit into this framework. Obviously the run-time environment cannot do just anything—the optimizations that it performs should be transparent to the application program. The condition we require of a correct run-time environment is that there exist a way (after the fact) to take a subset of the operations performed by the run-time environments at all the nodes and
order them to be consistent with some "sequential execution" of the programs at all the nodes. (Some of the operations performed by the run-time environment might end up not being used, for instance if they resulted from an incorrect prediction about a branch. These operations can be ignored in determining whether there is a corresponding sequential execution.) We emphasize that the order in which operations appear to execute is what is important, not the order in which they actually execute.

Our framework incorporates rollback and compensating operations in an implicit manner. In particular, we allow the run-time environment to communicate with the MCS in order to perform other operations on the data that are not part of the application programs but that are necessary for achieving the desired consistency condition (for example, operations to restore the state of the shared variables due to incorrect predictions). These operations are ignored when the subset of operations consistent with a sequential execution is taken.

8.2 System Components

An application program consists of a sequence of instructions, each with a unique label. There are two types of instructions, (shared) memory instructions and control instructions. A memory instruction specifies an access to a shared object. The specific kind of access depends on the data type of the object. A control instruction consists of a condition (a boolean function of the process' local state) and a branch (jump to the instruction with the given label).

As before, the memory consistency system (MCS) implements the shared objects that are manipulated by the application programs. It consists of a process at each node as well as possibly other hardware. The interface to the MCS consists of calls (also called invocations) and responses on particular objects. (Objects are defined as before.)

The run-time environment at a node takes as input the application program and executes instructions on the MCS. An operation is a specific instance of an execution of an instruction. A memory operation consists of two parts, a call (to the MCS process) and a matching response (from the MCS process). A control operation consists of an evaluation of its condition. A control operation is represented by the result (true or false) of the evaluation. Thus the run-time environment must keep track of the local state of the application process in order to perform the evaluation. The run-time environment and MCS process may also communicate
concerning issues such as rollback and compensating operations, necessary to implement certain optimizations. This communication can be modeled with calls to and responses from the MCS process.

We slightly generalize the notion of an event, i.e., an event is a call, a response, or a control operation (condition evaluation). An execution (of the system) is a sequence of events such that there is a correspondence between calls and responses (matching object and process) and each response follows its corresponding call.

8.3 Sequential Executions

Although an execution is a sequence of events, it can also be viewed as containing operations. Each control operation is itself an operation. The memory operations in an execution are obtained by matching up corresponding call and response events; we assume that the run-time environment matches the call and response events defining the memory operations. In the basic framework, operations are also obtained by matching up calls and responses. However, in that framework, matching the calls and responses is obvious since operations were issued one after the other and in program order.

We also assume that the run-time environment identifies, possibly after the fact, a subset of all the operations executed. This subset consists of the operations that we want to consider as “really happening”, and there must be an ordering of this subset that satisfies certain properties.

Next, we define the notion of a flow control sequence for a process \( p_i \) and the notion of a sequential execution. We build up inductively an execution of \( p_i \)'s program in which every instruction finishes executing before the next one begins. Given process \( p_i \)'s program, a flow control sequence, \( fcs_i \), is a sequence of operations defined as follows. The first element of \( fcs_i \) is an instance of the first instruction in \( p_i \)'s program. Suppose the \( k \)-th element of \( fcs_i \), denoted \( op_k \), is an instance of instruction \( I \) in the program. If \( op_k \) is a control operation and its condition evaluates to true, then the \((k + 1)\)-st element of \( fcs_i \) is an instance of the instruction whose label is the branch of instruction \( I \). Otherwise the \((k + 1)\)-st element of \( fcs_i \) is an instance of the instruction immediately after \( I \) in the program. A flow control sequence can either be finite or infinite (for non-terminating programs). Note that the flow control sequence implies a total order on the operations appearing in it; we denote this order by \( \Rightarrow \).
Let $fcs_i$ be a flow control sequence for $p_i$. A sequence $\tau$ of memory operations is \textit{fully $fcs_i$-admissible} if $\tau[j]$ is equal to the subsequence of $fcs_i$ consisting exactly of the memory operations. Intuitively, this implies that the ordering of operations by $p_i$ in $\tau$ agrees with some flow control sequence $fcs_i$ for $p_i$, and does not end unless the program terminates. A sequence $\tau$ of memory operations is \textit{partially $fcs_i$-admissible} if $\tau[j]$ is a prefix of the subsequence of memory operations in $fcs_i$. Intuitively, this implies that, so far, the ordering implied by the flow control sequence is obeyed by $\tau$, but it is not necessarily completed yet.

A sequence $\tau$ of memory operations is \textit{fully} (resp., \textit{partially}) \textit{admissible} with respect to a set of flow control sequences $\{fcs_i\}_{i=1}^n$, one for each $p_i$, if it is fully (resp., partially) $fcs_i$-admissible for all $i$.

A sequence of memory operations is a \textit{sequential execution} if it is legal and fully admissible (with respect to some set of flow control sequences). Hence, a sequential execution is a possible execution of a program on a hardware that can execute only a single operation of a single process at any given time (e.g., a time sharing machine). In particular, every sequential execution is sequentially consistent.

The next claim establishes the relation between partial admissible sequences of memory operations and sequential executions. This relation is important for the next two chapters where we investigate programming techniques for hybrid consistency and alpha consistency.

\textbf{Claim 8.1} Any legal partially admissible sequence of memory operations is a prefix of a sequential execution and vice versa.

When defining and analyzing consistency conditions, it is important to take into account the effects of control operations. A specific example of the pitfalls associated with failing to do so appears in Section 8.4, when we present our extended definitions of consistency conditions. To capture the effect of control operations, we define a partial order on operations in a flow control sequence; this partial order is featured in the consistency conditions presented below. Based on the relation $fcs_i$ we define a partial order $\leq_{cfs}$ called the \textit{control order}. Formally, for any two memory operations $op_i^1$ and $op_i^2$, $op_k^1 \leq_{cfs} op_i^2$ if there exists a control operation $op_k^3$ such that $op_i^1 \leq_{fcs} op_k^3 \leq_{cfs} op_i^2$.
8.4 Definitions of Non-Sequential Consistency Conditions

In this section, we demonstrate how to extend the definition of three consistency conditions, sequential consistency, weak consistency and hybrid consistency, for the non-sequential case using the framework developed earlier in this chapter. The reason they are extensions is that in non-optimized systems, where operations at each process are invoked in program order and only one operation may be pending at a time, \( fcs_i \) is simply the sequence of operations in the order they were invoked. In this case, these definitions are equivalent to the definitions that were given using the basic framework.

**Definition 8.1 (Sequential consistency)** An execution \( \sigma \) is sequentially consistent if there exists a subset \( S \) of the memory operations in \( \sigma \), a set \( \{ fcs_i \}_{i=1}^n \) of flow control sequences, and a legal permutation \( \tau \) of \( S \) such that \( \tau \) is fully admissible with respect to \( \{ fcs_i \}_{i=1}^n \).

**Definition 8.2 (Weak consistency)** An execution \( \sigma \) is weakly consistent if there exist a subset \( S \) of the memory operations in \( \sigma \) and a set of flow control sequences \( \{ fcs_i \}_{i=1}^n \), one for each \( p_i \), such that for each \( p_i \), there exists a legal permutation \( \tau_i \) of \( S \) which is fully \( fcs_i \)-admissible.

Here, we only extend the definition of hybrid consistency based on sequential consistency (Definition 5.2). Note that linearizability requires that the order in which all processes view operations reflects the real time of their execution. However, in non-sequential executions, operations of the same process are sometimes executed in real time in a different order than the order they appear in the program (the flow control sequence). Hence, there is no point in talking about linearizable executions in the context of non-sequential executions.

**Definition 8.3 (Hybrid consistency)** An execution \( \sigma \) is hybrid consistent if there exist a subset \( S \) of the memory operations in \( \sigma \), a set of flow control sequences \( \{ fcs_i \}_{i=1}^n \), and a permutation \( \rho \) of the strong operations in \( S \) such that for each process \( p_i \), there exists a legal permutation \( \tau_i \) of \( S \) with the following properties:

1. \( \tau_i \) is fully \( fcs_i \)-admissible.

2. If \( op_j^1 \xrightarrow{fcs_i} op_j^2 \), then \( op_j^1 \xrightarrow{\tau_i} op_j^2 \), for any \( j \).
3. If $\text{op}_j^1 \xrightarrow{f_c \land} \text{op}_j^2$ and at least one is strong, then $\text{op}_j^1 \xrightarrow{\tau_j} \text{op}_j^2$, for any $j$.

4. If $\text{op}_j^1 \xrightarrow{\theta} \text{op}_j^2$ (implying both are strong), then $\text{op}_j^1 \xrightarrow{\tau_j} \text{op}_j^2$, for any $j$ and $k$.

5. If $\text{op}_j^1 \xrightarrow{f_c \land} \text{op}_j^2$ and $\text{op}_j^1$ and $\text{op}_j^2$ access the same location, then $\text{op}_j^1 \xrightarrow{\tau_j} \text{op}_j^2$, for any $j$.

We now discuss the last property in the definition of hybrid consistency (Definition 8.3) in more detail. It states that all operations on the same object by the same process $p_j$ are viewed by every other process in the same order as they are viewed by $p_j$. This property does not appear in the definitions of hybrid consistency in Chapter 5 (Definitions 5.1 and 5.2). However, it is necessary in order for some of our results to hold, as is shown in Section 9.3.4. Evidence suggests it is a reasonable assumption, since some previous authors make the even stronger assumption that all processes view all operations on the same object, no matter which process invoked them, in the same order.\footnote{In [3, 34, 35], a total order on all the writes to the same location is assumed. In addition, it is assumed that a value read from a specific location can be uniquely identified with a write operation. Thus, all the processes view all the writes to the same location in the same order. Since a value read from a specific location can be uniquely identified with a write operation, each read is viewed by all the processes to be between the same writes to that location. These two assumptions imply that all the processes view all the operations on the same object in the same order.}

To illustrate the problem that occurs if hybrid consistency were defined without considering control operations, consider the example in Figure 8.2, assuming $x$ and $y$ are initially 0 and all instructions are weak.

In this program, the values returned by the reads affect the decision whether to invoke the writes and the invocations of the writes affect the possible values returned by the reads. For example, assume this program is being executed on a hybrid consistent memory, and all
operations are weak. If we ignore the existence of control instructions, the definition would allow the following execution $\sigma$ that appears in Figure 8.3.

In $\sigma$, the program for $p_1$ allows the read($x$) to return 5, whereas $x$ could have taken the value 5 only if $p_1$ writes 5 to $y$, which will happen only if $p_1$ reads 5 in $x$. It is important to eliminate such executions, with circular inference relations where the prediction about the result of a control operation could affect its actual result. If such behavior is allowed, then writing programs and arguing about them is much more complicated.

As will be defined in Chapter 9, this program is data-race free. If the second condition is omitted from the definition of hybrid consistency (Definition 8.3), then the execution $\sigma$ described in Figure 8.3 is hybrid consistent. However, it is not sequentially consistent. In Chapter 9, we define the notion of data-race free programs and show that every hybrid consistent execution of a data-race free program is sequentially consistent (Theorem 9.12). Hence, this result is not valid if we omit the requirement that the sequences $\{r_i\}_{i=1}^n$ are consistent with the control order.

8.4.1 Discussion of the Framework

Viewing Order vs. Execution Order: We would like to emphasize that the order in which operations may be viewed by different processes need not reflect the order in which they are executed. For example, consider the program in Figure 1.1 that was used in Chapter 1 to demonstrate the performance gains of non-sequential executions. Due to the control order, every process must view the write to $z$ after the read from $x$. On the other hand, a sophisticated run-time environment (or compiler) could detect that the write to $z$ should be invoked regardless of the outcome of the read from $x$. Hence, the run-time environment could invoke the write to $z$ before the read from $x$. This is allowed by the definition of hybrid consistency since it is possible to order the write to $z$ after the read from $x$ (even if they were executed in the reversed order).
Limitations of the Framework: Our frameworks for defining consistency conditions (both the basic framework and the extended framework) have one important limitation: they do not support indirect addressing. For example, the program in Figure 8.4 is not defined in our framework and nothing can be claimed about its possible executions. (We use & and * for indirect addressing in the style of C/C++.) This limitation is inherent in the way we define objects and legal sequences. At the moment, we are not aware of a simple way to overcome this problem.
Chapter 9

Programming Methods for Hybrid Consistency

In this chapter we develop several programming techniques that allow a programmer to write programs assuming sequential consistency and run them on hybrid consistent hardware. The first approach shows how to statically label each operation as either weak or strong. The second approach is to protect shared data within critical sections. For this, we show how to transform any non-cooperative solution to the mutual exclusion problem which is based on sequential consistency into a solution based on hybrid consistency. The third approach is based on avoiding data races, i.e., it requires to label enough operations as strong so there will be no data races in the program.

Remark: Throughout this chapter, we use Definition 8.3 of hybrid consistency.

9.1 Static Approaches

In this section we present techniques for writing programs for hybrid consistent shared memories that are based on statically classifying accesses according to their type (read or write) and the objects they access. In Section 9.1.1, we show that every hybrid consistent execution of a program in which all writes are strong is sequentially consistent. This result is used to develop efficient synchronization code in Section 9.2. We then show symmetrically, in Section 9.1.2,
that every hybrid consistent execution of a program in which all reads are strong is sequentially consistent. In order to prove this theorem, we make further assumptions on the execution and demonstrate their necessity.

9.1.1 Strong Writes

We prove the following theorem:

**Theorem 9.1** Every hybrid consistent execution of a program in which all writes are strong and all reads are weak is sequentially consistent.

**Proof:** Let \( \sigma \) be such an execution. Let \( S \) be a subset of memory operations in \( \sigma \), \( fcs_i \) for each \( p_i \) be a flow control sequence, \( \rho \) be a permutation of the strong operations (namely, the writes) in \( S \), and \( \tau_i \) for each \( p_i \) be a legal permutation of \( S \) as guaranteed by the definition of hybrid consistency. We will insert the (weak) read operations into \( \rho \) to construct \( \tau \), a legal permutation of \( S \) that is fully admissible with respect to the \( fcs_i \)'s.

A read by process \( p_i \) is inserted after any write that precedes it in \( \tau_i \) and before any write that follows it in \( \tau_i \). This can be done since \( \tau_i \) agrees with \( \rho \) on the order of strong operations. Furthermore, it will be inserted after any read by \( p_i \) that precedes it in \( \tau_i \) and before any read by \( p_i \) that follows it in \( \tau_i \). The ordering of read operations by different processes is unimportant.

Clearly, \( \tau \) is a permutation of \( S \). Since \( \tau_i \) preserves the ordering of operations by \( p_i \), it follows that \( \tau \) is \( fcs_i \)-admissible for all \( i \).

The fact that \( \tau \) is legal follows from the fact that \( \tau_i \) is legal, that \( \tau_i \) agrees with \( \rho \) on the order of strong operations and from the construction of \( \tau \).

9.1.2 Strong Reads

We now prove that labeling all reads as strong suffices for providing sequential consistency. The proof of this theorem relies on the following assumptions about the execution: (a) every value written to the same object is unique, and (b) every value written is returned by some read. We show by specific counter-examples that both these assumptions are necessary to prove the above result.
Assumptions (a) and (b) make the result of this section impractical. However, the actual importance of this result is in the counter-examples themselves. These examples show that in frameworks for defining consistency conditions where claims can only be made about the way operations are viewed, e.g., [11, 37, 48], these assumptions can affect the validity of certain properties. In particular, when arguing about properties of consistency conditions, it cannot be claimed that such assumptions are only technical, made in favor of clarity. Whenever one of these assumptions is made, it must be shown explicitly that it does not affect the correctness of the results.

**Theorem 9.2** Every hybrid consistent execution of a program in which all reads are strong and all writes are weak is sequentially consistent.

**Proof:** Let \( \sigma \) be such an execution. Let \( S \) be a subset of memory operations in \( \sigma \), \( fcs_i \) for each \( p_i \) be a flow control sequence, \( \rho \) be a permutation of the strong operations (namely, the reads) in \( S \), and \( \tau_i \) for each \( p_i \) be a legal permutation of \( S \) as guaranteed by the definition of hybrid consistent. We will construct \( \tau \), a legal permutation of \( S \) that is fully admissible with respect to the \( fcs_i \)’s.

The method for constructing \( \tau \) is to place the reads as in \( \rho \), and to place each write by \( p_i \) in the interval (between reads) where it appears in \( \tau_i \). All the writes that end up in the same interval can be ordered in any way, as long as the \( fcs_i \) orders are preserved.

The fact that \( \tau \) respects the \( fcs_i \)'s follows from the construction of \( \tau \) and the fact that \( \tau_i \) respects \( p_i \)'s ordering.

We now show that \( \tau \) is legal. Assume, by way of contradiction, that \( \tau \) is not legal. So there is some read of \( x \) (by \( p_i \)) that returns \( v \), where the most recent write to \( x \) (by \( p_j \)) was for \( w \neq v \).

\[
\tau = \ldots w_j(x, w) \ldots \text{(no write to } x) \ldots r_i(x, v) \ldots
\]

By construction of \( \tau \), \( w_j(x, w) \) comes before \( r_i(x, v) \) in \( \tau_j \). Since \( \tau_j \) is legal, some \( w_k(x, v) \) comes between \( w_j(x, w) \) and \( r_i(x, v) \) in \( \tau_j \). Since every value written is read (Assumption (b)) and \( \tau_j \) is legal, some \( r_m(x, w) \) comes between \( w_j(x, w) \) and \( w_k(x, v) \) in \( \tau_j \), i.e.,

\[
\tau_j = \ldots w_j(x, w) \ldots r_m(x, w) \ldots w_k(x, v) \ldots r_i(x, v) \ldots
\]
Figure 9.1: A possible execution $\sigma$ without the unique writes assumption

By construction of $\tau$, $w_j(x, w)$ comes before $r_m(x, w)$ in $\tau$. By hybrid consistency, $r_m(x, w)$ comes before $r_i(x, v)$ in $\tau$ since the reads are strong, i.e.,

$$\tau = \ldots w_j(x, w) \ldots r_m(x, w) \ldots r_i(x, v) \ldots$$

By hybrid consistency, $r_m(x, w)$ also comes before $r_i(x, v)$ in $\tau_k$. Since $\tau_k$ is legal and every write is unique (Assumption (a)), we also get:

$$\tau_k = \ldots w_j(x, w) \ldots r_m(x, w) \ldots w_k(x, v) \ldots r_i(x, v) \ldots$$

Thus by construction of $\tau$, $w_k(x, v)$ comes between $r_m(x, w)$ and $r_i(x, v)$ in $\tau$, i.e.,

$$\tau = \ldots w_j(x, w) \ldots r_m(x, w) \ldots w_k(x, v) \ldots r_i(x, v) \ldots$$

A contradiction.

We now give two counter-examples to show that the assumptions made in order to prove Theorem 9.2 are necessary. The first example demonstrates that the theorem is incorrect without the assumption that the values written to the same location are unique. The second example demonstrates that the theorem is incorrect without the assumption that every write is read.

**Counter-example 1:** For the first counter-example, consider the execution $\sigma$ in Figure 9.1, assuming the initial value of every object is 0. To show $\sigma$ is hybrid consistent, we define

$$\tau_1 = w_1^2(x, 2), sr_3^1(y, 0), w_3^1(y, 1), sr_3^2(y, 1), w_1^1(y, 1), sr_3^2(y, 1), w_1^2(x, 2), sr_3^2(x, 2)$$
$$\tau_2 = w_1^2(x, 2), sr_3^1(x, 2), sr_3^2(y, 0), w_3^1(y, 1), sr_3^2(y, 1), w_1^1(y, 1), sr_3^2(y, 1), w_1^2(x, 2), sr_3^2(x, 2)$$
$$\tau_3 = w_1^2(x, 2), sr_3^1(x, 2), sr_3^2(y, 0), w_3^1(y, 1), sr_3^2(y, 1), w_1^1(y, 1), sr_3^2(y, 1), w_1^2(x, 2), sr_3^2(x, 2)$$
Figure 9.2: A possible execution σ if not every write is read

σ is a hybrid consistent execution in which every write is read, but the values which are written are not unique. However, σ is not a sequentially consistent execution since \( sr_1^2(x, 2) \) means that either \( p_1 \) or \( p_2 \) has already written to both \( y \) and \( x \), and it’s not possible that \( \tau_2^3(y, 0) \) returns the initial value of \( y \). Therefore, the assumption that the values which are being written are unique is necessary for the proof of Theorem 9.2.

**Counter-example 2:** For the second counter-example, consider the execution σ in Figure 9.2, assuming the initial value of every object is 0. To show σ is hybrid consistent, we define

\[
\begin{align*}
\tau_1 &= w_1^2(x, 2), w_1^1(x, 1), w_1^2(y, 1), w_2^1(y, 2), sr_3^1(y, 2), sr_3^2(x, 1) \\
\tau_2 &= w_1^2(y, 1), w_2^1(y, 2), w_2^2(x, 2), w_1^1(x, 1), sr_3^2(y, 2), sr_3^2(x, 1)
\end{align*}
\]

σ is a hybrid consistent execution in which the values which are written are unique, but not every write is read. However, the reader can verify that σ is not sequentially consistent. Therefore, the assumption that each write is read is necessary for the proof of Theorem 9.2.

### 9.2 Programming with Critical Sections

Given a non-cooperative mutual exclusion algorithm designed for sequentially consistent memories, we produce a modified algorithm as follow: We add one strong write (to a location which is never read from) to the entry section and label all the writes in the synchronization part of the code as strong, while we label all other operations as weak. We prove that the modified algorithm guarantees logical mutual exclusion based on hybrid consistency.

Aside from the restrictions that are imposed by the assumption that the algorithm is non-cooperative (see Definition 4.2), we assume that the algorithm is designed to run with any
code in the critical section and in the remainder section, and that it can run on asynchronous systems. We start by adjusting the definition of logical mutual exclusion to the extended framework. The only difference between the following definition and the definition for the basic framework (Section 4.2) is that the $CS_i^k$'s are defined w.r.t. the flow control sequences rather than w.r.t. the execution itself.

Specifically, given a mutual exclusion algorithm (program) $A$, consider a flow control sequence for process $p_i$ and and let $CS_i^k$ be the set of operations invoked by process $p_i$ during the $k$th time that $p_i$ executes the critical section in this sequence. $A$ guarantees \textit{logical mutual exclusion based on a consistency condition} $C$ provided that the following holds. Let $\sigma$ be an execution of $A$ that is allowed by $C$ and let $\{\tau_r\}_{r=1}^n$ be a set of sequences as required in the definition of $C$ (the views of the different processes). Consider the $CS_i^k$'s induced by $f_{cs}$.

**Logical mutual exclusion:** For any four operations $op_i^1, op_i^2 \in CS_i^k$ and $op_j^1, op_j^2 \in CS_j^l$, $op_i^1 \xrightarrow{\tau_r} op_i^1$ if and only if $op_j^2 \xrightarrow{\tau_r} op_j^2$ for all processes $p_i$ and $p_j$. As before, this implies that there is a total order on all critical section executions; furthermore, this ordering is the same in all $\tau_r$'s.

**Deadlock freedom:** For every process $p_q$, if $\tau_q$ is infinite and there exists a process $p_i$ that is in its entry section from some point on in $\tau_q$, then there is another process $p_j$ that enters (and leaves) its critical section infinitely often in $\tau_q$.

\textbf{Claim 9.3} \textit{There is at least one write to a variable in $exc(A)$ in the exit section of every non-cooperative mutual exclusion algorithm based on sequential consistency.}

\textbf{Proof:} Assume, by way of contradiction, that there exists a non-cooperative mutual exclusion algorithm $A$ such that there is no write to any of the variables of $exc(A)$ in the exit section of $A$. Assume that we run the algorithm with the following critical section (for every process $p_i$), under the assumption that $x$ is initially 0:

\begin{align*}
tmp_{pi} &= r(x) \\
tmp_{pi} &:= tmp_{pi} + 1 \\
w(x, tmp_{pi})
\end{align*}
Since the algorithm is non-cooperative, $x$ is not accessed in the entry and exit sections of $A$. Consider now a sequentially consistent execution $\sigma$ of $A$ with this critical section in which $p_0$ starts at time 0 and runs until it enters the critical section, completes it, and exits; then $p_1$ starts and runs until it enters the critical section, completes it, and exits. Denote by $t$ the time at which $p_0$ completes the read operation from $x$.

We claim that there is a sequentially consistent execution $\sigma'$ of $A$ with this critical section in which $p_0$ starts at time 0 and runs until time $t$, when it completes the read operation; then $p_1$ starts and runs until it enters the critical section, completes it, and exits; then $p_0$ completes its critical section and exits. Note that there is no write in the exit section of $A$ and there is no write to variables in $\text{exc}(A)$ in the critical section of $p_0$. Thus, all the values read by $p_1$ in its entry section in $\sigma'$ are equal to the values it reads in its entry section in $\sigma$; since in $\sigma$, $p_1$ must eventually enter the critical section (to avoid deadlock) it must eventually enter the critical section in $\sigma'$.

Clearly, $\sigma'$ violates the definition of physical mutual exclusion, since $p_0$ and $p_1$ are inside the critical section at the same time. Furthermore, note that in $\sigma'$ both $p_0$ and $p_1$ read the value 0 from $x$ and thus the final value of $x$ at the end of $\sigma'$ is 1. This value could not happen in any execution that preserves logical mutual exclusion.

Given a non-cooperative algorithm $A$ for the mutual exclusion problem based on sequential consistency, label every write in the entry and exit sections as strong and every read in the entry and exit sections as weak; operations inside the critical section or remainder section are labeled as weak. Next, add a strong write to some object $\text{nac}(A)$ that is not accessed elsewhere in the program such that this write will be the last instruction executed (in a flow control sequence) before each critical section. Call the modified algorithm $A'$.

In order to prove that $A'$ guarantees mutual exclusion based on hybrid consistency, we have to show that it is free of deadlocks and that it guarantees logical mutual exclusion.

**Lemma 9.4** $A'$ guarantees deadlock freedom based on hybrid consistency.

**Proof:** Assume, by way of contradiction, that there exists a hybrid consistent execution $\sigma$ of $A'$ that has a deadlock. Denote by $\{\tau_i^{p_i}\}_{i=1}^{n}$ the set of sequences of operations as guaranteed by the definition of hybrid consistency. Let $\sigma'$ be the execution that results by eliminating from $\sigma$
all operations that are not invoked inside (w.r.t. the flow control sequences) the entry or exit section of \( A' \). Since the algorithm is non-cooperative, there are no writes to variables in \( exc(A') \) outside (w.r.t. the flow control sequences) the entry and exit sections of \( A' \). Thus, \( \sigma' \) includes all the writes (in \( \sigma \)) to variables in \( exc(A') \), and thus, \( \sigma' \) is hybrid consistent. Specifically, define for each \( j \) a flow control sequence \( fcs'_j \) by eliminating from \( fcs_j \) all the operations of \( p_j \) that do not appear in \( \sigma' \). Thus, the set of sequences \( \{ \tau'_j \}_{j=1}^n \) that results from eliminating all the operations that do not appear in \( \sigma' \) from \( \{ \tau_j \}_{j=1}^n \) obeys all the requirements in the definition of hybrid consistency (w.r.t. the \( \{ fcs'_j \}_{j=1}^n \)). Note that all the writes in \( \sigma' \) are strong, and thus, by Theorem 9.1, \( \sigma' \) is sequentially consistent. Furthermore, in \( \sigma' \) there is a deadlock. Since there are no reads from \( nac(A) \), the execution \( \sigma'' \) that results from eliminating all the writes to \( nac(A) \) is also sequentially consistent and there is a deadlock in \( \sigma'' \). Note that \( \sigma'' \) is an execution of \( A \) in the case of an empty critical section. Thus, there is a sequentially consistent execution of \( A \) that has a deadlock. This is a contradiction. 

**Lemma 9.5** \( A' \) guarantees logical mutual exclusion based on hybrid consistency.

**Proof:** Recall that a sequential execution is a sequence of operations. Hence, each sequential execution of a program can be viewed by itself as a sequence \( \tau \) of operations that obeys the requirements in the definition of sequential consistency. Therefore, since \( A \) guarantees mutual exclusion based on sequential consistency, every sequential execution of \( A \) must guarantee logical mutual exclusion as defined above. We will show that if \( A' \) does not guarantee mutual exclusion based on hybrid consistency, we may build a sequential execution of \( A \) in which mutual exclusion is violated.

Let \( \sigma \) be a hybrid consistent execution of an instance of \( A' \) in which every execution of the critical section consists of two writes, and let \( \{ \tau_j \}_{j=1}^n \) be the set of sequences of operations as guaranteed in the definition of hybrid consistency. Call the last operation executed by a process before entering the critical section the *entry point*. Note that the entry point is a write operation on \( nac(A) \). By Claim 9.3, there is at least one write in the exit section. Call the first write in the exit section the *exit point*. Each pair of matching entry and exit points is called a *critical pair*. We now show that in each \( \tau_i \) all the critical pairs are ordered in a non-overlapping way, which gives the result for critical section executions.

Assume, by way of contradiction, that there exists a sequence \( \tau_i \) in which two critical pairs overlap. Assume that \( [op_{i_k}^1, op_{i_k}^2] \) and \( [op_{i_i}^1, op_{i_i}^2] \) are critical pairs that overlap and assume, without
loss of generality, that \( op^2_k \xrightarrow{\tau_j} op^2_i \). Let \( \sigma' \) be the execution that results by eliminating from \( \sigma \) all operations that are not invoked inside entry and exit sections of \( A' \). Since the algorithm is non-cooperative, \( \sigma' \) includes all the writes (in \( \sigma \)) to variables in \( exc(A') \), and thus, \( \sigma' \) is hybrid consistent. By Theorem 9.1, \( \sigma' \) is sequentially consistent. Thus, we may build a legal sequence of operations \( \tau \) in the same way as in the proof of Theorem 9.1. That is, the strong writes are ordered in \( \tau \) in the order they appear in any sequence \( \tau_i \). A read by process \( p_i \) is inserted after any write that precedes it in \( \tau_i \) and before any write that follows it in \( \tau_i \). Furthermore, it will be inserted after any read by \( p_i \) that precedes it in \( \tau_i \) and before any read by \( p_i \) that follows it in \( \tau_i \). The ordering of read operations by different processes in unimportant. Note that \( \tau \) is a sequential execution of \( A' \) in the case of an empty critical section.

By the construction of \( \tau \), and since we assumed that \([op^1_k, op^2_k] \) and \([op^1_i, op^2_i] \) are overlapping in \( \tau_j \), the latest of \( op^2_k \) and \( op^1_i \) precedes both \( op^2_k \) and \( op^2_i \) in \( \tau \). Consider the prefix \( \tau' \) of \( \tau \) that ends with the latest of \( op^2_k \) and \( op^1_i \). Note that any operation in \( \tau' \) that is invoked during the exit sections that correspond to \([op^1_k, op^2_k] \) and \([op^1_i, op^2_i] \) is a read. Remember that there are no reads from \( nac(A) \). Thus, \( \tau'' \), the result of eliminating from \( \tau' \) all reads that are invoked during the exit sections that corresponds to \([op^1_k, op^2_k] \) and \([op^1_i, op^2_i] \) and all writes to \( nac(A) \), is a prefix of a sequential execution of \( A \). Denote the writes in the critical section that corresponds to \([op^1_k, op^2_k] \) by \( w^1_k \) and \( w^2_k \) and the writes in the critical section that corresponds to \([op^1_i, op^2_i] \) by \( w^1_i \) and \( w^2_i \). Add \( w^1_k, w^2_k, w^1_i, w^2_i \) in this order to \( \tau'' \), to form \( \tau''' \). Since all operations that were added to \( \tau'' \) are writes, \( \tau''' \) is legal and is therefore a prefix of a sequential execution. Moreover, \( w^1_k \xrightarrow{\tau'''} w^1_i \) but \( w^2_k \xrightarrow{\tau'''} w^2_i \), which is a violation of logical mutual exclusion. Thus, \( \tau''' \) can be extended to a sequential execution of \( A \) that violates mutual exclusion. This is a contradiction to the assumption that every sequential execution of \( A \) guarantees logical mutual exclusion.

Thus, we have shown that all critical pairs appear in every sequence \( \tau_j \) in a non-overlapping manner. Since the exit points are strong writes, all the processes agree (in the \( \tau_j \)'s) on the same non-overlapping order for the critical pairs. Hence, we only need to show that the order in which critical pairs are ordered in every sequence \( \tau_j \) is unique.

Let \( cp, cp' \) and \( cp'' \) be three critical pairs such that \( cp \) is ordered before \( cp' \) and after \( cp'' \) in any \( \tau \). Since the exit point of \( cp \) is a strong operation, and since it is ordered before the entry point of \( cp' \), all the operations of \( cp \) are ordered before the entry point of \( cp' \). Since the entry point of \( cp \) is a strong operation, and since it is ordered after the exit point of \( cp'' \), all
Entry: wait until read(\text{turn}[i]);

\text{critical section}

Exit: write(\text{turn}[i],false);
write(\text{turn}[i+1],true);

Remainder:
every so often do
if read(\text{turn}[i]) then
write(\text{turn}[i-1],false);
write(\text{turn}[i],false);
write(\text{turn}[i+1],true);
endif
enddo;

Figure 9.3: A cooperative solution for the mutual exclusion problem – code for \textit{p}_i

the operations of \textit{cp} are ordered after \textit{cp}'\textsuperscript{'}\textsuperscript{a}. Thus, (logical) mutual exclusion is also guaranteed by the algorithm.

Note that in cooperative algorithms, a process that participates in the mutual exclusion protocol, but does not wish to enter the critical section, can access variables in the exclusion set while executing the remainder section. If these accesses are labeled as weak, the solution might not be correct anymore.

For example, consider the cooperative solution in Figure 9.3. It can be seen that the algorithm solves mutual exclusion in any sequentially consistent execution. Essentially, the variable \textit{turn} represents a token that circulates around (it is always located at the process \textit{p}_i with \text{turn}[i]=true). The write(\text{turn}[i-1],false) instruction in the remainder section is not necessary under sequential consistency. However, it does not affect correctness to add it, and it is necessary for the counter-example.

We now describe a hybrid consistent execution for three processes, \textit{p}_0, \textit{p}_1 and \textit{p}_2, in which the token is “lost” and a deadlock is created.\textsuperscript{1} In this execution, \text{turn}[0] is initially true, \text{turn}[1]...
and $turn[2]$ are initially false and $p_0$’s operations, in the flow control sequence, are:

$$
\begin{align*}
& r_1^1(turn[0],true) \ w_0^2(turn[2],false) \ w_0^2(turn[0],false) \ w_0^3(turn[1],true) \ r_0^2(turn[0],false) \\
& \end{align*}
$$

The first four operations are from $p_0$’s remainder section; after once transferring the token to $p_1$, $p_0$ reads false from $turn[0]$ in its entry section forever (indicated by the ellipsis). In this execution, $p_1$’s operations, in the flow control sequence, are

$$
\begin{align*}
& r_1^1(turn[1],true) \ w_1^1(turn[0],false) \ w_1^2(turn[1],false) \ w_1^3(turn[2],true) \ r_1^1(turn[1],false) \\
& \end{align*}
$$

These operations are from $p_1$’s remainder section; after once transferring the token to $p_2$, $p_1$ continuously reads false from its turn variable (indicated by the ellipsis). In this execution, $p_2$’s operations, in the flow control sequence, are

$$
\begin{align*}
& r_2^1(turn[2],true) \ w_2^2(turn[1],false) \ w_2^3(turn[2],false) \ w_2^3(turn[0],true) \ r_2^1(turn[2],false) \\
& \end{align*}
$$

These operations are from $p_2$’s remainder section; after once transferring the token to $p_0$, $p_2$ continuously reads false from its turn variable (indicated by the ellipsis).

Recall that all operations under consideration are weak. The following $\tau_i$’s demonstrate that this execution is hybrid consistent (the ellipsis indicate the continuous reads of false from the $turn$ variables):

$$
\begin{align*}
\tau_0 &= r_0^1(turn[0],true) \ w_0^2(turn[2],false) \ w_0^3(turn[0],false) \ w_0^3(turn[1],true) \\
& \quad \ r_1^1(turn[1],true) \ w_1^1(turn[2],true) \ r_2^1(turn[2],true) \ w_2^3(turn[0],true) \\
& \quad \ w_1^1(turn[0],false) \ w_1^2(turn[1],false) \ w_1^3(turn[1],false) \ w_2^3(turn[2],false) \\
\end{align*}
$$

$$
\begin{align*}
\tau_1 &= r_0^1(turn[0],true) \ w_0^3(turn[1],true) \ r_1^1(turn[1],true) \ w_2^3(turn[0],false) \\
& \quad \ w_1^2(turn[1],false) \ w_1^3(turn[2],true) \ r_1^1(turn[2],true) \ w_2^3(turn[0],true) \\
& \quad \ w_2^3(turn[2],false) \ w_0^3(turn[0],false) \ w_2^3(turn[1],false) \ w_2^3(turn[2],false) \\
\end{align*}
$$

$$
\begin{align*}
\tau_2 &= r_0^1(turn[0],true) \ w_0^3(turn[1],true) \ r_1^1(turn[1],true) \ w_1^3(turn[2],true) \\
& \quad \ r_2^1(turn[2],true) \ w_2^3(turn[1],false) \ w_2^3(turn[2],false) \ w_2^3(turn[0],true) \\
\end{align*}
$$

by our methodology.
However, in $\tau_0$, $p_0$ continuously wants to enter the critical section while in $\tau_1$, $p_1$ never enters the critical section and in $\tau_2$, $p_2$ never enters the critical section, indicating a deadlock.

### 9.3 Running Data-Race-Free Programs

In this section we prove that data-race-free programs behave on hybrid consistent memory implementations as if they were sequentially consistent.

#### 9.3.1 Definition of Data-Race-Free Programs

The following definitions are modeled after those in [3]. Let $op^i_1$ and $op^j_2$ be two operations appearing in some sequence of memory operations $\alpha$. Then

- $op^i_1 \xrightarrow{\alpha} op^j_2$ if $i = j$ and $op^i_1 \xrightarrow{\alpha} op^j_2$.
- $op^i_1 \xrightarrow{\text{so}} op^j_2$ if both $op^i_1$ and $op^j_2$ are strong operations and $op^i_1 \xrightarrow{\alpha} op^j_2$.

The relation $happens before$, denoted by $\xrightarrow{\text{hb}}$, is the transitive closure of the union of $\xrightarrow{\alpha}$ and $\xrightarrow{\text{so}}$. This definition is similar to the definition of $happens before$ in [3] and is closely related to the definition of happened before defined by Lamport [42] for message passing systems.

Two memory accesses conflict if they both access the same memory location and at least one of them is a write. A data race occurs in a sequence of memory operations when two conflicting memory accesses are not ordered by the happens before relation.

**Definition 9.1** A program is data-race-free if none of its sequential executions contains a data race.

#### 9.3.2 The Influence Relation

Before we continue the discussion about data-race free programs, we formalize the notion of one operation influencing another, which relies on the control order. This notion is necessary for the proof that every data-race free program runs on every hybrid consistent memory as if
it was sequentially consistent. Recall that two operations by the same process are ordered in
the control order if there is a control operation ordered between them in the flow control order.

Formally, let $\tau$ be a sequence of memory operations and let $\equiv$ be a partial order on
the operations that is consistent with $\tau$, for each $p_i$. An operation $op^1_i$ directly influences an
operation $op^2_k$ in $\tau$ (with respect to the $\equiv$'s), if one of the following holds:

1. $op^1_j \equiv op^2_k$ and $op^1_j$ is a read. (Note that $j = k$ in this case.) That is, $op^1_j$ is a read
operation which could affect the execution of $op^2_k$ through a control operation.

2. $op^2_k = r_k(y, v), op^1_j = w_j(y, v), op^1_j \xrightarrow{\tau} op^2_k$ and there does not exist $w\_h(y, u)$ such that
$u \neq v$ and $w\_j(y, v) \xrightarrow{\tau} w\_h(y, u) \xrightarrow{\tau} r_k(y, v)$. That is, $op^2_k$ is a read of the value written
by $op^1_j$ and there is no intervening write of a different value.

The influence relation is the transitive closure of direct influence. Thus the influence relation
is also defined with respect to a set of partial orders. Although we will not usually explicitly
mention these partial orders, the influence relation will be used with admissible operation sequences and the relevant partial orders will be the control orders for the corresponding flow control sequences.

Note that an operation directly influences another operation only if it is ordered before it
in $\tau$. Since the influence relation is the transitive closure of direct influence, we have:

**Claim 9.6** If $op^1_j$ influences $op^2_k$ in $\tau$, then $op^1_j \xrightarrow{\tau} op^2_k$.

The following lemma captures the intuition that if a read operation $op^1_j = r_j(x, v)$ does not
influence operation $op^2_k$, then $op^2_k$ would have been generated even if $op^1_j$ had read a value other
than $v$.

**Lemma 9.7** Let $\tau$ be a sequence of memory operations that is partially admissible with respect
to a set of flow control sequences $\{fcs_i\}_{i=1}^n$. Let operation $op^1_j$ in $\tau$ be a read $r_j(x, v)$ that does
not influence any operation in $\tau$. Let $\tau'$ be the result of taking $\tau$ and changing $op^1_j$ to be
$r_j(x, w)$ for some $w \neq v$. Then $\tau'$ is partially admissible for some set of flow control sequences
$\{fcs'_{i}\}_{i=1}^n$. 
9.3.3 Running Data-Race-Free Programs on Hybrid Consistent Memory

We prove that every hybrid consistent execution of a data-race-free program is sequentially consistent.

To prove this result, we consider a legal sequence of memory operations \( \tau_i \), as guaranteed for some process \( p_i \) in the definition of hybrid consistency, that is minimal with respect to the number of switched operations (operations by the same process \( p_i \) whose order in \( \tau_i \) is not consistent with \( p_i \)'s flow control sequence). We show that if \( \tau_i \) is not fully admissible (i.e., a sequential execution), then there exists a prefix of a sequential execution of the program that contains a data race. If \( \tau_i \) is not fully admissible, it must contain at least one pair of switched operations. We locate the "first" pair of switched operations in \( \tau_i \), such that no other pair of switched operations is ordered between them. Because \( \tau_i \) is minimal we know this pair was switched to preserve legality. This fact is used to show that there is a data race between some pair of operations that precedes this switched pair. Our main problem is to place these two operations (and the data race between them) in a legal and partially admissible sequence. This is done by taking the two operations and the operations that influence them and ordering them as in \( \tau_i \), and adding any operations necessary to preserve the flow control sequences of all processes. The key point to prove about the resulting sequence is its legality. In doing so, we either change the value that a read returns (and invoke Lemma 9.7), or, if this does not help, we show that there is a data race earlier in the sequence. Thus we have constructed a prefix of a sequential execution with a data race, which is a contradiction. The details follow.

Fix a data-race-free program \( \text{Prog} \), a hybrid consistent execution \( \sigma \) of \( \text{Prog} \), and as guaranteed by the definition of hybrid consistency, a subset \( S \) of the memory operations and a flow control sequence \( \text{fcs}_i \) for each process \( p_i \).

Fix an arbitrary process \( p_i \). Let \( T \) be the set of all operation sequences \( \tau_i \) that satisfy Definition 8.3 (hybrid consistency). For \( \tau_i \) in \( T \), an ordered pair of operations of \( p_i \), \( \langle op^2_i, op^1_i \rangle \), is switched in \( \tau_i \) if \( op^2_i \overset{\tau_i}{\longrightarrow} op^1_i \), but \( op^1_i \overset{fcs_i}{\nrightarrow} op^2_i \).

Let \( \tau_i \) be some element of \( T \) with a minimal set of switches. That is, there does not exist \( \tau'_i \in T \), such that the set of pairs of switched operations of all processes in \( \tau'_i \) is strictly contained in the set of pairs of switched operations of all processes in \( \tau_i \).

We prove the main theorem of this section by way of contradiction, using the following lemma.
Lemma 9.8 If $\tau_i$ is not fully admissible, then there exists a prefix of a sequential execution of $\text{Prog}$ which contains a data race.

Proof: Assume that $\tau_i$ is not fully admissible. We first prove the following claim, which locates the pair of operations that is a candidate for a data race.

Claim 9.9 There exist two operations $\text{op}_2^j(x, v)$ and $\text{op}_k^1(x, w)$ in $\tau_i$ such that

1. $\text{op}_2^j(x, v) \xrightarrow{\tau_i} \text{op}_k^1(x, w)$,
2. there is a data race between $\text{op}_2^j(x, v)$ and $\text{op}_k^1(x, w)$ in $\tau_i$, and
3. there is no pair of switched operations in $\tau_i$ up to $\text{op}_k^1(x, w)$.

Proof: Since $\tau_i$ is not fully admissible, there exists at least one pair of switched operations in $\tau_i$. Let $\langle \text{op}_2^j(x, v), \text{op}_k^1(y, u) \rangle$ be the first ordered pair of switched operations, i.e., there is no other pair of switched operations which is completely ordered before $\text{op}_k^1(y, u)$ in $\tau_i$. If there is more than one ordered pair that share the same second operation (namely, $\text{op}_k^1$), then choose the pair whose first operation is latest.

Since $\tau_i$ is minimal, $\text{op}_2^j(x, v)$ and $\text{op}_k^1(y, u)$ are switched in order to preserve legality.

Since the operations are by the same process $p_i$, and since the order of operations by the same process to the same object is preserved (last property in Definition 8.3), it follows that $x \neq y$. Thus, there exists an operation $\text{op}_k^1(x, w)$ which conflicts with $\text{op}_2^j(x, v)$ and $\text{op}_2^j(x, v) \xrightarrow{\tau_i} \text{op}_k^1(x, w) \xrightarrow{\tau_i} \text{op}_k^1(y, u)$.

Note that since $\text{op}_2^j$ and $\text{op}_k^1$ are switched, by the definition of hybrid consistency, there are no strong operations by $p_i$ between $\text{op}_2^j$ and $\text{op}_k^1$ in $\tau_i$, including $\text{op}_2^j$ and $\text{op}_k^1$.

It is also the case that $j \neq k$. Suppose otherwise. Either $\text{op}_2^j$ and $\text{op}_k^1$ are switched or else $\text{op}_k^1$ and $\text{op}_2^j$ are switched. In the first case, $\langle \text{op}_2^j, \text{op}_k^1 \rangle$ would have been the pair chosen and in the second case, $\langle \text{op}_k^1, \text{op}_2^j \rangle$ would have been the pair chosen.

Since $j \neq k$, $\text{op}_2^j$ and $\text{op}_k^1$ conflict, and there is no strong operation by $p_i$ between them, there is a data race between $\text{op}_2^j$ and $\text{op}_k^1$ in $\tau_i$. This proves the claim. \qed
Let $I$ be the set of operations in $\tau_i$ that influence either $op_i^j$ or $op_i^k$ (including $op_i^j$ and $op_i^k$ themselves). Let $\tau'_i$ be the shortest prefix of $\tau_i$ that includes every operation in $I$. Let $\pi$ be the subsequence of $\tau'_i$ which contains exactly the set of all operations in $I$. The following claim shows that $\pi$ is consistent with $\xrightarrow{fcs_i}$, for all $l$.

Claim 9.10 For every pair of operations $op_l^j$ and $op_l^2$, if $op_l^j \xrightarrow{\tau} op_l^2$ then $op_l^j \xrightarrow{fcs_i} op_l^2$.

Proof: By Claim 9.6, the influence relation is consistent with $\xrightarrow{\tau}$. Therefore, no operation in $I$ follows $op_l^j$ in $\tau_i$, implying that $op_l^j$ is the last operation in $\tau'_i$. By Condition 3 in Claim 9.9, $\tau'_i$ does not include any pair of switched operations, and neither does $\pi$.

Now, we construct a new partially admissible (with respect to the $fcs_i$'s) sequence $\pi'$ by adding to $\pi$ every operation $op_l^j$ not in $\pi$, such that $op_l^j \xrightarrow{fcs_i} op_l^2$ and $op_l^2$ is in $\pi$, for each process $l$, as follows. For every such operation that was originally in $\tau'_i$, add it in the same place as in $\tau'_i$. For all other operations, add them arbitrarily, maintaining consistency with $\xrightarrow{fcs_i}$, for all $l$. By the definition of hybrid consistency, every added operation $op_l^j$ that is not in $\tau'_i$ must be weak. This is because $op_l^j$ precedes $op_l^2$ in $\xrightarrow{fcs_i}$, but follows $op_l^2$ in $\tau_i$, a possibility precluded of any strong operation by hybrid consistency (Condition 3 in Definition 8.3).

We would now like to determine if $\pi'$ is a legal sequence. By the definition of the influence relation and $\tau_i$, $\pi$ is a legal sequence. In particular, all reads in $\pi$ are legal. Thus, if there are illegal reads in $\pi'$, then each illegal read is either a read that was added to $\pi$, or a read that became illegal because of some write that was added to $\pi$.

Consider every illegal read $r_i$ in $\pi'$ that was added to $\pi$. We claim that $r_i$ does not influence any operation in $\pi'$, with respect to the $co_i$'s. Suppose it does. Since $r_i$ is a read, it must influence an operation through a control relation, i.e., $r_i \xrightarrow{co_i} op_l$, for some $l$. Since $r_i$ is not in $I$, $op_l$ is not in $I$. Thus $op_l$ was included in $\pi'$ because there exists $op'_l$ in $I$ such that $op'_l \xrightarrow{fcs_i} op_l$. This implies that $r_i \xrightarrow{co_i} op'_l$ and that $r$ is in $I$. We therefore have a contradiction.

Now, for each of the illegal reads in $\pi'$ that were added to $\pi$, change its value to match the value of the most recent write to the same object. (Note that this can be done since an illegal read in $\pi'$ does not influence any other operation in $\pi'$.) Call this sequence $\pi''$. Apply Lemma 9.7 successively for each fixed-up read, starting with $\pi'$, to deduce that $\pi''$ is partially admissible with respect to some set of flow control sequences $\{fcs'_i\}_{i=1}^\nu$. So, if there are no
illegal reads in \( \pi'' \), then \( \pi'' \) is legal and it follows that it is a prefix of a sequential execution of Prog.

To complete the proof of the lemma when \( \pi'' \) is legal, we now show that \( \pi'' \) contains a data race between \( op_j(x,v) \) and \( op_k(x,w) \). By Claim 9.9, there is no strong operation by \( p_j \) between \( op_j \) and \( op_k \) inclusive in \( \tau'_j \); also, note that no strong operation was added to \( \pi \). Also, \( j \neq k \), thus \( op_j \) and \( op_k \) are not ordered by the happens-before relation.

We are left with the case in which there is an illegal read in \( \pi'' \). It became illegal due to the insertion of some write. Let \( r^1_q(z,v_1) \) be the first illegal read in \( \pi'' \), and let \( w^1_p(z,v_2), v_1 \neq v_2 \), be the corresponding inserted write. Denote by \( \sigma \) the shortest prefix of \( \pi'' \) which includes \( r^1_q(z,v_1) \). Let \( \sigma' \) be the same sequence where \( r^1_q(z,v_1) \) is replaced by \( r^2_q(z,v_2) \). We complete the proof by showing:

**Claim 9.11** \( \sigma' \) is a prefix of a sequential execution of Prog which contains a data race between \( r^2_q \) and \( w^1_p \).

**Proof:** We first show that \( r^1_q \xrightarrow{r} w^1_p \). Assume otherwise that \( w^1_p \xrightarrow{r} r^1_q \). Since \( w^1_p \) is not in \( I \), it must not influence \( r^1_q \) in \( \tau_i \). Therefore, there is another write \( w^2_p(z,v_1) \) ordered between \( w^1_p \) and \( r^1_q \) which influences \( r^1_q \) in \( \tau_i \). But, then \( w^1_p, w^2_p \) and \( r^1_q \) would be ordered similarly in \( \pi' \). This gives a contradiction to the assumption that \( r^1_q \) becomes illegal due to the insertion of \( w^1_p \).

So we can assume that \( r^1_q \xrightarrow{r} w^1_p \); it follows that \( w^1_p \) is not in \( \tau'_i \), implying that \( w^1_p \) is weak. We claim that \( p \neq q \). Otherwise, if \( p = q \), then \( w^1_p(z,v_2) \xrightarrow{fcs} r^1_q(z,v_1) \). By the last property of hybrid consistency, it follows that \( w^1_p(z,v_2) \xrightarrow{r} r^1_q(z,v_1) \), a contradiction. Therefore, \( p \neq q \).

We now show that \( r^1_q(z,v_1) \) and \( w^1_p(z,v_2) \) are not ordered by the happens before relation in \( \pi'' \), and hence, they are not ordered by the happens before relation in \( \sigma \). Suppose they were. Then, there is a strong operation \( sop_p \) in between \( w^1_p \) and \( r^1_q \) in \( \pi'' \). Therefore, \( w^1_p \xrightarrow{fcs} sop_p \) and thus \( w^1_p \xrightarrow{sop_p} \). Therefore, \( sop_p \) is not in \( \tau'_i \) since \( w^1_p \) is not in \( \tau'_j \), and hence is weak, a contradiction. Therefore, \( r^1_q(z,v_1) \) and \( w^1_p(z,v_2) \) are not ordered by the happens before relation in \( \pi'' \), and hence, in \( \sigma \), and there exists a data race between them in \( \sigma \). It follows that \( \sigma' \) contains a data race.

Now, \( \sigma' \) is a legal sequence of operations due to the change in the value returned by the read, implied by replacing \( r^1_q \) with \( r^2_q \). Since \( r^2_q(z,v_2) \) is the last operation in \( \sigma' \), it clearly does
not influence any operation in $\sigma'$. By Lemma 9.7, $\sigma'$ is partially admissible for some set of flow control sequences $\{fcs_i^{(c)}\}_{i=1}^\omega$. Therefore, $\sigma'$ is a prefix of sequential execution of $Prog$ which contains a data race between $r^1_p(z, v_2)$ and $w^1_p(z, v_2)$, as needed.

Hence, we have shown that if $\tau_i$ is not fully admissible, then there exists a prefix of a sequential execution of $Prog$ which contains a data race.

The proof of the following theorem is now immediate.

**Theorem 9.12** Every hybrid consistent execution of a data-race-free program is sequentially consistent.

**Proof:** Let $\sigma$ be a hybrid consistent execution of a data-race-free program $Prog$. Fix a subset $S$ of the memory operations in $\sigma$ and a set of flow control sequences $fcs_i$ from the definition of hybrid consistency. Choose a process $p_i$. Let $\tau_i$ be a minimal legal permutation of $S$ as guaranteed by the definition of hybrid consistency. Since $Prog$ is data-race-free, no sequential execution has a data race. Then by Lemma 9.8, $\tau_i$ is fully admissible. Thus, $\sigma$ is a sequentially consistent execution of $Prog$.

**9.3.4 Reordering Operations of the Same Process**

The definition of hybrid consistency in Chapter 5 does not include the last property, i.e., that all writes by the same process to the same object appear in the views of all other processes in the order implied by the flow control sequence of their invoking process. (The definition in Chapter 5 also does not consider control operations.) Thus two weak operations by the same process $p_i$ accessing the same location could be viewed by other processes in a different order from the order in which they appear in $fcs_i$. We show that if this behavior is allowed, then it is not true that every hybrid consistent execution of a data-race-free program is sequentially consistent.

Consider the data-race-free program in Figure 9.4, assuming $x$ and $y$ are initially 0 and all instructions are weak, and the execution $\sigma$ in Figure 9.5 generated by this program. Note that $\sigma$ is not sequentially consistent. This is because in every legal sequence $\tau$ in which all operations of the same process appear in the same order as they appear in the flow control sequence for that process, the following must hold:
\[ p_1 \text{'s program} \]
\[
\begin{align*}
\text{tmp}_1 & := \text{read}(x); \\
\text{write}(x, 5); \\
\text{if } \text{tmp}_1 = 5 \text{ then} \\
\text{write}(y, 5); \\
\end{align*}
\]
\[ p_2 \text{'s program} \]
\[
\begin{align*}
\text{tmp}_2 & := \text{read}(y); \\
\text{write}(y, 5); \\
\text{if } \text{tmp}_2 = 5 \text{ then} \\
\text{write}(x, 5); \\
\end{align*}
\]

Figure 9.4: Reordering the operations of the same process to the same object

<table>
<thead>
<tr>
<th></th>
<th>( p_1 )</th>
<th></th>
<th>( p_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( r_1^1(x, 5) )</td>
<td>1</td>
<td>( r_1^1(y, 5) )</td>
</tr>
<tr>
<td>2</td>
<td>( w_1^1(x, 5) )</td>
<td>2</td>
<td>( w_2^1(y, 5) )</td>
</tr>
<tr>
<td>3</td>
<td>( w_1^3(y, 5) )</td>
<td>3</td>
<td>( w_2^3(x, 5) )</td>
</tr>
</tbody>
</table>

Figure 9.5: A possible execution \( \sigma \) if operations of the same process to the same object can be reordered

- \( w_1^3(x, 5) \xrightarrow{\tau} r_1^1(y, 5) \) and \( w_2^3(x, 5) \xrightarrow{\tau} r_1^1(x, 5) \), and
- \( r_1^1(x, 5) \xrightarrow{\tau} w_1^3(y, 5) \) and \( r_2^1(y, 5) \xrightarrow{\tau} w_2^3(x, 5) \).

Hence, there is a cycle in \( \tau \) and it is therefore not a sequence. On the other hand, \( \sigma \) is hybrid consistent; for this we can take:

\[
\begin{align*}
\tau_1 &= w_2^3(y, 5), r_1^2(y, 5), w_2^3(x, 5), r_1^1(x, 5), w_1^3(x, 5), w_1^3(y, 5) \\
\tau_2 &= w_1^3(x, 5), r_1^1(x, 5), w_1^3(y, 5), r_2^1(y, 5), w_2^3(y, 5), w_2^3(x, 5)
\end{align*}
\]

9.4 Programmability vs. Hardware Optimizations

In the definition of hybrid consistency, we have made several choices according to our beliefs about the guarantees that programmers want to get from a consistency condition. Of course, these choices may disallow some hardware optimizations. However, these choices are not an inherent limitation of the framework. In particular, if a certain choice is found to be too restrictive for some invaluable hardware optimization, then the definition can be adjusted very easily so that the hardware optimization can be supported.
For example, consider the program in Figure 9.6 and assume that the initial values of $x$ and $y$ are 0. In this program, the two writes by $p_1$ are labeled as strong while the two reads of $p_2$ are labeled as weak. The definition of hybrid consistency requires that every process views its own weak reads in the same order as they appear in its flow control sequence. Hence, if the first read by $p_2$ returns 1 from $y$, then the second read by $p_2$ cannot return 0 from $x$.

We believe that this corresponds to the semantics expected by programmers. However, this implies that implementations of hybrid consistency that do not incorporate roll-back cannot afford to execute weak reads out-of-order. This is due to the fact that when the run-time system decides on the execution order for two weak reads by the same process, it may not know whether there are concurrent strong writes to the same objects (as in Figure 9.6). However, if designers of a certain system decide that it is important to allow a process to view its own weak reads out-of-order (in order to allow more hardware optimizations), then the definition of hybrid consistency can be modified accordingly. This is done simply by removing the requirement that every process must view its own operations in the order they appear in its flow control sequence. In this case, the result about running data-race free programs (Theorem 9.12) is still correct, but the programming techniques developed in Sections 9.1 and 9.2 are no longer correct.

Figure 9.6: The ordering of weak reads

$p_1$'s program

\begin{align*}
\text{swrite}(x, 1); \\
\text{swrite}(y, 1);
\end{align*}

$p_2$'s program

\begin{align*}
\text{read}(y); \\
\text{read}(x);
\end{align*}
Chapter 10

Alpha Consistency

In this chapter, we define *alpha consistency*, a formal definition of the shared memory model of a DEC-Alpha based multiprocessor, and develop two basic programming techniques for it. The first approach is to protect shared data within critical sections. For this approach, we show how to transform any given non-cooperative solution for the mutual exclusion problem based on sequential consistency into a solution based on alpha consistency. We also show that this general method is not optimal; in some cases, a specific analysis of the code yields a more efficient solution. The second approach is to run data-race free programs. For this approach, we show that every data-race free program runs on alpha consistent hardware as if it was sequentially consistent.

10.1 The Alpha Shared Memory Model

The definition presented in this section follows the description in [1, 57]. It ignores the issues of fetching instructions and updating the translation buffer and assumes that each object has an initial value.

10.1.1 Informal Description

The Alpha has four types of memory instructions: *load*, *load-locked*, *store* and *store-conditional*. Intuitively, a load-locked operation reads from a memory location and marks this location. A
store-conditional tries to store data into a memory location. If the memory location is still marked, then the memory location is updated and we say the the store-conditional is successful. Otherwise, the store-conditional has no affect on the memory location and we say that the store-conditional is unsuccessful. A store or a successful store-conditional always cancel the marking of the memory location it accesses (if such marking exists). This includes a store or a store-conditional by the processor that invoked the load-locked that caused the memory location to be marked.

Load-locked and store-conditional allow to implement sequences that imitate “atomic” read-modify-write operations in the following optimistic manner: First, the memory location is read into a local register using a load-locked. The result is then modified in the local register. Finally, the contents of the local register are written into the memory location using a store-conditional. If the store-conditional is successful, then it means that no other processor tried to store to the memory location between the load-locked and the store-conditional. Hence, the sequence has an “atomic” effect and it terminates. Otherwise, it starts all over again.

Note that in practice, a load-locked marks a complete cache line, which usually includes more than one memory location. Similarly, a store/store-conditional cancels the marking of a whole cache line. Thus, sometimes a store-conditional does not succeed with no apparent reason. This is because of another store/store-conditional writing into a memory location that is included in the same cache line and clearing the marking of the whole line.

Alpha is a highly optimized architecture which allows operations to be invoked in a pipeline and out of order. Every processor must “view” all operations in some order. However, the order in which operations of the same processor are viewed may be different from the order in which they were invoked (or the order they appear in the program). Whenever the order between two or more memory operations must be preserved, a memory-barrier (MB) operation must be used. It is guaranteed that all operations invoked by a processor before a memory-barrier operation will be viewed by all processors before the memory-barrier operation. Symmetrically, all operations invoked by a processor after a memory-barrier operation will be viewed by all processors after the memory-barrier operation. (This and the following requirement form what is called the program issue order in [1]).

Finally, two operations that are invoked by the same processor and access the same memory location must be “viewed” by all processors in the order implied for these operations by the program. (This is the location access order defined in [1].)
10.1.2 Defining Alpha Consistency

We start with some simple notation. The operations allowed on an alpha object are load, store, load-locked and store-conditional. A read is either a load or a load-locked and a write is either a store or a store-conditional. Whenever we wish to emphasize that a read by $p_j$ is a load operation that reads a value $v$ from an object $x$, we denote it by $l_j(x, v)$. A load-locked operation by $p_j$ that reads a value $v$ from an object $x$ is denoted by $ll_j(x, v)$. A store operation by $p_j$ that writes a value $v$ to an object $x$ is denoted by $s_j(x, v)$. A store-conditional operation by $p_j$ that attempts to write a value $v$ to an object $x$ is denoted by $sc_j(x, v)$.

The memory of an Alpha based multiprocessor consists of a collection of alpha objects and one special null object. Given a sequence of operations $\tau$, a store-conditional is labeled as either successful or unsuccessful, according to the following rule: A store-conditional $sc_j(x, v)$ must be labeled as unsuccessful in $\tau$ if one of the following holds:

a. there is no load-conditional $ll_j^3(x, u)$ such that $ll_j^3(x, u) \xrightarrow{\tau} sc_j^1(x, v)$ or

b. there is another store or successful store-conditional $w_j^3(x, w)$ such that $w_j^3(x, w) \xrightarrow{\tau} sc_j^1(x, v)$ and there is no load-locked $ll_j^2(x, u)$ ordered between them in $\tau$.

Other store-conditional operations in $\tau$ may be labeled as either successful or unsuccessful. A write is successful if it is either a store or a successful store-conditional; otherwise it is unsuccessful. The serial specification of an alpha object includes every sequence $\tau$ of read and write operations in which every read returns the value of the last successful write in $\tau$, or the initial value of the object if there is no such write. A read is legal in a sequence of operations $\tau$ if it returns the value of the last successful write to the same object in $\tau$; otherwise, the read is illegal in $\tau$.

A memory-barrier (MB) operation always access the null object. The serial specification of the null object includes every possible interleaving of memory-barrier operations.

**Definition 10.1 (alpha consistency)** An execution $\sigma$ is alpha consistent if there exists a set of flow control sequences $\{fcs_j\}_{j=1}^n$ and a legal permutation $\tau$ of all operations in $\sigma$ such that the following hold for any $j$:

---

1. In order to capture the implementation considerations discussed above, this definition allows a store-conditional to be unsuccessful with no apparent reason.
1. If \( op_1^j \xrightarrow{s_j} op_2^j \), then \( op_1^j \xrightarrow{\tau} op_2^j \).

2. If \( op_1^j \xrightarrow{s_j} op_2^j \) and either \( op_1^j \) or \( op_2^j \) is an MB operation, then \( op_1^j \xrightarrow{\tau} op_2^j \).

3. If \( op_1^j \xrightarrow{s_j} op_2^j \) and they both access the same object, then \( op_1^j \xrightarrow{\tau} op_2^j \).

**Remark:** The definitions in Chapter 9 include a choice of a set of memory operations, to support speculative evaluation of the code. Since Alpha decides on branches before the following operations reach their write-back stage, it does not rely on speculative execution. While this feature could have been added to the above definition we decided not to include it for the sake of transparency.

### 10.1.3 Examples

It is clear that alpha consistency is weaker than sequential consistency. Hence, executions \( \sigma_1 \) and \( \sigma_2 \) in Figure 3.2 are alpha consistent.

**Execution \( \sigma_3 \):** We claim that this execution is alpha consistent. For this, we can take

\[
\tau = r_2^3(x,0), w_1^1(x,1), r_1^3(y,0), w_2^1(y,1).
\]

However, if we add to \( \sigma_3 \) an MB operation by \( p_1 \) between \( w_1^1(x,1) \) and \( r_1^3(y,0) \) and an MB operation by \( p_2 \) between \( w_2^1(y,1) \) and \( r_2^3(x,0) \), then the resulting execution \( \sigma_3' \) (Figure 10.1) is no longer alpha consistent. The reason for this is that in every legal serialization of \( \sigma_3' \), the order of either the operations of \( p_1 \) or the operations of \( p_2 \) must be changed w.r.t. their flow control sequences. However, due to the MB operations, in any sequence of operations that obeys the requirements in the definition on alpha consistency, the operations of \( p_1 \) and \( p_2 \) must appear in the order they appear in their flow control sequence. Therefore, no legal serialization of \( \sigma_3' \) can obey all the requirements in the definition of alpha consistency.

**Execution \( \sigma_4 \):** We claim that this execution is not alpha consistent. The reason for this is that in each legal serialization of \( \sigma_4 \), the order of either the operations of \( p_1 \) or the operations of \( p_2 \) must be changed w.r.t. their flow control sequences. However, all operations of \( p_1 \) access the same object and all operations of \( p_2 \) access the same object. Hence, in every sequence...
that obeys the requirements in the definition of alpha consistency, the operations of \( p_1 \) and \( p_2 \) must appear in the order they appear in their flow control sequences. Therefore, no legal serialization of \( \sigma_4 \) can obey all the requirements in the definition of alpha consistency.

**Execution \( \sigma_3 \):** We claim that this execution is alpha consistent. For this we can take

\[
\tau = r_2^2(y, 0), w_1^1(y, 1), w_1^1(y, 1), w_1^1(y, 1), w_1^2(x, 2), w_2^2(x, 2), r_3^1(x, 2).
\]

**Remark:** Since executions \( \sigma_3 \) and \( \sigma_4 \) are alpha consistent and not sequentially consistent (Section 3.2.1), it follows that alpha consistency is strictly weaker than sequential consistency.

### 10.1.4 Relation to the Specification of Alpha

The Alpha Architecture Handbook ([1]) defines a processor issue order and a location access order for each processor and every memory location, respectively. These partial orders are defined with respect to “a hypothetical simple implementation that contains one processor and a single shared memory, with no caches or buffers” [1, page 5-10]. For each processor, there exists a partial order, called processor issue order, on all operations of this processor; every pair of operations by the same processor must be ordered by the processor issue order if at least one of them is an MB operation or they both access the same memory location. For each memory location, there exists a partial order, called location access order, on all operations to the same location; every pair of memory operations that are issued by the same processor to the same location are required to appear in the location access order in the same order as in the processor issue order. The transitive closure of these two partial orders is used to define the notions of before and after in the natural way.

Note that the notion of a flow control sequence in Definition 10.1 formally captures the order in which operations are issued by “a hypothetical simple implementation”. It can be seen
that given a legal sequence of operations $\tau$ as guaranteed in Definition 10.1 and two operations $op_1^j$ and $op_2^j$, if $op_1^j$ is “before” $op_2^j$ (in the sense defined above), then $op_1^j \rightarrow op_2^j$. Similarly, if $op_1^j$ is “after” $op_2^j$, then $op_2^j \rightarrow op_1^j$.

The Alpha Architecture Handbook describes 9 litmus tests for verifying if a given execution obeys the alpha shared memory model or not [1, pages 5-13]. As further evidence, we show that litmus tests 3, 4 and 7 are satisfied by Definition 10.1. The other litmus tests can be verified similarly.

**Litmus test 3 (impossible sequence):** According to litmus test 3, assuming that the initial value of $x$ is 1, the execution $\sigma$ described in Figure 10.2 should not be alpha consistent.

We need to show that there does not exist a legal sequence of operations $\tau$ that obeys all requirements in the definition of alpha consistency (Definition 10.1). Assume, by way of contradiction, that such a sequence exists. By the legality of $\tau$ and the last requirement in the definition of alpha consistency, $w_i(x, 2) \rightarrow w_j(x, 3) \rightarrow r_i(x, 3)$. Similarly, $w_j(x, 3) \rightarrow r_k(x, 3) \rightarrow r_k(x, 2)$. A contradiction to the assumption that $\tau$ is legal.

**Litmus test 4 (sequence okay):** According to litmus test 4, assuming that the initial value of both $x$ and $y$ is 1, the execution $\sigma$ described in Figure 10.3 should be alpha consistent.

It is easy to verify that the following sequence of operations $\tau$ obeys all requirements in the definition of alpha consistency (Definition 10.1):

$$\tau = r_j(x, 1), w_i(x, 2), w_j(y, 2), r_j(y, 2)$$

Thus, $\sigma$ is an alpha consistent execution.
Litmus test 7 (impossible sequence): According to litmus test 7, assuming that the initial value of both $x$ and $y$ is 1, the execution $\sigma$ described in Figure 10.4 should not be alpha consistent.

We need to show that there does not exist a legal sequence of operations $\tau$ that obeys all requirements in the definition of alpha consistency (Definition 10.1). Assume, by way of contradiction, that such a sequence exists. By the legality of $\tau$ and the second requirement in the definition of alpha consistency,

$$w_i(x, 2) \xrightarrow{\tau} M B_i \xrightarrow{\tau} r_i(y, 1) \xrightarrow{\tau} w_j(y, 2) \xrightarrow{\tau} M B_j \xrightarrow{\tau} r_j(x, 1).$$

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

10.1.5 Control Order

The requirement that the sequence $\tau$ is consistent with the control order of each process (the first requirement in the definition of alpha consistency), is not an explicit part of the definition of the Alpha shared memory model that appears in [1, 57]. However, we believe that this requirement is an implicit part of the specification. This is because branch predictions are resolved by Alpha chips before the following instructions reach their write-back stage [49]. In other words, an operation does not have a permanent (global) effect before a previous conditional branch has been resolved. Also, the conditions in branches are resolved by examining the contents of local registers only. Therefore, any implementation will satisfy this condition anyway.

To understand why this requirement is also desirable, consider the example program in Figure 8.2, except that with alpha consistency, there is no distinction between weak and strong operations. In this program, the values returned by the reads affect the decision whether to invoke the writes and the invocations of the writes affect the possible values returned by the
reads. Assume this program is being executed on an alpha consistent memory. If we omit the first condition, then the definition would allow the execution $\sigma$ in Figure 8.3. To show $\sigma$ is alpha consistent we take

$$\tau = w_2^2(x, 5), w_1^1(y, 5), r_1^1(x, 5), r_2^1(y, 5).$$

As discussed in Section 8.4, it is important to eliminate such executions, with circular inference relations where the prediction about the result of a control operation could affect its actual result.

In Section 10.3, we define GDRF programs and show that every alpha consistent execution of a GDRF program is sequentially consistent (Theorem 10.7). According to these definitions, the program in Figure 8.2 is GDRF. If the first condition is omitted from the definition of alpha consistency, then the execution $\sigma$ in Figure 8.3 is alpha consistent. However, this execution is not sequentially consistent. Hence, Theorem 10.7 is not valid if we omit the the requirement that the sequence $\tau$ is consistent with the control order.

10.2 Programming with Critical Sections

In this section we present a general method that transforms any non-cooperative mutual exclusion algorithm based on sequential consistency into an algorithm based on alpha consistency. Following this, we give two examples in which an analysis of a specific mutual exclusion algorithm yields a more efficient code: Lamport’s bakery algorithm [41] and Peterson’s algorithm [53].

10.2.1 The General Method

Let $A$ be a non-cooperative mutual exclusion algorithm based on sequential consistency. Add to the entry and exit sections of $A$ the minimal number of MB instructions such that:

1. there is either an MB instruction or a control instruction between every two memory instructions that access different objects, and
2. the last instruction in both the entry and exit sections as well as the first instruction in the exit section is an MB instruction.

Denote the resulting algorithm by $A'$. We have:

**Theorem 10.1** $A'$ guarantees logical mutual exclusion based on alpha consistency.

The proof of this theorem follows closely the lines of the proof of a similar result for hybrid consistency that appears in Section 9.2. We outline this proof below.

By the definition of alpha consistency (Definition 10.1), the order of any pair of operations by the same processor such that at least one of them is either an MB operation or a control operation is the same as in the flow control sequence for their invoking processor. Similarly, the order of any pair of operations by the same processor that access the same memory object is the same as in the flow control sequence for their invoking processor. Thus, every alpha consistent execution of a program in which there is either an MB instruction or a control instruction between every two memory instructions of the same processor that access different locations is sequentially consistent.

Recall that in every execution of $A'$, there exists an MB operation immediately before entering a critical section and immediately after leaving a critical section. Assume there is a violation of logical mutual exclusion in some sequence $\tau$ from the definition of alpha consistency. Then either there is a deadlock in $\tau$, or there exists a prefix $\tau''$ of $\tau$ such that there are at least two processes executing the critical section together at the end of $\tau''$.

For an alpha consistent execution of $A'$, let $\tau$ be the sequence guaranteed by the definition of alpha consistency. We then eliminate from $\tau$ all operations that are not part of an entry or exit section. Denote the resulting sequence by $\tau'$. Since $A$ is non-cooperative, $\tau'$ includes all accesses to every object that is accessed in $\tau$. Also, since $\tau$ is legal, $\tau'$ is legal too. By the above discussion, $\tau'$ is sequentially consistent. Hence, if $\tau$ violates any of the requirements of logical mutual exclusion, then so does $\tau'$. Therefore, we have a sequentially consistent execution of $A$ (with empty critical sections) that violates some of the requirements of logical mutual exclusion. This contradicts the assumption that $A$ is a mutual exclusion algorithm based on sequential consistency.
1: \( s(\text{doorway}[i], \text{true}) \);
2: \( MB \);
3: \( my\_num := 1 + \max \{\text{number}[j] \mid j \neq i\} \);
4: \( MB \);
5: \( s(\text{number}[i], my\_num) \);
6: \( MB \);
7: \( s(\text{doorway}[i], \text{false}) \);
8: \( MB \);
9: \text{for } i := 1 \text{ to } n \text{ do}
10: \quad \text{if } (i \neq j) \text{ then}
11: \quad \quad \text{if } l(\text{doorway}[j]) \text{ then goto 11;}
12: \quad \quad \text{other} := l(\text{number}[j]);
13: \quad \quad \text{if } (\text{other} \neq 0) \text{ and } (\text{other}, j) < (my\_num, i) \text{ then goto 12;}
14: \quad \langle \text{critical section} \rangle
15: \quad s(\text{number}[i], 0);\)
16: \langle \text{remainder section} \rangle

Figure 10.5: Lamport’s bakery algorithm using alpha consistency – code for processor \( p_i \), \( i = 1, \ldots, n \).

10.2.2 Lamport’s Bakery Algorithm

Figure 10.5 presents Lamport’s mutual exclusion algorithm [41] with several MB operations added to guarantee its correctness based on alpha consistency. Lines 1–13 are the entry section; Lines 14–15 are the exit section. Note that the algorithm in Figure 10.5 uses less MB operations than required by the general method: there are no MB operations between the loads from \( \text{number}[j] \) in Line 3. This saves \( n - 2 \) MB operations for each execution of the entry section. Also, there is no MB operation at the beginning of the entry section and at the end of the exit section. Note the similarities between the proof of the algorithm w.r.t. alpha consistency and the proofs w.r.t. sequential consistency [14, 41, 54].

Lemma 10.2 The algorithm in Figure 10.5 guarantees logical mutual exclusion based on alpha consistency.

Proof: Consider an alpha consistent execution of the algorithm in Figure 10.5 and let \( \tau \) be the sequence guaranteed by the definition of alpha consistency. Due to the control operation
immediately before entering the critical section and the MB operation immediately after leaving the critical section, operations belonging to one critical section do not overlap with operations belonging to another critical section by the same processor. Thus, we need to show that the critical sections of different processors do not overlap in $\tau$.

Assume, by way of contradiction, that the operations of two critical sections $cs_i^k$ and $cs_j^k$ overlap in $\tau$ and let $\tau'$ be some prefix of $\tau$ that includes operations of both $cs_i^k$ and $cs_j^k$.

For the rest of this proof, let $\tau'$ be the shortest prefix of $\tau$ that includes operations of both $cs_i^k$ and $cs_j^k$. Let $s_i^{k_1}(number[i], u_i)$ be the last store to $number[i]$, $l_i^{k_2}(number[j], v_{ij})$ be the previous load from $number[j]$ and $l_i^{k_3}(number[j], z_{ij})$ be the last load from $number[j]$ before $p_k$ enters $cs_i^k$. Hence, $u_i$ is the last value written to $number[i]$ during $\tau'$; $v_{ij}$ is the last value read by $p_k$ from $number[j]$ before $u_i$ is calculated (Line 3 in the code); $z_{ij}$ is the last value read by $p_j$ from $number[j]$ before entering $cs_j^k$ (Line 12 in the code). We define $s_j^{k_1}(number[j], u_j)$, $l_j^{k_2}(number[i], v_{ji})$ and $l_j^{k_3}(number[i], z_{ji})$, and in particular $u_j$, $v_{ji}$ and $z_{ji}$, similarly for $p_j$.

Note that by the code of Line 3, $my\_num$ is always greater than 0. Hence, by the MB instructions in the entry section, the following claim is immediate:

**Claim 10.3** $u_i > 0$ and $u_j > 0$.

**Claim 10.4** $(u_j, j) > (u_i, i)$.

**Proof:** Since $p_k$ is executing the critical section, then by the code of the algorithm either $(z_{ij}, j) > (u_i, i)$ or $z_{ij} = 0$. If $(z_{ij}, j) > (u_i, i)$, then by Claim 10.3 and the legality of $\tau$, $u_j \geq z_{ij}$. Hence, $(u_j, j) > (u_i, i)$.

If $z_{ij} = 0$, then by the legality of $\tau$, $s_i^{k_1}(number[i], u_i) \xrightarrow{\tau} l_i^{k_2}(number[j], z_{ij}) \xrightarrow{\tau} s_j^{k_1}(number[j], u_j)$. Thus, the stores and loads from $doorway[j]$ guarantee that $s_i^{k_1}(number[i], u_i) \xrightarrow{\tau} l_j^{k_2}(number[j], v_{ji}) \xrightarrow{\tau} s_j^{k_1}(number[j], u_j)$.

Hence, by Claim 10.3 and the legality of $\tau$, $u_j > u_i$.

By interchanging the roles of $i$ and $j$ in the proof of Claim 10.4, we can show that $(u_i, i) > (u_j, j)$. This contradicts Claim 10.4.

**Lemma 10.5** The algorithm in Figure 10.5 is deadlock free based on alpha consistency.
Proof: Assume, by way of contradiction, that there exists a execution $\sigma$ of the algorithm in Figure 10.5 that contains a deadlock, and let $\tau$ be a sequence of operations from the definition of alpha consistency in which the deadlock occurs. We say that processor $p_i$ is *deadly* if $p_i$ is executing the entry section from some point on in $\tau$, but there is no processor that enters the critical section following this point. Denote by $u_j$ the last value stored in $\tau$ by $p_i$ to $number[j]$ due to the execution of Line 5, for any $j$ (if such a store exists). Let $p_i$ be the processor for which the value of $u_i(i)$ is minimal among the deadly processors. Thus, either $p_i$ continually executes Line 11, reading $doorway[k]$ for some $k$, or $p_i$ continually executes Lines 12–13, reading $number[k]$ for some $k$.

If $p_i$ continually executes Line 11, then since no processor enters its critical section, there is a prefix of $\tau$ that includes all operations that are invoked by $p_i$ while executing Lines 1–6. Let $s_k^1(doorway[k], true)$ and $s_k^2(doorway[k], false)$ be the last stores to $doorway[k]$. Since there is an MB instruction in the program between these two stores, $s_k^1(doorway[k], true) \xrightarrow{MB} s_k^2(doorway[k], false)$. Since there are infinite number of loads by $p_i$ from $doorway[k]$, there exists a load $l_1^1(doorway[k], z)$ such that $s_k^2(doorway[k], false) \xrightarrow{z} l_1^1(doorway[k], z)$. Since $\tau$ is legal and $s_k^2(doorway[k], false)$ is the last store to $doorway[k]$, $z = false$. A contradiction to the assumption that $p_i$ continually executes Line 11, reading $doorway[k]$.

Assume that $p_i$ continually executes Lines 12–13, reading $number[k]$ for some $k$. By the code of the algorithm, this can only happen if the value returned from $number[k]$ is greater than 0, meaning that there exists a prefix of $\tau$ in which $p_i$ is executing the entry section. By the minimality of $(u_i, i)$ and the code in Line 13, following some prefix of $\tau$, $p_i$ has left its critical section and never tries to enter it again. Let $s_k^1(number[k], 0)$ be the last store by $p_i$ to $number[k]$. Due to the MB instruction in Line 14, $s_k^1(number[k], 0)$ is the last store to $number[k]$ in $\tau$. Since there are infinite number of loads by $p_i$ from $number[k]$, there exists a load $l_1^2(number[k], z)$ such that $s_k^1(number[k], 0) \xrightarrow{z} l_1^2(number[k], z)$. Since $\tau$ is legal and $s_k^1(number[k], 0)$ is the last store to $number[k]$, $z = 0$. A contradiction to the assumption that $p_i$ continually executes Lines 12–13 reading $number[k]$. \hfill $\blacksquare$

**Lemma 10.6** The algorithm in Figure 10.5 is starvation free based on alpha consistency.

Proof: Assume, by way of contradiction, that there exists an execution $\sigma$ of the algorithm in Figure 10.5 in which some process $p_i$ is starved, and let $\tau$ be the sequence of operations from the
definition of alpha consistency in which the starvation occurs. By assumption, $p_k$ continually executes the entry section from some point on in $\tau$, but never executes the corresponding critical section in $\tau$. Thus, either $p_i$ continually executes Line 11 or $p_k$ continually executes Line 12–13 in $\tau$.

By Lemma 10.5, there exists another processor $p_k$ that enters and leaves its critical section infinitely often in $\tau$. Let $s_i^1(number[i], v)$ be the last store in $\tau$ to $number[i]$. Thus, there exists a load $l_k^1(number[i], u)$ that is invoked in Line 3 which is ordered in $\tau$ after $s_i^1(number[i], v)$. By the legality of $\tau$, $u = v$. Let $s_k^1(number[i], w)$ be the store to $number[k]$ that follows $l_k^1(number[i], u)$ in $\tau$. Thus, $w > u = v$. Let $l_k^2(number[i], z)$ be the next load by $p_k$ from $number[i]$. Hence, $z = u < w$. A contradiction to the assumption that $p_k$ enters and leaves its critical section infinitely often in $\tau$.

10.2.3 Peterson’s Algorithm

Figure 10.6 presents Peterson’s mutual exclusion algorithm [53] for two processors with the addition of several MB operations that guarantee its correctness based on alpha consistency. Lines 1–5 are the entry section; Lines 6–7 are the exit section. Note that the algorithm in Figure 10.6 uses less MB operations than required by the general method; One MB operation is saved at the end of the exit section. Even more significant is the fact that there is no MB operation between the two loads in Line 5 which is likely to be executed many times during every execution of the entry section. In the code we use the notation $l(x) = v$ to denote a load of $x$ returning the value $v$, for any read/write object $x$ and value $v$.

The proof that the algorithm in Figure 10.6 guarantees logical mutual exclusion based on alpha consistency is omitted (See [18, Section 5]). It is almost identical to the proof that the algorithm in Figure 5.1 guarantees logical mutual exclusion based on hybrid consistency, that appears in Chapter 5.

10.3 Running Generalized Data-Race Free Programs

In Chapter 9 and elsewhere [2, 3, 4, 34, 35, 51, 52], the notion of data-race free programs is defined for memory models where the memory operations can be classified into synchronization (or strong) operations and data (or weak) operations. This classification, however, is not
Figure 10.6: Peterson’s algorithm using alpha consistency – code for processor $p_i$, $i = 0, 1$.

Figure 10.6: Peterson’s algorithm using alpha consistency – code for processor $p_i$, $i = 0, 1$.

The code is applicable for alpha consistency, as alpha consistency does not include an obvious parallel for synchronization operations. Thus, we need to define sequences of operations that will imitate the effect of synchronization operations and adjust the definitions accordingly.

### 10.3.1 Generalized Data-Race Free Programs

Let $\sigma$ be an alpha consistent execution and let $\tau$ be a legal sequence of operations $\tau$ as guaranteed by the definition of alpha consistency (Definition 10.1). For some process $p_j$, consider a finite consecutive subsequence $\alpha$ of $\tau \mid j$. The subsequence $\alpha$ is a release sequence if the following hold:

1. the first operation in $\alpha$ is an MB operation,
2. all operations in $\alpha$ that are not MB operations access the same memory object, and
3. the last operation that is not an MB operation in $\alpha$ is either a successful write or a read.

Symmetrically, $\alpha$ is an acquire sequence if the following hold:

1. the last operation in $\alpha$ is an MB operation,
2. all operations in $\alpha$ that are not MB operations access the same memory object, and
3. the last operation that is not an MB operation in $\alpha$ is either a successful write or a read.
An example of such sequences appears below in Section 10.3.2.

Denote by $rel_i$ a release sequence by processor $p_i$ and by $acq_i$ an acquire sequence by processor $p_i$. That is, $rel_i$ includes all operations in the release sequence in the order they appear in the sequence; $acq_i$ includes all operations in the acquire sequence in the order they appear in the sequence. We slightly abuse the notation and denote by $op_j^1 \xrightarrow{r} rel_k$ the fact that $op_j^1$ appears in $\tau$ before every operation in $rel_k$ and by $acq_k \xrightarrow{a} op_j^1$ the fact that $op_j^1$ appears in $\tau$ after every operation in $acq_k$. Similarly, $rel_k \xrightarrow{r} acq_j$ denotes the fact that all operations in $rel_k$ appear in $\tau$ before all operations in $acq_j$.

Next we define the generalized happens before relation. This definition is similar to the definition of asymmetric happens before in [16, Section 7.2]. The only difference is that here we define the synchronization order w.r.t. release and acquire sequences rather than w.r.t. release and acquire operations. Formally, let $\tau$ be a sequence of operations.

- For two operations $op_i^1$ and $op_j^2$ in $\tau$, denote $op_i^1 \xrightarrow{p} op_j^2$ if $i = j$ and $op_i^1 \xrightarrow{r} op_j^2$ (this is the processor order).

- For a release sequence $rel_i^1$ and an acquire sequence $acq_j^2$ in $\tau$, denote $rel_i^1 \xrightarrow{g} acq_j^2$ if $rel_i^1 \xrightarrow{r} acq_j^2$ (this is the generalized synchronization order).

The generalized happens before relation, denoted by $\xrightarrow{gb}$, is the transitive closure of the union of $\xrightarrow{p}$ and $\xrightarrow{g}$.

We remind the reader that two memory accesses conflict if they both access the same memory location and at least one of them is a write. A generalized data-race occurs in a sequence of memory operations when two conflicting memory accesses that are not part of a release sequence or an acquire sequence are not ordered by the generalized happens before relation.

**Definition 10.2** A program is generalized data-race free (GDRF) if none of its sequential executions contains a generalized data-race.

### 10.3.2 MBV and MBP Sequences

We present an example of release and acquire sequences, MBV and MBP, which implement Dijkstra’s V and P operations on semaphores [27]. Specifically, some of the memory objects
are semaphore objects. These objects may be accessed only by operations belonging to MBV and MBP sequences and each such sequence may access only one semaphore object. The serial specification of a semaphore object includes all possible interleaving of MBV and MBP sequences such that in every prefix of the interleaving, the number of completed MBP sequences is larger than the number of completed MBV sequences by at most one.

Assuming that obj is a semaphore object initiated to 1 and reg is a local register the following are possible MBV and MBP sequences: (We remind the reader that we use the shorthand notation ll for load-locked and sc for store-conditional.)

<table>
<thead>
<tr>
<th>Code for MBV(obj)</th>
<th>Code for MBP(obj)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB;</td>
<td>ploop:</td>
</tr>
<tr>
<td>vloop:</td>
<td>reg = ll(obj);</td>
</tr>
<tr>
<td></td>
<td>if reg = 0 then goto loop;</td>
</tr>
<tr>
<td></td>
<td>reg = reg - 1;</td>
</tr>
<tr>
<td></td>
<td>sc(obj, reg);</td>
</tr>
<tr>
<td></td>
<td>if the store was unsuccessful then</td>
</tr>
<tr>
<td></td>
<td>goto vloop;</td>
</tr>
<tr>
<td></td>
<td>if the store was unsuccessful then</td>
</tr>
<tr>
<td></td>
<td>goto ploop;</td>
</tr>
<tr>
<td></td>
<td>MB</td>
</tr>
</tbody>
</table>

10.3.3 Running GDRF Programs on Alpha Consistent Memories

**Theorem 10.7** Every alpha consistent execution of a GDRF program is sequentially consistent.

The proof of this theorem follows the outline of the proof that every hybrid consistent execution of a data-race free program in sequentially consistent that appears in Chapter 9, with few simple modifications. We repeat most of the details for clarity.

**Proof:** We start the proof by generalizing the notion of one operation influencing another, defined on Page 111, to incorporate store-conditional operations. Let \( \tau \) be a sequence of memory operations and let \( co_i \) be a partial order on the operations that is consistent with \( \tau \), for each \( p_i \). An operation \( op_j^1 \) directly influences an operation \( op_k^2 \) in \( \tau \) (with respect to the \( co_i \)'s), if one of the following holds:

1. \( op_j^1 \xrightarrow{co_i} op_k^2 \) and \( op_j^1 \) is a read. (Note that \( j = k \) in this case.)
2. \( op^k_2 = r_k(y, v) \), \( op^j_1 = w_j(y, v) \), \( op^j_1 \xrightarrow{r} op^k_2 \) and there does not exist \( w_h(y, u) \) such that \( u \neq v \) and \( w_j(y, v) \xrightarrow{r} w_h(y, u) \xrightarrow{r} r_k(y, v) \).

3. \( op^k_2 \) is a successful store-conditional, \( op^j_1 \) is a write to the same object and \( op^j_1 \xrightarrow{r} op^k_2 \).

The influence relation is the transitive closure of direct influence. Thus the influence relation is also defined with respect to a set of partial orders. Note that Lemma 9.7 is valid also with the new definition of the influence relation.

We now turn to the proof of the main result of this section. Fix a GDRF program \( \text{Prog} \), an alpha consistent execution \( \sigma \) of \( \text{Prog} \) and, as guaranteed by the definition of alpha consistency, a flow control sequence \( fcs_i \) for each processor \( p_i \).

Let \( T \) be the set of all operation sequences \( \tau \) that satisfy Definition 10.1 (alpha consistency). Let \( \tau \) be some element of \( T \) with a minimal set of switches. That is, there does not exist \( \tau' \in T \), such that the set of pairs of switched operations in \( \tau' \) is strictly contained in the set of pairs of switched operations in \( \tau \). Recall that in a sequence \( \tau \in T \) with a minimal set of switches, a pair of operations is switched only if the “switching” is necessary in order to preserve legality.

We now define the notion of two operations being \emph{properly ordered} in a sequence of operations. We use this as an intermediate step in the proof of the theorem: We first prove that if there exists an alpha consistent execution of \( \text{Prog} \) that includes a pair of conflicting operations which are not properly ordered, then there exists a sequentially consistent execution of \( \text{Prog} \) that includes a pair of operations that are not properly ordered. From this, we derive the claim of the theorem as shown below.

\textbf{Definition 10.3} A pair of conflicting operations \( op^i_1 \) and \( op^j_2 \), \( i \neq j \), is properly ordered in a sequence of operations \( \tau \) if there exist two operations, \( MB_i \) and \( MB_j \), such that \( op^i_1 \xrightarrow{r} MB_i \xrightarrow{r} MB_j \xrightarrow{r} op^j_2 \). (It is possible that \( op^i_1 = MB_i \) and/or \( op^j_2 = MB_j \).)

The following claim is immediate:

\textbf{Claim 10.8} For every pair of conflicting operations \( op^i_1 \) and \( op^j_2 \), \( i \neq j \), and a sequence of operations \( \tau \), if \( op^i_1 \) and \( op^j_2 \) are not properly ordered in \( \tau \), then there exists a generalized data race between \( op^i_1 \) and \( op^j_2 \) in \( \tau \).
As before, we prove the main theorem of this section by way of contradiction, using the following lemma.

**Lemma 10.9** If $\tau$ is not fully admissible, then there exists a prefix of a sequential execution of $\text{Prog}$ which contains two operations that are not properly ordered.

The proof of this lemma is similar to the proof of Lemma 9.8, with MB operations playing the role of strong operations. Care is needed in handling store-conditional operations since a successful store-conditional may become unsuccessful due to the reordering of other stores to the same object. We repeat the main details for completeness.

**Proof:** Assume that $\tau$ is not fully admissible. The following claim locates the pair of operations that is a candidate for not being properly ordered. The proof of this claim is almost identical to the proof of Claim 9.9 and is therefore omitted.

**Claim 10.10** There exist two operations $op_i^2(x, v)$ and $op_i^1(x, w)$ in $\tau$ such that

1. $op_i^2(x, v) \xrightarrow{\tau} op_i^1(x, w)$,
2. $op_i^2(x, v)$ and $op_i^1(x, v)$ are not properly ordered in $\tau$, and
3. there is no pair of switched operations in $\tau$ up to $op_i^1(x, w)$.

Let $I$ be the set of operations in $\tau$ that influence either $op_i^2$ or $op_i^1$ (including $op_i^2$ and $op_i^1$ themselves). Let $\tau'$ be the shortest prefix of $\tau$ that includes every operation in $I$. Let $\pi$ be the subsequence of $\tau'$ which contains exactly the set of all operations in $I$. Claim 9.10 shows that $\pi$ is consistent with $\xrightarrow{\text{fc}_{\pi}}$, for any $l \in \{1, \ldots, n\}$. This claim is valid here as well.

Now, we construct a new partially admissible (with respect to $\{\text{fc}_{\pi}\}_{i=1}^n$) sequence $\pi'$ by adding to $\pi$ every operation $op_i^1$ not in $\pi$, such that $op_i^1 \xrightarrow{\text{fc}_{\pi}} op_i^2$ and $op_i^2$ is in $\pi$, for each processor $l$, as follows:

For every such operation that was originally in $\pi'$, add it in the same place as in $\tau$. For all other operations, add them arbitrarily, maintaining consistency with $\xrightarrow{\text{fc}_{\pi}}$, for all $l$. Label every store-conditional $sc_i(x, v)$ in $\pi'$ as successful if $sc_i(x, v)$ is labeled as successful in $\tau$ and $sc_i(x, v)$ appears in $\pi'$ after a load-locked $ll_i(x, u)$ such that there are no writes by other
processes between them; otherwise, label \( sc_l(x, v) \) as unsuccessful in \( \pi' \). By the definition of alpha consistency, none of the added operations (that is not in \( \pi' \)) is an MB operation.

We would now like to determine if \( \pi' \) is a legal sequence. By the definition of the influence relation and \( \pi, \pi \) is a legal sequence. In particular, all reads in \( \pi \) are legal. Thus, if there are illegal reads in \( \pi' \), then each illegal read is either a read that was added to \( \pi \), or a read that became illegal because of some write that was added to \( \pi \) (including writes that caused successful store-conditionals in \( \pi \) to become unsuccessful in \( \pi' \)).

Consider every illegal read \( r_i \) in \( \pi' \) that was added to \( \pi \). We claim that \( r_i \) does not influence any operation in \( \pi' \), with respect to the \( co_i \)'s. Suppose it does. Since \( r_i \) is a read, it must influence an operation through a control relation, i.e., \( r_i \xrightarrow{co} op_i \). Since \( r_i \) is not in \( \pi \), \( op_i \) is not in \( \pi \). Thus \( op_i \) was included in \( \pi' \) because there exists \( op_i' \) in \( I \) such that \( op_i' \xrightarrow{fcs} op_i \). This implies that \( r_i \xrightarrow{co} op_i' \) and that \( r_i \) is in \( I \). We therefore have a contradiction.

Now, for each of the illegal reads in \( \pi' \) that were added to \( \pi \), change its value to match the value of the most recent write to the same object. (Note that this can be done since an illegal read in \( \pi' \) does not influence any other operation in \( \pi' \).) Call this sequence \( \pi'' \). Apply Lemma 9.7 successively for each fixed-up read, starting with \( \pi' \), to deduce that \( \pi'' \) is partially admissible with respect to some set of flow control sequences \( \{ fcs_i \}_{i=1}^n \). So, if there are no illegal reads in \( \pi'' \), then \( \pi'' \) is legal. Hence, \( \pi'' \) is a legal partially admissible sequence of operations and by Claim 8.1, \( \pi'' \) is a prefix of a sequential execution of \( \text{Prog} \).

To complete the proof of the lemma when \( \pi'' \) is legal, we now show that \( op_p^\pi(x, v) \) and \( op_p^\pi(x, w) \) are not properly ordered in \( \pi'' \). By Claim 10.10, there is no MB operation by \( p_j \) between \( op_p^\pi(x, v) \) and \( op_p^\pi(x, w) \) inclusive in \( \pi' \); also, note that no MB operation was added to \( \pi \). Also, \( j \neq k \). Therefore, \( op_p^\pi(x, v) \) and \( op_p^\pi(x, w) \) are not properly ordered in \( \pi'' \).

We are left with the case in which there is an illegal read in \( \pi'' \). It became illegal due to the insertion of some write. Let \( r^l_\phi(z, v_1) \) be the first illegal read in \( \pi'' \), and let \( w^1_\phi(z, v_2) \), \( v_1 \neq v_2 \), be the corresponding inserted write. Denote by \( \sigma \) the shortest prefix of \( \pi'' \) which includes \( r^l_\phi(z, v_1) \). Let \( v_3 \) be the value written by the last successful write to \( z \) that appears before \( r^l_\phi(z, v_1) \) in \( \pi'' \), or the initial value of \( z \) if such a write does not exist. Let \( \sigma' \) be the same sequence where \( r^l_\phi(z, v_1) \) is replaced by \( r^2_\phi(z, v_3) \). We complete the proof by showing:

**Claim 10.11** \( \sigma' \) is a prefix of a sequential execution of \( \text{Prog} \) in which \( r^2_\phi \) and \( w^1_\phi \) are not properly ordered.
The proof of the above claim is almost identical to the proof of Claim 9.11 and is therefore omitted. Hence, we have shown that if $\tau$ is not fully admissible, then there exists a prefix of a sequential execution of $\text{Prog}$ which contains two operations that are not properly ordered. ■

The next lemma follows immediately from Claim 10.8 and Lemma 10.9.

**Lemma 10.12** If $\tau$ is not fully admissible, then there exists a prefix of a sequential execution of $\text{Prog}$ which contains a generalized data-race.

The proof of the theorem is now immediate. Let $\sigma$ be an alpha consistent execution of a GDRF program $\text{Prog}$. Fix a set of flow control sequences $fcs_i$ from the definition of alpha consistency. Let $\tau$ be a minimal legal permutation of the operations in $\sigma$ as guaranteed by the definition of alpha consistency. Since $\text{Prog}$ is GDRF, no sequential execution has a generalized data-race and Lemma 10.12 implies that $\tau$ is fully admissible. Thus, $\sigma$ is a sequentially consistent execution of $\text{Prog}$. ■

**Remark:** It is possible to define a data race to occur between two conflicting operations if they are not properly ordered. Clearly, the proof of Theorem 10.7 can be used for the modified definition, yielding a slightly stronger result. We have chosen to define data races using release and acquire sequences since this definition is more constructive. Moreover, it seems that it is easier to modify the existing techniques for detecting data races to work with the current definition, rather than with the alternative definition.
Chapter 11

Discussion

11.1 Review

As the demand for powerful computers grows faster than the technology to develop new processors, the need for highly parallel multiprocessors increases. Shared memory is an important paradigm that can help to reach a better utilization of these machines. However, the desire to provide the programmer with the same model of the world as can be found in real shared memory machines is often confronted with many technological limitations. It is often the case that the simplicity of the programming model is sacrificed so the performance and cost requirements can be met.

In order to use highly parallel shared memory multiprocessors, a crisp definition of the consistency condition supported by the distributed shared memory must be provided. In order to write correct programs, a programmer must have a good understanding of the consistency condition. Also, in order to actually exploit the potential performance of the distributed shared memory, the programmer must be provided with programming techniques that allow to write efficient programs using the specific consistency condition. Similarly, architects must be familiar with different consistency conditions and the effect they have on the performance of the system, the complexity of possible implementations for the different conditions and the programming models these conditions provide. This information is vital in order to choose the consistency condition that fits the design goals in an optimal manner.

The main goal of this thesis is to obtain a better understanding of consistency conditions
and their implications on both hardware and software. We have presented a framework in which consistency conditions can be defined and studied in a clean and systematic way. We started with a basic framework, and then generalized it to incorporate non-sequential executions. We used the framework to show that weak conditions are too weak to solve some of the fundamental problems of concurrent programming. This result serves as a theoretical justification for using hybrid conditions instead of weak conditions.

Following this, we have presented a formal definition of hybrid consistency, capturing the main features of many existing hybrid conditions. We used this definition to develop several algorithms for providing hybrid consistency, under various assumptions. In these algorithms, weak operations are executed instantaneously while the time required to implement strong operations is $O(d)$. We have also shown lower bounds on possible implementations of hybrid consistency, indicating that our algorithms are optimal, up to a constant multiplicative factor.

In order to make hybrid consistency more accessible to programmers, we present three programming methods that allow to write programs assuming a sequentially consistent hardware, and run them on a more efficient hybrid consistent hardware. The first method is to label all writes (or all reads) as strong. The second method, is to protect shared data within critical sections. For this method, we show a general method that transforms any given non-cooperative solution to the critical section based on sequential consistency into a solution based on hybrid consistency. The third method is to run data-race free programs.

Finally, we have defined alpha consistency, a formal definition of Alpha based multiprocessors, and developed programming methods for it. These methods are similar to the ones developed for hybrid consistency.

11.2 Further Research

11.2.1 Understanding Consistency Conditions

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 criteria must be defined and checked. These criteria will help to decide which consistency conditions are better suited for which tasks and could be useful in designing new multiprocessors and architectures. They
should reflect the efficiency and complexity of the possible implementations of the consistency conditions, their expressiveness and ease of use.

An important example of such a criterion is the execution time of parallel programs. Up till now the theoretical research of consistency conditions has concentrated on comparing the execution time of single operations. However, in practice, the important measure is the total execution time of the whole program. There could be many factors that affect the total execution time of the program, that we currently have no way of modeling. For example, in some consistency conditions it is possible to execute operations faster than with others, but in order for a program to run correctly, the programmer might have to use more operations. Thus, there is a need for a framework and criteria that can incorporate these issues.

11.2.2 Implementations

Partial Copies

All the algorithms developed in this thesis for providing hybrid consistency assume that each process holds a complete copy of the entire memory. This assumption has two severe drawbacks in real systems: (a) the memory requirements of these algorithms are very large and (b) the communication overhead caused by each memory access (in order to send messages to all processes) may deteriorate the performance of the whole system. However, with few modifications, our algorithms can be extended to handle partial copies.

If we allow process to hold partial copies of the memory, we must add to the algorithms a scheme for handling misses, i.e., attempts to access objects that are not included in the local copy of the memory. One such scheme is as follows: We assume that the memory is logically divided into pages and each page includes several objects. Whenever there is a miss, the invoking process sends a request for a copy of the page that includes the missing object to the nearest process which holds a copy of this page. Then, until the requested copy arrives, all messages directed to this page are buffered. Finally, when the requested copy arrives, the invoking process applies to this page all buffered messages whose timestamp indicate that they are more recent than the copy that was just received. Note that it is not sufficient to just copy the page that holds a missing object. This is because some messages that are directed to this page may be delivered at the process that sends the copy of the page between the time this page is sent and the time it arrives at the invoking process. These messages are not reflected
in the copy that is sent to the invoking process. Hence, if the invoking process ignores these messages, the data they carry will be lost.

In the case of read/write objects, instead of buffering all messages that are directed to the requested page, it is possible to apply these messages to a shadow copy of this page. Then, when the requested page arrives, it can be merged with the shadow copy using timestamps to decide what is the most updated value for each object. Choosing between the two schemes, in the case of read/write objects, can be done depending on the size of pages and their access pattern. We leave the problem of developing more efficient schemes for handling misses for further research.

Multi-threading

It would be also interesting to have implementations of hybrid consistency that support multi-threading. That is, several processes sharing a single processor, or in other words, sharing a single MCS process and a single run-time environment. Being able to run several threads on the same process in one of the oldest techniques to hide some of the latency of memory accesses in uniprocessors; when one thread waits for the completion of a long memory access, the operations of other threads may get executed. This technique has been also incorporated successfully with multiprocessors in the MIT Alewife [8]. Hence, it seems that implementations of hybrid consistency could also benefit from this.

Coarse Granularity

Another problem with the algorithms developed in this thesis is that they assume that the MCS process may send messages after every write. However, for current microprocessor architectures, this means that the operating system must catch every write which results in severe performance penalties. If hybrid consistency is to be implemented on existing architectures, algorithms that work with a granularity of a complete page and not a single object should be developed. This can be done, for example, by restricting the number of simultaneous writable copies of each page to one.
Network Topologies

Most of the theoretical study of implementing consistency conditions has been done assuming uniform networks. However, in many parallel and distributed multiprocessors more complicated topologies are used. Specific examples include trees, meshes and hypercubes. Many optimizations that have little or no meaning in uniform networks, e.g., optimal routing, finding the shortest path and low congestion, become very important in more complex topologies. The effect of choosing a consistency condition on the ability to use these optimizations should be investigated. In particular, theoretical implementations of various consistency conditions should be developed and optimized for these topologies.

Fault Tolerance

Most of the known algorithms that implement various consistency conditions are not fault tolerant. Others, implement only single-writer objects and are therefore inefficient for the more general case of multi-writer objects [15]. It would be interesting to develop efficient fault tolerant implementations for various consistency conditions.

11.2.3 Programming Techniques

As been shown before, sequential consistency cannot be implemented efficiently in large multiprocessors. On the other hand, weaker conditions are much more complicated to work with. Hence, developing programming techniques and methodologies for consistency conditions that are weaker than sequentially consistent seems vital in order to continue using distributed shared memories in the future. This includes, in particular, more heuristic rules and transformations of programs from sequential consistency to hybrid consistency, alpha consistency and PowerPC based multiprocessors. These rules and transformations can then be used by a smart compiler to transform correct programs based on sequential consistency into correct programs based on hybrid consistency, alpha consistency or PowerPC based multiprocessors, as the case may be.

It would be also interesting to develop transformations from one weaker condition to the other. In the short term, the two conditions that are of particular interest are alpha consistency and the memory model of PowerPC based multiprocessors, since they represent two existing commercial architectures.
Finally, an important question left open by this thesis is how to write correct programs, using minimal amount of synchronization. This means, for hybrid consistency, with the minimal number of strong operations, for alpha consistency, with the minimal number of MB operations and for PowerPC based multiprocessors, with the minimal number of sync operations.

### 11.2.4 Extending the Framework

As discussed in Section 8.4.1, the current framework does not support indirect addressing. However, this type of addressing is very important in real programs and our framework should be extended accordingly. At the moment we have some preliminary ideas about possible ways of doing this.

Our current framework is very low level and it does not support sophisticated flow control operations. In particular, it does not have an explicit support for procedures and functions and it does not have any support for multi-threading operations. It would be both interesting and important to incorporate these constructs.
Bibliography


Report TR 89-1042, Department of Computer Science, Cornell University, November 
1989.

TRCS-93-11, Department of Computer Science, University of California at Santa Barbara, 
July 1993.


Proc. of the 19th International Symposium on Computer Architecture, pages 2–12, May 