INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Views and Consistencies in Distributed Shared Memory

Gabriel Girard

A Thesis

in the

Department of Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montreal, Quebec, Canada

December 2000

© Gabriel Girard, 2000

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-59232-4

Canada

Abstract

Views and Consistencies in Distributed Shared Memory

Gabriel Girard, PhD
Concordia University, 2000

The distributed shared memory (DSM) abstraction is a very popular programming paradigm in parallel and distributed environments. However, DSM often suffers from performance problems as consistency requirements often incur long access latencies that cannot be overlapped with other operations in a process. Sequential consistency is the most general consistency requirement for DSM systems.

This thesis explores two different avenues to solve the performance problem for DSM systems. First, for sequentially consistent DSM, we introduce a new strategy to minimize synchronization cost and maximize the hiding of synchronization delays in a process. The strategy is based on the knowledge of spatial locality in the sharing of memory objects. An *access graph* is used to capture the sharing relationship among processes via the shared objects. We show that if accesses in all cycles are 'properly' synchronized, then the execution is guaranteed to be sequentially consistent. We develop two distinct solution strategies to ensure proper synchronization. (i) Neighbor protocol: conflicting accesses between two neighbors in an access cycle must be synchronized, and (ii) flush protocol: asynchronous accesses in an access cycle must be eventually synchronized by a special flush-access in the cycle. Simulation experiments have shown significant improvements in performance in our protocols, especially in the case of the flush protocol.

Another strategy to improve performance of DSM systems is to adopt a weaker consistency model so that blocking among some memory operations can be removed. In this thesis, we use the primitive notion of program-order and value-order to define global view. Using this as a seed, various consistency models evolve and form multiple hierarchies of models. The creation of these models and hierarchies comes via one of the following means: (i) a global view is augmented with additional ordering among its operations whenever some orderings exist, or (ii) besides linearizability of a global view, certain orderings must not co-exist in it. The former involves augmentation rules, and the latter involves causality requirements. The creation of these hierarchies leads to several novel consequences: the notion of exact implementation is introduced, new protocols are discovered and the precise analysis of access behaviors of an application is now possible.

Acknowledgements

I would like to begin by expressing my gratitude to my thesis supervisor, Dr. Hon F. Li. Without his patience, guidance and precious advice this research will not have been possible.

I would like to thank the Département de mathématiques et d'informatique de l'Université de Sherbrooke which has provided me with the proper equipment and software, which were necessary to complete this thesis. I would particuiarly like to thank M. Richard St-Denis who provided me with useful information and wrote the TEX macros that are used to write this thesis.

I would also like to thank the Université de Sherbrooke which has freed me from my teaching obligations for one year to help me in my research and for their financial support. This support has helped me to finally reach my research objectives and finish this thesis.

Table of Contents

List of Figures x

List of Tables xii

1 Introduction 1

Problem Statement 2

Contributions 4

Organization 7

2 Views and Access Graph 8

2.1 Distributed Application 9

2.2 The View Model 10

2.2.1 Local View 10

2.2.2 Partial and Global Views 13

2.2.3 Subset Restriction 15

2.2.4 Augmented Views 16

2.3 Access Graph 18

2.4 View Cycle and Access Cycle 22

3 Sequential Consistency 23

3.1 Views and SC 24

3.1.1 Necessary View 24

3.1.2 Possible View 29

3.1.3 Necessary Ordering vs Possible Ordering 31

3.1.4 Sequential Consistency 31
3.2 Algorithms 33
3.2.1 Execution Model 33
3.2.2 Virtual Access Cycle Based Synchronous Protocol 34
3.2.3 Flush Protocol 46
3.2.4 Multiple Flush Protocols 51
3.3 Conclusion 58

4 DSM Consistency Models Based on Global Views **61**
4.1 Minimal Consistency Based on Global View 61
4.2 Augmentation Rules and Consistency Models 62
4.2.1 Reachability Relations 63
4.2.2 Augmentation Rules 64
4.2.3 Consistency Models Based on Local Augmentation Rules 65
4.2.4 Consistency Models from Global Augmentation Rules 70
4.3 Weak Consistency Models Without Augmentation 76
4.3.1 Global Properties ($R_i \Rightarrow R_{\bar{j}}$) 79
4.3.2 Mixed Properties ($R_i^p \Rightarrow R_{\bar{j}}$) 80
4.3.3 Mixed Properties ($R_i \Rightarrow R_{\bar{j}}^p$) 80
4.3.4 Local Properties ($R_i^p \Rightarrow R_{\bar{j}}^p$) 81
4.4 Conclusion 82

5 DSM Consistency Models Based on Partial Views **84**
5.1 Restricted Views 84
5.1.1 Restricted Local Views 84
5.2 Consistency Models Based on Partial Views 87
5.2.1 Consistency Models using Object Restriction 88
5.2.2 Process Relative Consistency 89
5.2.3 Other Consistency Models 91
5.3 Conclusion 92

6 Exact Implementability of Weak Consistency Models **93**
6.1 Exact Implementation 94
6.2 Asynchronous Protocols 94
6.2.1 Multiple Vector Timestamps Protocol (MVTP) 95
6.2.2 Disjoint Version Number Protocol (DVNP) 96
6.2.3 Totally Ordered DVNP (TODVNP) 97
6.2.4 Extended Asynchronous Update Protocol (EAUP) 98
6.2.5 Logical Time Protocols (LTP) 99
6.2.6 Logical Clock Protocol (LCP) 102
6.2.7 Totally Ordered LCP (TOLCP) 103
6.2.8 Causal Update Protocol (CUP) 104
6.2.9 Vector Time Protocol (VTP) 105
6.2.10 Ahamad Vector Clock Protocol (AVCP) 107
6.3 Semi-synchronous Protocols 108
6.3.1 2-phase Write Protocol (2WP) 108
6.3.2 Invalidation Protocol 109
6.3.3 Causal Read Protocol (CRP) 111
6.3.4 Asynchronous 3-phase Protocol (A3P) 111
6.3.5 Direct/Indirect Vector Clock Protocol (DIVCP) 112
6.4 Almost Synchronous Protocols 113
6.4.1 Possibly Asynchronous Read (PAR) 113
6.4.2 Possibly Asynchronous Write Protocol (PAWP) 114
6.4.3 Fast-Read Three-Phase Protocol (FRTPP) 115
6.4.4 Ahamad's Owner Protocol for Causal Memory (OP) 117
6.5 Synchronous Protocols 119
6.5.1 Possibly Asynchronous Protocol (PAP) 119
6.6 Conclusion 120

7 Programmability of Weak Consistency Models **122**
7.1 Readers/Writers based restrictions 123

7.1.1 Single-reader Variables and Algorithms 123
7.1.2 Single-writer Variables and Algorithms 124
7.1.3 Single-writer/Single-reader Variables and Algorithms 126
7.2 Special Forms of Synchronization 126
7.2.1 Modeling of Barrier Synchronization 127
7.2.2 Implementation of Barrier Synchronization 127
7.2.3 Correctness Requirement (CR) of Barrier Synchronization Based Algorithms 128
7.3 Other Applications 129

8 Performance Evaluation of Neighbor and Flush Protocols 130
8.1 The Simulator 130
8.1.1 The Shared Memory Kernel Simulator 131
8.1.2 The Network Simulator 132
8.1.3 The Application Simulator 132
8.2 Analysis of Results 134

9 Implementation Considerations 139
9.1 Virtual Access Graph Construction and Analysis 139
9.1.1 Virtual Access Graph Construction 140
9.1.2 Neighbor Protocol 141
9.1.3 Flush Protocol 143
9.1.4 Choice of Synchronization 146
9.2 Conclusion 147

10 Conclusion 148
Contributions 148
Future Works 150

Bibliography 153

List of Figures

1 Sequence of memory operations by three processes 10
2 Views for example of Figure 1 11
3 An inadmissible local view 12
4 Global view 13
5 Global view that contains a cycle. 14
6 Partial view $PV_{\{1,2\}}$ 15
7 Examples of restricted views 17
8 An example of access graphs 19
9 Examples of directed access cycles 21
10 Sequentially consistent execution 24
11 Non-Sequentially consistent execution 24
12 Execution views 26
13 Examples of virtual cycles 27
14 A cycle in the necessary view 28
15 Acyclic necessary view without acyclic possible view 30
16 Access graph with 2 independent access cycles 42
17 Possible organizations for hierarchical flush events 59
18 Asynchronous Update Protocol (AUP) 62
19 Hierarchy of relations 64
20 GV_{13}^p and GV_{19}^p 66
21 Augmented views based on the guard R_3^p 67
22 Augmented views based on the guard R_4^p 67
23 Augmented views based on the guard R_6^p 68

24 Acyclic global view with a cyclic augmentation 68
25 Augmented views based on Rule R_1 71
26 Augmented views based on Rule R_3 72
27 Augmented views based on Rule R_4 72
28 Augmented views based on Rule R_6 72
29 Global hierarchy 74
30 An execution that violates $C_{4\overline{9}}$ 77
31 Sequence of memory operations by three processes 85
32 Local view LV_1 and some restricted local views 85
33 Necessary local view and its restricted views 87
34 Examples of object relative consistency 89
35 Examples of process relative consistency 90
36 Legal executions for protocol MTVP 96
37 Legal executions for protocol DVNP 97
38 Legal executions for protocol TODVNP 98
39 Executions that violate $C^p_{3\overline{8}}$ and $C^{-p}_{3\overline{8}}$ 99
40 Acyclic GV^p_{46} and cyclic GV^p_{49} 100
41 Cyclic GV^p_{19} allowed by LWTP 101
42 Cyclic GV_{19} allowed by VTP 106
43 Legal execution for protocol AVCP that violates C_{46} 108
44 Synthetic application's access graphs 133
45 Mutual exclusion's and dining philosopher's access graphs 134
46 Synthetic application simulation results (access graphs 1-6) 137
47 Synthetic application simulation results (access graph 7) 138
48 Mutual exclusion simulation results for two configurations 138
49 Dining philosopher simulation results 138
50 Example of virtual access graph 140
51 Access table 141
52 Synchronization tables for the neighbor protocol 144
53 Flush tables 145

List of Tables

1 Consistency models and hierarchy using program-ordered guards . . . 69
2 Consistency models using global guards 71
3 Consistency hierarchies using consequences 75
4 Weak consistency models and hierarchies based on global properties . 79
5 Weak consistency models and hierarchies based on local guards 80
6 Weak consistency models and hierarchies based on local restrictions . 81
7 Weak consistency models and hierarchies on locals guards and restrictions 82
8 Summary of some key examples 121

Chapter 1

Introduction

Today, computer networks are becoming commonplace. The potential computing power of all the processors connected to a particular network can be very important. This situation has created challenging opportunities in research in high performance distributed computing and many current research efforts are targeted toward developing techniques to exploit this computing power maximally.

Many existing distributed environments already provide facilities to efficiently share and use the multiple resources of computer networks. To fully exploit the distributed processing capabilities of a network, the distributed environment must also provide facilities for the user to write distributed applications. However, writing distributed applications is a difficult task. The user must specify all the independent computational units and how they interact.

Among the interaction facilities provided by the distributed environment, message passing is probably the most natural and efficient tool, but it is difficult to use. Indeed, with this tool, an application must explicitly manipulate all the state information. Moreover, the access between local and remote state is not uniform.

Several higher level facilities based on message passing have been developed to render its use easier. The remote procedure call (RPC) facility, which provides a procedure call semantics over the message passing system, is probably the most popular message passing abstraction. It is used to execute remote operations. Broadcast and multicast facilities, either normal, causal or atomic, are other higher level facilities

available in a message passing system.

The distributed shared memory (DSM) paradigm is another facility that can be provided by a distributed environment. In this paradigm, the environment provides the distributed application with the illusion of a global shared memory across multiple processors. This paradigm is becoming popular because it has advantages over the message passing paradigm. First, it is easier to use since it offers a uniform access to information. There is no need to use separate mechanisms to access local and remote data as in message passing. One can use the easy-to-follow shared variable paradigm to program distributed applications. As an example, many classical solutions for process synchronization were developed using the shared variable paradigm. Second, because of the absence of multiple address spaces, data partitioning and dynamic load balancing are simplified. Finally, many parallel programs written for shared memory multiprocessors can be ported easily to a distributed environment that uses the DSM paradigm.

Problem Statement

The DSM abstraction provides the programmer with a useful and simple model based on shared data with all the advantages mentioned earlier. However this comes at a performance cost. Indeed, in order to provide fast access to the shared data, multiple copies of the information are often maintained at different nodes. Moreover, the DSM system allows processes to access the same location simultaneously. Since the accesses are not instantaneous across the network, a consistency mechanism is required to guarantee that operations will appear in some ordering that is consistent under some condition. This condition, called a memory consistency model, is essential to allow programmers to use the shared memory correctly.

To maintain programmability, the memory consistency model should be intuitive and simple to use. Since the DSM abstraction provides the illusion of a centralized shared memory, it is natural to expect its behavior to be a simple extension of the centralized shared memory consistency model. A simple, widely accepted extension

to the centralized model, formulated by Lamport [41] for the multiprocessor case, is *sequential consistency* (SC) which is stated as follow:

> The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Unfortunately, implementing DSM systems that support sequential consistency is expensive. Indeed, all the suggested solutions [2, 4, 14, 18, 21, 46] require some form of blocking, which is expensive on networks that involve long access latencies. Blocking deters overlapping of operations. Processes in such systems are then slowed down considerably because their operations must be globally synchronized and executed locally in program order. So, the problem is to achieve efficient and scalable DSM systems.

There are several optimizations to reduce or hide synchronization delays. One is to minimize communication. This can be achieved by reducing the number of messages that must be exchanged to implement an operation. Another is to hide synchronization. This is achieved mostly by using non-blocking operations, non-atomic multicast facilities and out-of-order messages. Reducing the number of processes involved in a synchronization can also help.

Unfortunately, most of these optimizations involve memory consistency models that are weaker than sequential consistency and that are more difficult to use. Some of these models are based on the concept of data race free programs [3, 16, 25, 24, 34] in which only a subset of the operations are synchronized. They require the programmer to use special labels to guarantee consistency. Other models [8, 9, 10, 18, 23, 33, 44, 49] do not use special operations but are even more difficult to use in general. A program may be restricted to using specific algorithms that are known to execute correctly on such a model. To circumvent such a limitation DSM systems that support multiple consistency models are proposed [18]. Again, even in this case, the programmer must choose a proper model for its distributed application.

There is still controversy about the usefulness of weak consistency models. Hill [30] has suggested that the weaker models are too difficult to use and that sequential consistency is still the convenient model in multiprocessor systems.

Contributions

A challenging problem in DSM systems is to provide an abstraction that is both easy to use and efficient. The issues to study include: what consistency models should be supported? which consistency models have efficient implementation? and which consistency models can be easily used in an application? These issues involve either the development of a better implementation for sequential consistency or the development of weaker models that are as easy to use (or almost) as sequential consistency. Another possible solution is to develop a distributed application as if it were to execute on a sequentially consistent DSM system and for the environment to automatically detect the weaker consistency model that can guarantee sequentially consistent execution.

This thesis explores most of these issues. First, we introduce new optimized protocols for sequential consistency. Second, we present hierarchies of weak consistency models and develop implementations for some of them. These protocols are more efficient than SC protocols and still ensure SC execution under some restrictions.

The major contributions of this thesis are the following:

- We propose a new abstraction, called a view, for capturing consistency models. This abstraction differs from many existing models [23, 28, 29, 39, 46, 51] as it uses a logical ordering based on value coupling rather than the usual causal ordering. We can define such hierarchical consistency models by inferring additional logical ordering from the ones that already exist and/or by imposing some additional properties.

  The development of the view model was motivated by the fact that existing models based on the linearizability of a global execution history lacks the capacity to specify some weak consistency requirements. This is based on the fact

that the execution history always contains the global causality among events. It is then difficult to use them to specify weak consistency requirements such as slow memory or to determine which consistency requirements a particular protocol implements.

To develop a model to capture consistency requirements we have to use a different approach. We have based our model on the local perception of a process. This local perception is captured by the local program order and the values of variables read, without taking into account the global causality among events. By using these individual local perceptions, called local views, as building blocks, we can generate a taxonomy of consistency requirements. Hence, local views or different compositions of local views can be used to specify different consistency requirements.

Stronger consistency requirements are derived by adding global causality relations to form global views. With global causality, we are able to specify sequential consistency in terms of a strong union of local views. Hence, the separation between the local perception of a process and the global causality among events enables us to develop a powerful basis for modeling consistency requirements.

- We introduce an access graph to represent the static access restrictions of a shared object by the processes. It is different from the *conflict graph* [53] concept, which is dynamic.

- Using views and access graphs, we introduce two novel protocols for sequential consistency. These protocols exploit the information in an access graph to reduce the synchronization and hence improve performance. They are update-based, like the ones presented in [2, 4, 14, 13, 21]. However, the update does not always need atomic-broadcast or a 3-phase update protocol. Some protocols [18, 46] implement sequential consistency without the use of atomic broadcast or 3-phase protocols. However, they are invalidation-based protocols, which involve non-local read operations.

In most existing protocols, all update operations are blocking. However, some protocols [4, 46] have introduced some form of asynchrony on write operations issued by a single process. One of our protocols allows asynchronous operations not only between writes but also between write and read operations issued by a single process. All other protocols that allow asynchronous operations implement a relaxed memory model [2, 3, 13, 25, 34] in which only specially labeled operations are synchronized. None of our protocols use special labeling of information.

Simulations conducted with these algorithms show significant performance improvement over many existing SC implementations.

- Consistency hierarchies are proposed by starting from a minimal consistency model based on the global view of an execution. This view can be refined in many different ways using a hierarchy of refinement rules. These hierarchies are an original contribution of this thesis since no formal comparisons nor classifications are done as thoroughly as in our case. Indeed, even though some comparisons and classifications are proposed in the literature [19, 29, 33, 39, 47, 51], they are done only among existing models and the classifications are done in a rather ad hoc manner.

- Using views and the hierarchies of consistency models, it becomes easier to determine more precisely which model a particular protocol implements. We define *exact implementation* as an implementation that is strict enough to guarantee a model but not a stronger model in a consistency hierarchy.

- Using views and the hierarchies of consistency models, we show that many popular applications do not require sequential consistency to execute correctly. Similar work is done in [9, 36, 50] but only for the causal memory model. In this thesis, we determine more precisely the required consistency models for applications such as those using single-writer objects, barrier-synchronization and mutual exclusion.

Organization

The rest of this thesis is divided into ten chapters. Chapter 2 introduces the two key components in our models: views, which are used to capture consistency requirements; and access graphs, which are used to model the sharing restrictions among processes.

In Chapter 3, we present two optimizing protocols, the neighbor and flush protocols, that implement sequential consistency. This chapter first defines sequential consistency using the view model and then presents the abstract description of the protocols. Finally, the implementation of each protocol is described.

In Chapters 4 and 5, multiple hierarchies of consistency models are introduced. The creation of these models and hierarchies comes naturally via augmentation rules, which infer additional orderings among the operations whenever some orderings already exist. Chapter 4 presents the hierarchies derivable from the global view while Chapter 5 presents those derivable from the local and partial views.

In Chapter 6, the hierarchies of consistency models are used to define the concept of exact implementation. The rest of the chapter presents many protocols and the consistency models they implement or exactly implement.

In Chapter 7, the hierarchies of consistency models are used to study the consistency requirements of some distributed applications. This analysis is based on a characterization of the applications or on some specific synchronization.

In Chapter 8, we present the results of simulation experiments conducted to evaluate the efficiency of the neighbor and flush protocols. These experiments show significant performance improvement over conventional blocking protocols. Indeed, in some specific cases, the simulation shows that we can expect as much as 50% performance improvement.

In Chapter 9, we discuss how the implementation of the neighbor and flush protocols can be automated by a compiler.

Chapter 10 presents the conclusion.

Chapter 2

Views and Access Graph

The results of memory operations in a shared memory parallel program consist of a set of sequences of memory operations and their associated objects and values. Each sequence is program-ordered, i.e., in accordance to the order in which they are invoked in a sequential process. To identify the legality of an execution and hence the consistency semantics of shared memory, this chapter develops some primitive notions involving the individual (local) view of a process and the composite (global) view of all the processes together.

In an approximate abstraction, the local view of a process consists of its program ordered memory operations and the external writes whose values it has read. The global view is the direct composition (union) of the local views.

Both local and global views can be modified to generate other views. Two such modifications are introduced in this chapter. (1) A view can be modified using subset restriction (or projection). For example, a new view can be constructed by retaining only a subset of the operations and their order relations in a given view. (2) A view can also be modified using additional order relations. Such a modification is governed by augmentation rules. An augmentation rule stipulates that if a certain ordering exists in the view, then an additional ordering should also exist in the resultant view.

Shared memory consistency can be defined using various types of (global) view constructs. Two non-exclusive types of requirements can be asserted from a consistent global view. (1) The view should correspond to an acyclic graph (a partial order of

the operations). (2) In addition, we may require that the view does not contain some undesirable orderings.

In this chapter (Section 2.1 and 2.2), the various notions of views and their constructions are detailed. Their use in defining various weak consistency hierarchies are dealt with in later chapters.

Views are dynamic entities constructed from the execution. On the other hand, a process in a given parallel program may have certain static access restrictions to shared memory. For example, process P_i may write to only a subset of the shared objects. This access restriction is modeled by an access graph whose nodes correspond to processes and edges correspond to potential writer-reader relationships: if a process can read a value written by another process, then an edge leads from the latter to the former in the access graph. By capturing the access restrictions of a parallel program, it is possible to develop less expensive protocols to implement shared memory, which is the topic of Chapter 3.

In Section 2.3 and 2.4, the notion of access graph is formally introduced and the relationship between an access graph and a cyclic (inconsistent) global view is established by an important lemma. This formal result lays the foundation for protocol design in Chapter 3.

2.1 Distributed Application

A distributed system is formed by a set of cooperating processes exchanging information through communication channels. However, we define a distributed application as the abstraction of a distributed system that uses distributed shared memory rather than communication channels as the basis of cooperation. Distributed shared memory consists of a set of objects on which processes execute read and write operations. These objects could be replicated in the kernel implementation. In the following, we assume, unless explicitly stated, that each node maintains a local copy of each object. For convenience, we also assume that all copies are automatically initialized to the same state.

P_1 :	$x!2$; $y?1$; $y?2$; $x?1$; $y!3$
P_2 :	$x!1$; $y!2$; $x?1$
P_3 :	$y!1$

Figure 1: Sequence of memory operations by three processes

2.2 The View Model

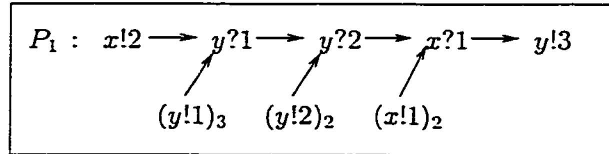
An execution of the system results in a set of linear traces, one per process. Each trace contains the sequence of program-ordered memory operations performed by the process, and the values associated with them. In particular, $(x!v)_i$ represents the writing of value v into object x by process i, and $(x?v)_i$ represents the reading of value v from object x by process i. For ease of explanation and without loss of generality, we assume the values written to an object are distinct. We may omit the process label i whenever the context is clear or the value of i is of no significance.

2.2.1 Local View

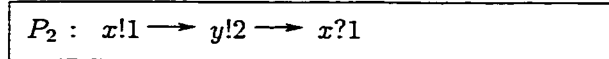
The memory operations associated with a process, say P_i, are invoked in program order, i.e., in the order in which they appear in the program. Process P_i observes changes in the shared memory objects through its own writing and reading of these objects. When it reads x, it may observe a value written by itself or another process. Hence there is a logical ordering among these related read/write operations. Taking all this into consideration, a local view of P_i, denoted by LV_i, can be defined. Before introducing the definition, a motivating example is presented.

Figure 1 shows a sequence of memory operations performed by three processes P_1, P_2, and P_3. Figure 2 gives (a) the local view LV_1 of P_1, (b) the local view LV_2 of P_2 and (c) the local view LV_3 of P_3. In these views, P_1 perceives its local operations in its program order. In addition, some write operations in P_2 and P_3, are ordered with respect to some operations in P_1. A logical ordering between $x!v$ and $x?v$ is introduced since a write should logically precede a read that returns the value written by it. As a consequence of this, LV_1 contains $(x!1)_2 \rightarrow (x?1)_1$, $(y!1)_3 \rightarrow (y?1)_1 \rightarrow (y?2)_1 \rightarrow (y!3)_1$

as well as $(y!2)_2 \rightarrow (y?2)_1 \rightarrow (y!3)_1$. We also observe that $(y!1)_3$ and $(y!2)_2$ are concurrent events. Two properties can be observed in LV_1: (i) LV_1 is a partial order of events relevant to the observation by P_1, and (ii) only write operations in other processes can be observed by P_1. LV_2 and LV_3 each contain only operations local to P_2 and P_3 respectively as they do not read any value written by other processes in the given sequence of operations.



a) Local view of process P_1 (LV_1)



b) Local view of process P_2 (LV_2)

c) Local view of process P_3 (LV_3)

Figure 2: Views for example of Figure 1

We are now ready to formalize the above notions of local view. We assume that in an execution, each process is represented by a sequence of memory operations it performs. Each memory operation is of the form $x!v$ or $x?v$. An execution is a set of such sequences.

Definition 1 : *Given an execution, the* **local view** LV_i *of* P_i *is constructively defined as a partially ordered set of events* $LV_i = (E_i, O_i)$ *such that:*

1. *for an event op, op* $\in E_i$ *iff op is a memory operation in* P_i*, or op is a write operation* $x!v$ *in another process* P_j *and* $x?v$ *is a read operation in* P_i*,*

2. *for two events* op_1 *and* op_2*,* $(op_1, op_2) \in O_i$ *iff*

   (a) event op_1 *appears before event* op_2 *in program order (denoted* $\xrightarrow{p}$*), or*

(b) event $op_1 = (x!v)_j$ *and event* $op_2 =$ *"earliest"* $(x?v)_i$ *in* P_i *(denoted* $\xrightarrow{v}$*).*

The notation $\rightarrow$ is used when the distinction between the two ordering relations is not important.

In Figure 2(a), the arrows drawn from P_2/P_3 to P_1 represent those orderings in O_i introduced by condition 2(b) of the definition.

Hence from the definition, the local view of a process can be interpreted as the minimal order perceived by the process, according to the local program order and the value order created when it reads the values written by other processes.

From the definition of the local view, we know that it represents a partial order of events. Moreover, we assume that all local views are *admissible*. A local view LV_i is admissible if for any read event $x?v$, there is no other event x!v'/x?v' between $x!v$ and $x?v$ in LV_i. The concept of admissible local view eliminates inconsistent observations by a single process. Figure 3 gives such an inconsistent observation not admissible as a local view. This example reveals that once a new value has been read, no old value of an object can any longer be read by the observing process. It will be assumed that an execution never produces inadmissible local views.

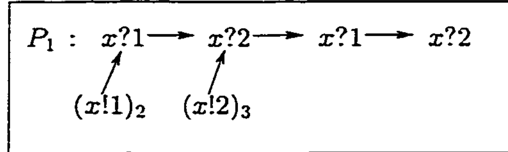


Figure 3: An inadmissible local view

In general, a partial order is represented in its transitively reduced form and we use $LV_i{}^*$ to represent the transitive closure of LV_i. LV_i^* represents all ordering relationships among the events.[1]

[1] A partial order can be visualized equivalently as a directed acyclic graph whose nodes represent the events and whose edges represent the transitively reduced O_i.

2.2.2 Partial and Global Views

In part or in whole, the local views of individual processes may be composed together to form partial and global views. The composition of a subset of the local views forms a partial view while the composition of all the local views forms a global view. The composition is by means of (disjoint) union of the local views. Suppose $LV_1 = (E_1, O_1)$ and $LV_2 = (E_2, O_2)$ are two local views, then their composition is $LV_1 \cup LV_2 = (E.O)$. such that $E = E_1 \cup E_2$ and $O = O_1 \cup O_2$. The composition of two or more local views reveals the global ordering among all memory operations as observed by the processes. Obviously, the ordering observed by one process need not exist in that of another. In the example of Figure 1, LV_3 contains a singleton whereas LV_1 and LV_2 contain multiple memory operations and their observed orderings by P_1 and P_2 respectively. Figure 4 gives the global view $GV = LV_1 \cup LV_2 \cup LV_3$.

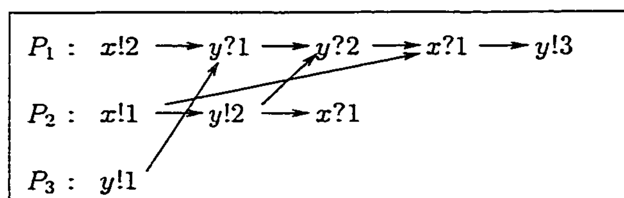
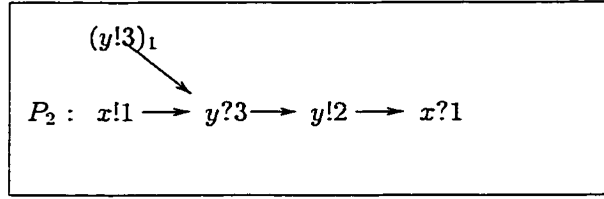


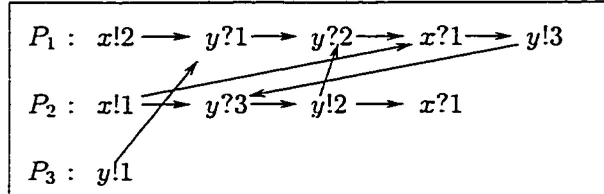
Figure 4: Global view

In Figure 4, the disjoint union of the local views forms a directed acyclic graph and the corresponding events are partially ordered, i.e., $LV_1 \cup LV_2 \cup LV_3$ is a partially ordered set represented by (E, O). In general, since the union operator does not preserve a partial order, the global view obtained may form a cyclic graph. In the latter case, we retain the graphical representation of the union, as the transitive closure of O no longer represents meaningful ordering relationships.

As an example of the latter scenario, consider modifying the execution in Figure 1 by appending an extra event, $(y?3)_2$ before $(x?1)_2$ in P_2. This will lead to the new local view LV_2', shown in Figure 5(a), and to the new global view $LV_1 \cup LV_2' \cup LV_3$, shown in Figure 5(b). The new global view contains the cycle $(y!2)_2 \rightarrow (y?2)_1 \rightarrow$



a) Local view of process P_2 (LV_2')



b) Global view $V_1 \cup V_2' \cup V_3$

Figure 5: Global view that contains a cycle.

$(x?1)_1 \rightarrow (y!3)_1 \rightarrow (y?3)_2 \rightarrow (y!2)_2$ and is no longer a partial order. Figure 12 in Chapter 3 gives another example of an execution sequence with its corresponding local (Figure 12(b)) and global (Figure 12(c)) views.

Again, to simplify the notation, we use $\rightsquigarrow$ to indicate ordering that can be direct ($\xrightarrow{p}$ or $\xrightarrow{v}$) or transitively induced via other operations. For the preceding example, $(y!2)_2 \rightsquigarrow (y!3)_1$ since the coupling is not direct.

Definition 2 : *A* **global view** *GV of an execution is defined as the union of the local views:* $\bigcup_i LV_i = (\bigcup_i E_i, \bigcup_i O_i) = (E, O)$.

The transitive closure O^* preserves transitivity but not necessarily the asymmetry. If O^* is asymmetric then GV is a partially ordered set of events. If it is not the case, GV becomes a cyclic graph.

The global view is used to represent the ordering of operations deducible from the program order and the direct coupling when a reader picks up the value written by a writer. If the global view is cyclic, then we conclude immediately that the execution is not sequentially consistent: there cannot exist a linear order of the operations that is consistent with individual program orders without contradicting the direct coupling between a writer and a reader. However, the converse is not true. Hence the global

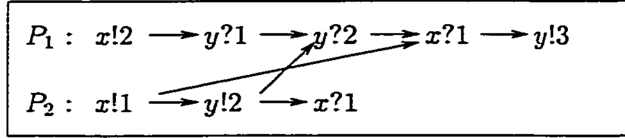


Figure 6: Partial view $PV_{\{1,2\}}$

view is used in Chapter 4 as a building block to define a family of weak consistency models.

Partial views are composed only of a subset of the local views. As an example, Figure 6 shows the partial view $PV_{\{1,2\}}$ for the execution shown in Figure 1. This partial view is obtained by the union of LV_1 and LV_2 and it represents the combined views of the execution of these two processes. The notation $LV_1 \cup LV_2$ is also used to represent a partial view and is particularly useful when projections are used as we will see later.

Definition 3 : *A* **partial view** *PV_s is obtained by the union of a subset s of the local views of the processes.*

As in the case of global view, partial views are also used as building blocks to define some weak consistency models. There are two basic mechanisms to derive other views from a partial or global view.

2.2.3 Subset Restriction

Subset restriction, which is often called projection [28], can be applied to a given view to construct a simpler one. The restriction is enforced on a subset of operations without changing the ordering relations among them.

Definition 4 : *The* **restriction** *of a (local, partial or global) view $V = (E, O)$ on a subset s of its nodes E is given by $V|_s = (E_s, O_s^*) = (E|_s, O|_s)$. $O^*|_s$ is the restriction of the transitive closure of O to s. If $O^*|_s$ is asymmetric, then O_s is simplified into its transitive reduction and $V|_s$ is treated as a partial order. In general, s can be chosen*

by (i) one or more of the processes, (ii) one or more of the objects, (iii) all write operations, or (iv) some combinations of the above.

A couple of illustrations will be appropriate here, using LV_1 in Figure 2, GV in Figure 4 and $PV_{\{1,2\}}$ in Figure 6 as examples.

Example 1 : *Consider a local view LV_i associated with process P_i. We define $LV_i|_{P_j}$ as the restriction of LV_i to the events of P_j, and hence include only those writes performed by P_j whose values are read by P_i. According to the definition of the local view, if $i \neq j$ then the ordering of these writes is not observed by P_i and hence $LV_i|_{P_j}$ contains unordered writes. Figure 7(a) shows $LV_1|_{P_2}$ for the example LV_1 in Figure 2.*

Example 2 : *Consider $LV_i|_{P_j,x}$ which is the local view of P_i restricted to the writes of P_j into object x. In this case, $LV_i|_{P_j,x}$ contains only a subset of writes in $LV_i|_{P_j}$ that involve x. Figure 7(b) shows $LV_1|_{P_2,x}$*

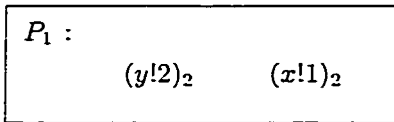
Example 3 : *Consider $LV_i|_w$ which is the local view of P_i restricted to write operations only. In this case, the read operations in LV_i are removed without destroying the ordering among the write operations. Figure 7(c) shows $LV_1|_w$.*

Example 4 : *Consider the global view GV shown in Figure 4. The restriction of GV on object y (denoted by $GV|_y$) is shown in Figure 7(d) and contains only those operations involving y. Since GV is a partial order, the restriction also preserves the partial ordering. Hence $GV|_y$ is drawn in its transitively reduced form.*

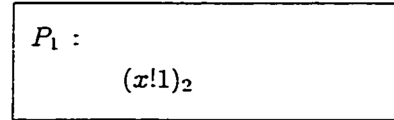
Example 5 : *Consider the partial view $PV_{\{1,2\}}$ shown in Figure 6. The restriction of $PV_{\{1,2\}}$ to operations involving y is drawn in Figure 7(e).*

2.2.4 Augmented Views

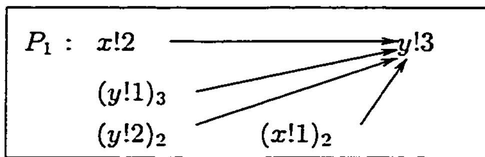
Augmented views are obtained by the application of augmentation rules on a given view. In this thesis, the view so involved is a local or a global view. An augmentation rule is expressed in the form of a guarded command.



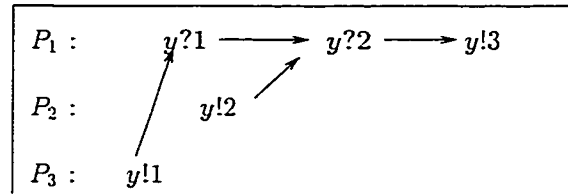
a) Restricted local view $LV_1|_{P_2}$



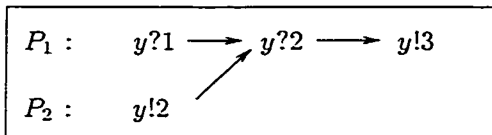
b) Restricted local view $LV_1|_{P_2,x}$



c) Restricted local view $LV_1|_w$



d) Restricted global view $GV|_y$



e) Restricted partial view $PV_{\{1,2\}}|_y$

Figure 7: Examples of restricted views

Definition 5 : *An* **augmentation rule** *"$op_1 \rightsquigarrow op_2 \Rightarrow op_3 \rightarrow op_4$" on a view $V = (E, O)$ results in a new view $V' = (E', O')$ such that :*

1. *$E' = E$, and*

2. *$O' = O \cup \{$ if $(op_1, op_2) \in O^*$ then $(op_3, op_4)\}$*

$op_1 \rightsquigarrow op_2$ is called the **guard** *and $op_3 \rightarrow op_4$ is called the* **consequence** *of the rule. For generality, the guard can be restricted to program-order, i.e., op_1 and op_2 belong to the same process ($op_1 \xrightarrow{p} op_2$).*

As an example, the augmentation rule $(x?v)_i \xrightarrow{p} (x?v')_i \Rightarrow (x?v)_i \rightarrow (x!v')_j$ means that whenever a process reads two different values of a same object x then the first read is ordered before the write of the second value, possibly at another process. Hence even if $(x?v)_i$ and $(x!v')_j$ are not ordered in O, they are made ordered in O'.

A global view obtained by applying an augmentation rule r is denoted GV_r. Chapter 4 presents the set of useful augmentation rule to generate interesting consistency models.

2.3 Access Graph

The shared memory objects are not uniformly shared among all processes. Indeed, they need not be writable and readable by all processes. An important consequence of this is that the resulting kernel implementation of the shared memory objects may require less synchronization and thus may incur less runtime latency. An access graph is used to capture read/write restrictions of a shared memory object by the distributed processes.

Figure 8(a) shows a simple example of three processes sharing three objects, x, y and z. Object z can be read by all three processes but written only by P_1 and P_3. Object x can be written by P_1 and read by P_2. Object y can be written by P_2 and read by P_3. In the access graph, nodes represent processes and edges represent objects. An edge labeled x leading from P_i to P_j means that P_i can write x and that P_j can read x.

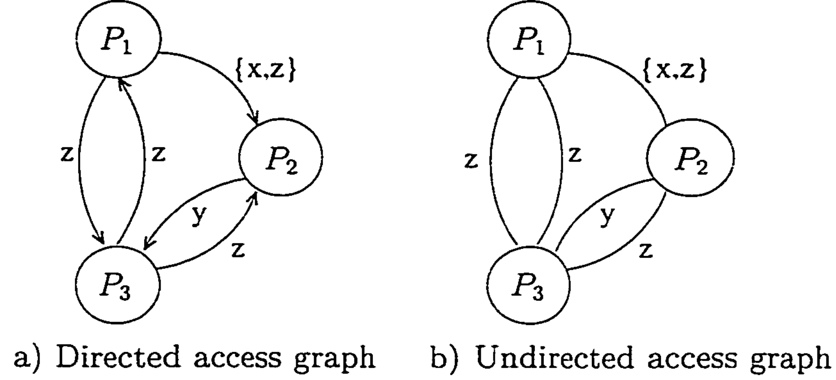


Figure 8: An example of access graphs

Definition 6 : *A* **directed access graph** *is a labeled directed graph* $G = (N, A, L)$*, such that*

1. *N is a set of nodes $\{P_1, ..., P_n\}$;*

2. *A is a set of directed access edges in $N \times N$;*

3. *$L(A)$ is a subset of the memory objects O such that a memory object, say x, is in $L(P_i, P_j)$, iff P_i is a writer and P_j is a reader of x.*

Definition 7 : *An* **undirected access graph** *is obtained by removing the direction of all edges in a directed access graph.*

Figure 8(b) shows an undirected access graph.

In an undirected access graph, an edge labeled by object o with $o \in L(P_i, P_j)$ is formed either by P_i writing into object o and P_j reading the value written or by P_i reading o followed by P_j writing a new value into o. Because of this, an edge in the undirected access graph is called a direct access edge.

Definition 8 : *A* **direct access edge** *exists between two processes P_i and P_j of the undirected access graph G, if there exists an object o such that $o \in L(P_i, P_j)$.*

We write $P_i \xrightarrow{o} P_j$ to denote that P_i can write into object o and P_j can read from object o. We use $P_i \overset{o}{-} P_j$ instead of $P_i \xrightarrow{o} P_j$ or $P_i \xleftarrow{o} P_j$ when the direction of the

direct access edge is unimportant. When the object name is unimportant, the label "o" will be omitted.

A cycle in the undirected access graph is called an *access cycle.* It is formed by a sequence of direct access edges starting from some process and ending in the same process, without repeating any process in between.

Definition 9 : *An* **access cycle** *is a cyclic sequence of direct access edges of the form* $\{P_1 \overset{o_1}{-} P_2 \overset{o_2}{-} ...P_n \overset{o_n}{-} P_1\}$[2] *with* $n > 1$*, in which no edge is repeated and where* $P_i \neq P_j$ *when* $i \neq j$.

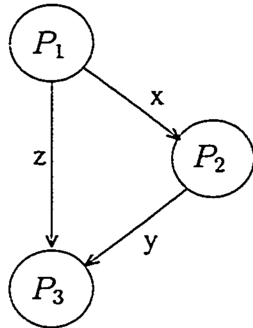
Definition 10 : *A* **directed access cycle** *is an access cycle with directed edges.*

An undirected access cycle can correspond to multiple directed access cycles. Figure 9 illustrates all directed access cycles involving multiple objects for the directed access graph of Figure 8. Figure 9(a) and (b) shows two distinct directed access cycles that correspond to a single undirected access cycle. Figure 9(a) differs from Figure 9(b) in that P_1 writes into z, which is read by P_3 in the former but which is written by P_3 in the latter. Figures 9(d), (e), and (g) represent unconventional cycles in which object z is accessed more than once in the cycle. It is noteworthy that the notion of access cycle allows an object but not a process to be repeated in the cycle.

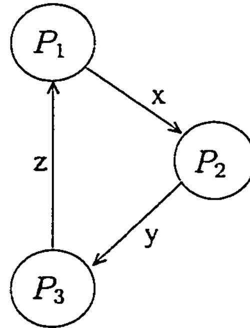
Definition 11 : *An edge in the access graph is* **asynchronous** *if it does not lie in an access cycle. Otherwise, it is* **synchronous**.

The usefulness of the access graph lies in the minimization of synchronization delay in each process. Indeed, the synchronization needs of a distributed application depend on the underlying access graph involved. Particularly, we show, in Chapter 3, that in order to maintain consistency, we can focus principally on the synchrony of operations involved in an access cycle. As an example, we show that if every access cycle and some other simple constructs to be defined later are properly "synchronized", sequential consistency is guaranteed.

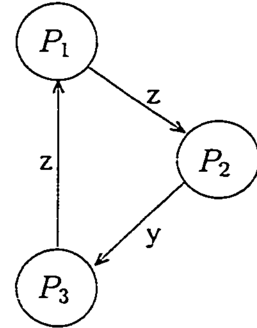
[2] *Normally, we should use two functions f and g such that a cycle is represented by* $\{P_{f(1)} \overset{o_{g(1)}}{-} P_{f(2)} \overset{o_{g(2)}}{-} ... P_{f(n)} \overset{o_{g(n)}}{-} P_{f(1)}\}$ *where* $P_{f(i)} \neq P_{f(j)}$ *when* $i \neq j$*. However, without loss of generality we abbreviate both* $f(i)$ *and* $g(i)$ *to* i*.*



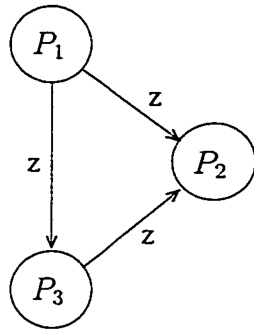
a) $P_1 \xrightarrow{z} P_3 \xleftarrow{y} P_2 \xleftarrow{x} P_1$



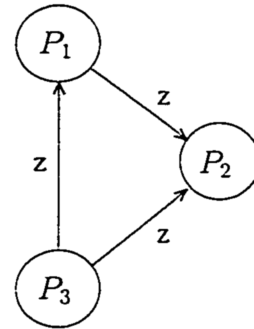
b) $P_1 \xleftarrow{z} P_3 \xleftarrow{y} P_2 \xleftarrow{x} P_1$



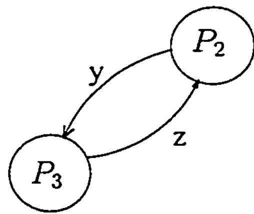
c) $P_1 \xleftarrow{z} P_3 \xleftarrow{y} P_2 \xleftarrow{z} P_1$



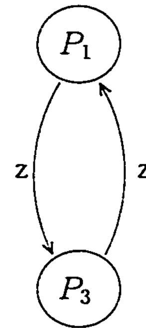
d) $P_1 \xrightarrow{z} P_3 \xrightarrow{z} P_2 \xleftarrow{z} P_1$



e) $P_1 \xleftarrow{z} P_3 \xrightarrow{z} P_2 \xleftarrow{z} P_1$



f) $P_3 \xleftarrow{z} P_2 \xrightarrow{y} P_3$



g) $P_1 \xrightarrow{z} P_3 \xrightarrow{z} P_1$

Figure 9: Examples of directed access cycles

2.4 View Cycle and Access Cycle

The construction of global view by the union of local views may create cycles in the global view. This is equivalent to $GV = (E, O)$ in which O is not asymmetric. There is a relationship between a view cycle and the associated access graph. Specifically, the existence of a view cycle also reveals the existence of a corresponding access cycle in the access graph. Since a global view that is not asymmetric cannot possibly be linearized, i.e., the operations cannot have a total order consistent with the global view, the absence of view cycle is inherently important to get consistent executions. The shared memory protocol should then be designed to avoid the occurrence of view cycles. This is the subject of later chapters. In this section, we state and prove the following important lemma.

Lemma 1 : *The existence of a cycle in a global view implies the existence of an access cycle in the associated access graph.*

Proof:

Consider a view cycle of the form:

$$op_1^1 \xrightarrow{p} op_2^1 \rightarrow op_2^1 \xrightarrow{p} op_2^2 \rightarrow \ \ldots \ \rightarrow op_1^i \xrightarrow{p} op_2^i \rightarrow \ \ldots \ op_1^k \xrightarrow{p} op_2^k \rightarrow op_1^1$$

where op_1^i and op_2^i are the first and last operation in process P_i contained in the cycle. Without loss of generality, the above representation assumes that the processes traversed in the view cycle are processes $1, 2, \ldots$, and k.

According to the construction of local view, $op_2^i \rightarrow op_1^{i+1}$ must be due to the value order caused by process P_{i+1} in op_1^{i+1} reading the value written by process P_i in op_2^i. By definition, this means that there is a direct access edge $P_i \xrightarrow{x} P_{i+1}$ between the two processes. Hence, by construction, the above events of the view cycle exactly trace a series of direct access edges in the access graph, starting from P_1 and ending at P_1, of the form: $P_1 \overset{o_1}{-} P_2 \overset{o_2}{-} \ldots \overset{o_{n-1}}{-} P_n \overset{o_n}{-} P_1$.

■

Chapter 3

Sequential Consistency

The most commonly assumed memory consistency model for DSM systems is sequential consistency (SC). This model provides a clear semantics for the execution of memory operations. As a consequence, programming under this model is relatively easy.

Taken literally, the definition of SC provided in the introduction implies that an algorithm that implements SC must maintain program order among operations from a single processor and a single sequential order among all operations. This last condition makes a memory operation appear to execute atomically with respect to other operations. In the context of DSM, sequential consistency is re-defined more clearly as follows:

Definition 12 : *An execution is* **sequentially consistent** *iff there exists a total ordering of all the operations such that (i) it is consistent with each program order, and (ii) in the ordering, $x!v$ must appear before $x?v$ and there are no other operation $x!v'$ or $x?v'$ appearing between $x!v$ and $x?v$.*

Figure 10 shows a sequentially consistent execution since there exists for all operations a total order that satisfies the previous conditions. One such total order for this execution is $(x!1)_1; (x?1)_1; (y!1)_2; (y?1)_3; (z!1)_1; (z?1)_2; (x!2)_3; (x?2)_3; (x?2)_1$. Figure 11 shows an execution that is not sequentially consistent since there is no sequential order that satisfies the previous definition.

P_1 :	$x!1 \; ; \; z!1 \; ; \; x?2$
P_2 :	$x?1 \; ; \; y!1 \; ; \; z?1$
P_3 :	$y?1 \; ; \; x!2 \; ; \; x?2$

Figure 10: Sequentially consistent execution

P_1 :	$x!1 \; ; \; z!1 \; ; \; x?2$
P_2 :	$x?1 \; ; \; y!1 \; ; \; z?1$
P_3 :	$y?1 \; ; \; x!2 \; ; \; x?1$

Figure 11: Non-Sequentially consistent execution

In this chapter, we formulate the corresponding conditions of sequential consistency in the view model. The concepts of necessary and possible views are introduced. We also present some new algorithms for sequential consistency. These algorithms aim at minimizing synchronization by exploiting the knowledge represented in access graphs.

3.1 Views and SC

The global view, presented in the preceding chapter, is used to represent the ordering of operations deducible from the program order and also the direct coupling when a reader picks up the value written by a writer. If the global view is cyclic, we conclude immediately that the execution is not sequentially consistent as defined earlier, i.e., there cannot exist a linear order of the operations that is consistent with individual program orders without contradicting the direct coupling between a writer and a reader. However, the converse is not true.

3.1.1 Necessary View

The coupling between readers and writers must also ensure that every object is atomic through indirect coupling among the processes. Hence, there are additional orderings that must be satisfied in the global view, and they are captured by the following augmentation rule.

Rule 1 : $(x!v)_i \leadsto (x?v')_{j'}/(x!v')_j \Rightarrow$

$(x?v)_k \rightarrow (x!v')_j$ for each process P_k *that contains* $(x?v)_k$ *and*

$(x!v)_i \rightarrow (x!v')_j$ if the value v is never read.

Rule 1 orders every read or the write of the value v to appear before the write of another value v' into x, if the write of the former value is ordered before some read or the write of the latter value.

Definition 13 : *A* **necessary view** *(NV) is a global view augmented with Rule 1.*

Figure 12 shows an example of execution and its corresponding local, global and necessary views. The necessary view includes $y?3 \rightarrow y!2$ because $y!3 \leadsto y!2$.

The necessary view is the maximal ordering among the operations that can be deduced from an execution. If the necessary view is cyclic, by the same reasoning as in section 2.2.2, the execution is not sequentially consistent. Unfortunately, the converse is still not true. However, it is easily verifiable that the necessary view can be derived from an execution in polynomial time and is unique up to transitive closure (i.e. reachability relation between nodes) in the representation.

Lemma 2 : *The reachability relation in any necessary view for a given global view is independent of the order in which augmentation is performed.*

Proof: This follows immediately from the stability of the guard (the left-hand side) of Rule 1. The satisfaction of the guard is not affected by other augmentations. Hence the claim. ■

Lemma 2 allows us to augment a global view in arbitrary order, and this facilitates proofs that use constructive arguments.

Moreover, in Lemma 1, we show that the presence of a view cycle in the global view implies the presence of an access cycle in the access graph. This is an important result that we must extend to necessary views in order to make use of access graph to reduce synchronization among processes.

However, with the application of the augmentation rule, cyclic views involving a single access edge are possible in the access graph. Indeed, the application of the

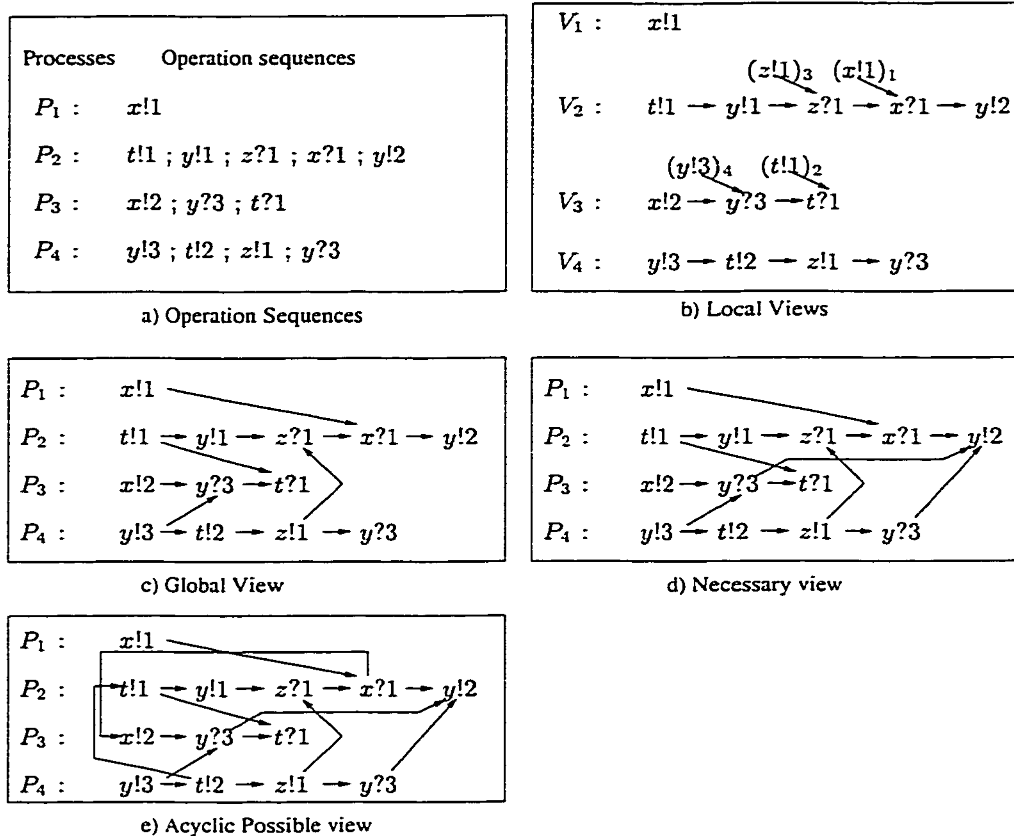


Figure 12: Execution views

augmentation rule may introduce a direct precedence relation between processes that are not directly connected in the access graph, creating what we call a virtual access edge. This virtual access edge, labeled with the shared object (o_i), introduces a direct precedence relation between two write events (and possibly two read events with different augmentation rules) in the access graph. For ease of explanation, we call the object labeling this virtual access edge a virtual object. An access graph augmented with these virtual access edges is called a virtual access graph. Every cycle in the virtual access graph is called a virtual cycle.

Definition 14 : *A* **virtual access graph** *is formed by an access graph augmented with virtual access edges between two conflicting writers into an object that has a single reader.*

In other words, if P_1 and P_3 can write into x and P_2 is the only process that can read x, then we add a virtual access edge between P_1 and P_3.

Definition 15 : *A cycle in the virtual access graph is called a* **virtual cycle**.

A virtual cycle can be either an access cycle or a new cycle introduced by the virtual access edges. Figure 13 shows examples of virtual access graphs that contain virtual cycles that were not present in the access graph. In each case, the special sequence $P_1 \xrightarrow{z} P_2 \xleftarrow{z} P_3$ is found and the virtual access edge labeled z is illustrated with a dashed edge. In Figure 13(a) we have a virtual cycle formed by a virtual access edge that implies the repetition of a single edge in the sequence. In Figure 13(b) we have a virtual cycle formed by a virtual edge that implies the repetition of two access edges in the sequence. Finally, in Figure 13(c) we have a virtual cycle that is formed by completely distinct edges.

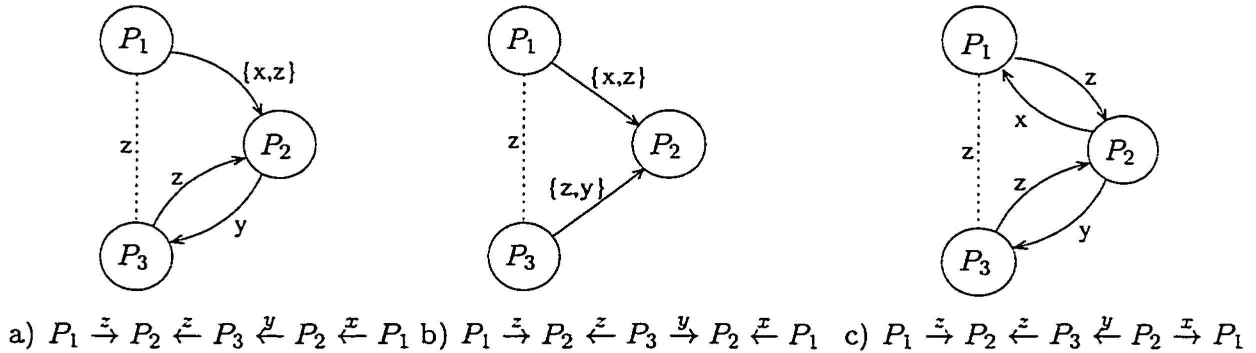


Figure 13: Examples of virtual cycles

The new ordering introduced by the application of Rule 1 implies that program order must be preserved among operations which involve a single edge and among operations on a virtual cycle. The particularity of the new cycles introduced by a virtual access graph is that they are a combination of two edges, a single edge with an access cycle, or two access cycles. Hence, their proper synchronization requires only to enforce program order in the repeated process. In Figure 13 program order

must be enforced in P_2 between x and y because they are both involved in a virtual access cycle with z.

Suppose a cycle in a view involving P_1 is obtained. Let the first and last events in the cycle of P_1 be the *in* and *out* events of P_1 respectively. Without loss of generality, suppose the event in the cycle that follows the *out* event of P_1 is in P_2. The latter is now the *in* event of P_2. The last event of P_2 in the cycle is the *out* event of P_2. This repeats with other processes until the *in* event of P_1 is traced in the cycle. Hence we define a view cycle to consist of *critical events* labeled as: $op_1^1 \rightarrow op_2^1 \rightarrow op_1^2 \rightarrow op_2^2 \rightarrow ... \rightarrow op_1^i \rightarrow op_2^i \rightarrow ... \rightarrow op_1^k \rightarrow op_2^k \rightarrow op_1^1$ where op_1^i and op_2^i are the *in* and *out* events of process P_i in the cycle. Notice that because of the use of Rule 1, the *in* and *out* events of a process may be the same in a cycle of a necessary view, unlike that in a global view. Figure 14 shows such an example.

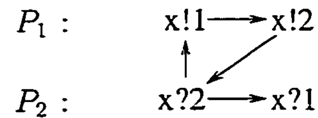


Figure 14: A cycle in the necessary view

Suppose $op_1^1 \leadsto op_2^k$ is a path in a view from op_1^1 to op_2^k. Without loss of generality, assume op_1^1 is in P_1 and op_2^k is in P_k, and op_2^1 is the last event in P_1 and op_1^k is the first event in P_k in the path.

Definition 16 : *The* **critical events** *of the path $op_1^1 \leadsto op_2^k$ consists of events (op_1^1; op_2^1; op_1^2; op_2^2; ; op_1^k; op_2^k) such that the op_1^i is the first and op_2^i is the last event in process i in the path. op_1^i and op_2^i may not be distinct events.*

Definition 17 : *Two processes, P_i and P_j, are* **synchronous** *if they are connected by an access edge that lies in an access cycle.*

Definition 18 : *$(x!v)_i$ and $(x?v')_j$ is a synchronous read/write (conflict) pair if P_i and P_j are synchronous.*

Definition 19 : *Two writes, say* $(x!v)_i$ *and* $(x!v')_j$*, are synchronous if there exists a reader process, say* P_k*, such that* (P_i, P_k) *and* (P_k, P_j) *are synchronous process pairs. Given these two synchronous pairs,* $(x?v)_k$ *and* $(x!v')_j$ *form a synchronous read/write conflict.*

Definition 20 : *If all the critical events* op_2^i *and* op_1^{i+1} *of a path are synchronous read/write or write/write conflicts, then the path is called* synchronous. *Otherwise it is* asynchronous.

Lemma 3 : *The sequence of critical events in a cycle in a necessary view correspondingly traces a virtual cycle or a single edge traversed in both directions.*

Proof:

$op_1^i \rightarrow op_2^i$ follows the program order of process P_i. But $op_2^i \rightarrow op_1^{i+1}$ corresponds to some x in the form of:

(i) $x!v \rightarrow x?v$ (from the global view), or

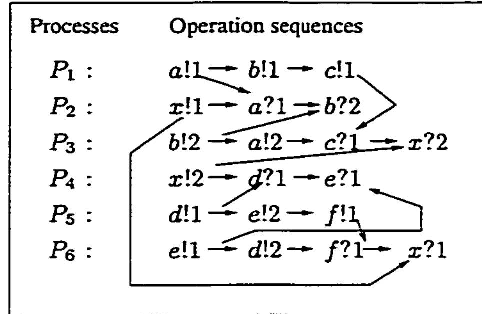
(ii) $x?v \rightarrow x!v'$ (from augmentation Rule 1).

(iii) $x!v \rightarrow x!v'$ (from the ordering of writes if v is never read).

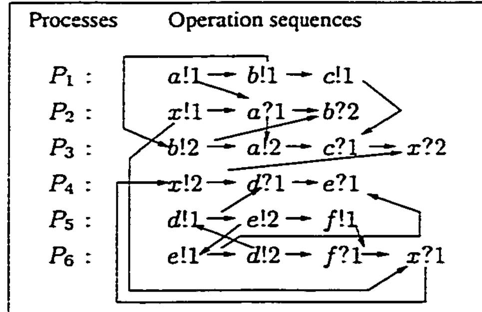
In the first two cases, an access edge between P_i and P_{i+1} is traced. From the third case, a path between P_i and P_{i+1} is traced via a reader P_k. This path is composed of two access edges $P_i \overset{x}{-} P_k \overset{x}{-} P_{i+1}$. Hence the cyclic sequence of critical events exactly traces either a virtual cycle or a single edge (traversed in both directions) in the access graph. In the latter case, the trace involves $P_1 \overset{x}{\rightarrow} P_2 \overset{x}{\leftarrow} P_1$ or $P_1 \overset{x}{\leftarrow} P_2 \overset{x}{\rightarrow} P_1$. Thus the claim. ■

3.1.2 Possible View

For an execution to be sequentially consistent, the acyclic necessary view must satisfy an additional property. Suppose two writes, say $(x!v)_i$ and $(x!v')_j$ at least one of whose values has been read by some process, are unordered in the acyclic necessary view. These writes are called concurrent writes.



a) Acyclic Necessary View



b) A Cyclic Possible View

Figure 15: Acyclic necessary view without acyclic possible view

Definition 21 : *A* **possible view** *is obtained from the necessary view by ordering every pair of concurrent writes, say* $(x!v)_i$ *and* $(x!v')_j$*, such that* $(x!v)_i \rightarrow (x!v')_j$ *implies every* $(x?v)_k \rightarrow (x!v')_j$

These selections are arbitrary and lead to different possible views. The existence of an acyclic possible view is related to sequential consistency as we prove in Section 3.1.4.

Figure 12(e) shows an acyclic possible view of the execution given in 12(a). We show in Section 3.1.4 that this implies the execution is sequentially consistent. Figure 15(a) shows another execution which has an acyclic necessary view but which does not possess an acyclic possible view. Figure 15(b) shows a cyclic possible view for this execution. To establish this possible view, we use one possible ordering between all writes on a single object. Other possible orderings exist but they all create a cycle in the possible view. Thus it is not sequentially consistent.

Lemma 3 establishes the correspondence between a cycle in a necessary view (hence inconsistency) and a virtual cycle or a single edge. This correspondence also applies to a possible view, as we prove here.

Lemma 4 : *The sequence of critical events in a cycle in a possible view correspondingly traces a virtual cycle or a single edge traversed in both directions.*

Proof: Following the same strategy as in the proof of Lemma 3, we show that the coupling $op_2^i \rightarrow op_1^{i+1}$ must be of the form:

(i) $x!v \rightarrow x?v$ (from the global view),

(ii) $x?v \rightarrow x!v'$ (from necessary or possible view), or

(iii) $x!v \rightarrow x!v'$ (from necessary or possible view).

In each of the above, it traces an access edge or a virtual access edge between P_i and P_{i+1}. Hence, the critical events correspond to a virtual cycle or a single edge traversed in both directions. ■

3.1.3 Necessary Ordering vs Possible Ordering

It is shown in [26] that deciding if an execution is sequentially consistent is an NP-complete problem. Hence given an arbitrary execution, even though we can easily derive the necessary ordering and hence the necessary view (in polynomial time), we do not think it is possible to derive an acyclic possible view without enumeration. So the necessary ordering is the maximal information we can deduce which must exist among the operations in order that the read/write semantics of memory objects are not violated. The possible ordering includes runtime choices which are not deducible from the execution trace and are often not unique.

3.1.4 Sequential Consistency

In this section, we prove that an execution is sequentially consistent iff the corresponding possible view is partially ordered.

Lemma 5 : *An execution is sequentially consistent iff it possesses an acyclic possible view.*

Proof:

($\Rightarrow$) Given an acyclic possible view, we can obtain a total order of the execution by iteratively selecting an operation, among the subset of operations which are not preceded by other operations in the remaining possible view, as the next operation in the total order. Because of the property of the possible view, this total order satisfies the requirement of Definition 12.

($\Leftarrow$) The reverse is immediate as the total order is an acyclic possible view itself. ■

Lemma 5 clearly indicates that a distributed protocol correctly implements any sequential consistency defined with views only if it guarantees that every execution has an acyclic possible view.

Moreover, from Lemmas 3 and 4, we have shown that a cycle in a necessary or possible view corresponds to a virtual cycle or an edge traversed in both directions. To get rid of view cycles, we can design a protocol to ensure their non-occurrence based on our knowledge of virtual cycles. It follows that synchronizing accesses in every cycle may give us sequential consistency. To demonstrate this possibility, we use the projection of views as presented in Section 2.2.4. We use a global view restricted to the processes and objects involved in a particular cycle.

Any virtual cycle c can be represented by a set of processes, $\{P_1, P_2, ..., P_k\}$, and a set of objects, $\{o_1, o_2, ..., o_m\}$, contained in the cycle.

Theorem 1 : *For every virtual cycle or access edge c, $U_i(LV_i|_c)$ has an acyclic possible view iff $U_i(LV_i)$ has an acyclic possible view.*

Proof:

($\Rightarrow$) Suppose all possible views of $U_i(LV_i)$ are cyclic. Then in each possible view, there is a cyclic sequence of critical events involving $P_1; P_2; \ ... \ ; P_1$. From Lemma 4, this traces a virtual cycle or a single edge traversed in both directions in the virtual access graph. The same events form a cycle in $U_i(LV_i|_c)$.

($\Leftarrow$) The reverse is immediate from the fact that $(LV_i|_c)$ is a subset of (LV_i). Hence an acyclic possible view from LV_i leads to the same from $LV_i|_c$.

■

3.2 Algorithms

The definition of sequential consistency in terms of views is very important in order to develop and prove new algorithms for sequential consistency using this tool. So, a correctly designed distributed protocol for sequential consistency in shared memory must ensure that at least one acyclic possible view exists, or that view cycle cannot occur. This can be achieved in a number of ways. Different approaches lead to two different protocols. In this section we introduce some new algorithms. The first two algorithms are based on blocking protocols and they use the access graph concept to reduce synchronization. The others that follow are extensions of the second protocol.

3.2.1 Execution Model

The view space presented earlier captures only abstract read and write events occurring in the shared memory system. It is too abstract to be used to capture the runtime details of the distributed protocol which implements the shared memory. Another model, the execution model, is introduced to capture the runtime details corresponding to events in the view model.

The execution space contains events that occur in the underlying message passing distributed system. We associate with each operation (event) op_i in the view space two events in the execution space: the start $(st_(op_i))$ and the end $(end_(op_i))$ event. Events in the execution space are related by the happens-before ($\mapsto$) relation introduced by Lamport[40]. For obvious reasons, it is assumed that for each operation (event) op_i, $st_(op_i) \mapsto end_(op_i)$.

The correspondence between the view precedence relation and the happens-before relation is established by a set of conditions. These conditions specify for each protocol the required correspondence between ordering relations in the view space ($\rightarrow$) and

ordering relations in the execution space ($\mapsto$). As an example, the following condition can be used in such a set: $op_i \rightarrow op_j \Rightarrow end_(op_i) \mapsto end_(op_j)$. The protocol that implements a distributed shared memory system tries to enforce every condition specified in such a set.

The *start* and/or *end* events of the execution space are synchronization points that can be mapped to the events in the abstract view space. These synchronization points are established by identifying which statements in the implementation correspond to the *start* and *end* of the operation implemented.

3.2.2 Virtual Access Cycle Based Synchronous Protocol

A synchronous protocol (also called neighbor protocol) avoids the cycles in a possible view by enforcing some set of conditions as mentioned in section 3.2.1. The following set of conditions specify, for the neighbor protocol, the correspondence between the ordering relations in the view space and the ordering relations in the execution space.

Condition 1 : PO - *Program Order.*

If op_i and op_j are two instances of operations by a process such that they are associated with a same edge or two edges that lie in a same virtual cycle, then we require: $op_i \xrightarrow{p} op_j \Rightarrow end_(op_i) \mapsto end_(op_j)$

In other words, PO ensures that if op_i and op_j are program ordered and satisfy the above condition, then their ending must follow the happens-before order as well. The latter happens-before order is useful to eliminate view cycles and hence achieving consistency. Under PO, operations that do not lie in a same edge or virtual cycle do not have to be ordered at runtime (happens-before order). Moreover, it also means that $x!v \xrightarrow{p} x!v' \Rightarrow end_(x!v) \mapsto end_(x!v')$ and $x?v \xrightarrow{p} x?v' \Rightarrow end_(x?v) \mapsto end_(x?v')$. This case is used also to avoid view cycles that can involve a single edge.

Condition 2 : CO - *Conflict Order in a Shared Object*

CO models the atomicity of each shared object so that conflicting accesses are serialized properly.

1. *Writing of a value must precede any reading of that value:* $end_(x!v) \mapsto end_(x?v)$ *for every* $x?v$.

2. *Synchronization of operations associated with a synchronous edge:*

   (a) *Consider* $P_1 \xrightarrow{x} P_2$ *being a synchronous pair (edge), and* $(x!v)_1$ *and* $(x?v')_2$ *operations performed by* P_1 *and* P_2. *Notice that* $(x!v')$ *could have been performed by some process other than* P_1, *say* P_3, *and* $P_3 \xrightarrow{x} P_2$ *can be asynchronous. We require either:*

   - $end_(x!v)_1 \mapsto end_(x?v')_2$ *or*
   - $end_(x?v')_2 \mapsto end_(x!v)_1$.

   In other words, the reader P_2 *must be locally consistent with itself. For example, in the former case,* P_2 *must not allow* $x!v'$ *to happen before* $x?v$ *and at the same time* $x!v$ *before* $x!v'$.

   (b) *Consider* $P_1 \xrightarrow{x} P_2$ *and* $P_2 \xleftarrow{x} P_3$ *being two synchronous process pairs, and* $(x!v)_1$, $(x!v')_3$ *and* $(x?v)_2$ *are operations on those processes. Then a stronger condition than CO(2a) applies. Specifically, we require the two writes by* P_1 *and* P_3 *to be ordered in the happens-before world (for example, ordering at* P_2 *transitively also asserts the ordering on these writes):* $end_(x!v)_1 \mapsto end_(x!v')_3$ *or* $end_(x!v')_3 \mapsto end_(x!v)_1$.

3. $end_(x!v)_1 \mapsto end_(x?v')_2 \Rightarrow \neg(end_(x!v')_3 \mapsto end_(x?v)_2/end_(x!v)_1)$.

   This condition enforces the consistency in the observation of events in a process.

In essence, condition CO(2) ensures that every pair of conflicting operations (read/write or write/write) on a same object and lying on a same virtual cycle are ordered in the protocol space. This means that as soon as for two conflicting operations (events) on an object x, $op_1 \leadsto op_2$, this implies that $end_(op_1) \mapsto end_(op_2)$ for all processes. Condition CO(3) combined with PO ensures the same condition on asynchronous edges. Together, CO(3) and PO specify FIFOness update which is required on asynchronous edges. PO and CO are collectively called the *neighbor invariants*.

We claim that the neighbor invariants (NI) guarantee sequential consistency. The proof is not simple. Hence, we prove this claim by a sequence of proofs.

Lemma 6 : *GV is acyclic under NI.*

Proof: Suppose otherwise, there is a cycle in GV. Since GV contains only either value or program order, the critical events in this cycle must trace an access cycle and hence correspond to synchronous conflicts. Hence from PO and CO(1), they must also follow the happens-before order. This is a contradiction. So, GV is acyclic. ■

Lemma 7 : *If $op_i \rightsquigarrow op_j$ is a synchronous path then $end_(op_i) \mapsto end_(op_j)$.*

Proof: $op_i \rightsquigarrow op_j$ contains orderings that are either (i) program order, or (ii) value order (i.e., $x!v \xrightarrow{v} x?v$). From CO(1), value order also corresponds to happens-before order between the two involved events. Moreover, since $op_i \rightsquigarrow op_j$ is synchronous, the critical events must follow the happens-before order because of PO and CO. Hence the claim. ■

Knowing that conditions PO and CO avoid any view cycle in GV, we need to show that no augmentation derived from Rule 1, $(x!v)_1 \rightsquigarrow (x?v')_2 \Rightarrow (x?v)_4/(x!v)_1 \rightarrow (x!v')_3$, can create a view cycle without creating also a happens-before cycle. Notice that P_2 and P_4 may be the same process and that the processes involved in the augmentation rule are related by $P_1 \xrightarrow{x} P_2/P_4 \xleftarrow{x} P_3$. Hence we start with asymmetric GV, and use the result of Lemma 2 to show that successive augmentations preserve this property. In Phase I, we show that ordering of conflicts on synchronous edges preserve the asymmetry. In Phase II, we proceed by showing that any augmentation applied on synchronous edges does not create any view cycle. In Phase III, we show that adding augmentations on asynchronous edge does not create any view cycle. Finally, in Phase IV, we show that the additional ordering required to obtain a possible view does not create any cycle in the resulting view and hence NI ensures sequential consistency.

Phase I: Ordering conflicts on synchronous edges

In this first phase, we order all unordered synchronous write conflicts and read/write conflicts according to their happens-before order. For example, suppose $x!v$ and $x!v'$ are unordered in GV. In such a case, we augment GV with $x!v \rightarrow x!v'$ if $x!v \mapsto x!v'$. Similarly, order any unordered read/write conflicts. This must exist according to CO(2a) and CO(2b).

Hence, at the end of Phase I, GV contains inter-process "$\rightarrow$" each of which coincides with the happens-before order (from the above and CO(1)). Moreover, the critical events of every synchronous path are happens-before ordered. We call this property (acyclic and happens-before ordered synchronous path) well-ordered. So, according to Lemma 6, GV is well-ordered (before Phase I).

Lemma 8 : *Every augmentation in Phase I preserves the well-ordering in GV.*

Proof: This is immediate as each augmentation in Phase I involves synchronous conflicts which are ordered according to their happens-before order. Cyclicity will immediately contradict the anti-symmetry of the happens-before relation. ∎

Phase II: Augmentation on synchronous edges

In this part, we deal with the simpler case in which we apply successively Rule 1 only in cases where the consequence (right-hand-side) is synchronous, i.e., every access edge lies in some access cycle. This leads to a GV that is well-ordered.

Lemma 9 : *Every augmentation in Phase II preserves the well-ordering in GV.*

Proof: Before an augmentation with Rule 1, by inductive argument, GV is well-ordered (Lemma 8). Now consider the application of an instance of Rule 1 $((x!v)_1 \rightsquigarrow (x?v')_2 \Rightarrow (x?v)_4/(x!v)_1 \rightarrow (x!v)_3)$ when:

1. both (P_1, P_2) and (P_4, P_3) are synchronous process pairs:

   (a) $(x!v)_1 \rightsquigarrow (x?v')_2 \Rightarrow (x?v)_4 \rightarrow (x!v')_3$

   By the assumption in Phase II and definition of synchronous conflicts, (P_1, P_2) and (P_4, P_3) are synchronous processes, i.e., both the guard and the consequence of Rule 1 involve synchronous conflicts. Hence, from the synchrony of (P_1, P_2) and Lemma 8, $end_(x!v)_1 \mapsto end_(x?v')_2$. Now, suppose GV already contains $(x!v')_3 \rightsquigarrow (x?v)_4$, then, from the synchrony of (P_4, P_3) and Lemma 8, $end_(x!v')_3 \mapsto end_(x?v)_4$. This immediately contradicts CO(3).

   (b) $(x!v)_1 \rightsquigarrow (x?v')_2 \Rightarrow (x!v)_1 \rightarrow (x!v')_3$

   This augmentation is needed only if $x?v$ does not occur in GV, otherwise case (a) will render this case redundant. The proof is immediate: as before, we know that for the guard, $end_(x!v)_1 \mapsto end_(x?v')_2$. Now, suppose $(x!v')_3 \rightsquigarrow (x!v)_1$ exists. From Lemma 8 and the inductive assumption, this implies that $end_(x!v')_3 \mapsto end_(x!v)_1$. This contradicts CO(3).

2. (P_1, P_2) is an asynchronous process pair:

   In this case, P_2 and P_4 must be the same process. Otherwise we have two writers and two readers rendering (P_1, P_2) synchronous.

   (a) $(x!v)_1 \rightsquigarrow (x?v')_2 \Rightarrow (x?v)_2 \rightarrow (x!v')_3$

   The asynchrony of (P_1, P_2) implies that $(x!v)_1 \rightarrow (y!u)_1 \rightarrow (y?u)_2 \rightarrow (x?v')_2$. From PO (virtual cycle and single edge), we know that $end_(x!v)_1 \mapsto end_(y!u)_1 \mapsto end_(y?u)_2 \mapsto end_(x?v')_2$. Now, suppose GV already contains $(x!v')_3 \rightsquigarrow (x?v)_2$. From Lemma 8, the inductive assumption and the synchrony of (P_2, P_3), we have $end_(x!v')_3 \mapsto end_(x?v)_2$. The above two happens-before relations immediately violate CO(3).

   (b) $(x!v)_1 \rightsquigarrow (x?v')_2 \Rightarrow (x!v)_1 \rightarrow (x!v')_3$

   This case is the same as case (a) except that we have $end_(x!v')_3 \mapsto$

$end_(x!v)_1$ contradicting also CO(3).

Moreover, in all of the above, the augmentation must coincide with the happens-before order between the augmented conflict pair (guard), since they are already ordered in the latter order after Phase I. Hence GV is acyclic. ■

From the above lemmas, at the end of Phase II, all synchronous conflicts are ordered in the resulting GV. Moreover, all orderings in GV coincide with the happens-before order and GV is well-ordered. The remaining unordered conflicts in GV belong to asynchronous read-write or write-write conflicts.

Phase III: Augmentations on asynchronous edges

In this third phase, we consider application of Rule 1 on asynchronous edges, i.e., the two possible consequences of Rule 1, (i) $(x?v)_2 \rightarrow (x!v')_3$ and (ii) $(x!v)_1 \rightarrow (x!v')_3$, are applied on asynchronous edges. In this phase, we have two possibilities of (P_1, P_2) and (P_2, P_3):

	(P_1, P_2)	(P_2, P_3)
(i)	Synchronous	Asynchronous
(ii)	Asynchronous	Asynchronous

We need to show that in every augmentation via Rule 1, each of these two cases preserves acyclicity in GV.

Lemma 10 : *Augmenting the GV from Phase II using Rule 1 preserves the well-ordering in GV.*

Proof: Consider:

1. $(x!v)_1 \leadsto (x?v')_2 \Rightarrow (x?v)_4 \rightarrow (x!v')_3$:

   First, we claim that $P_2 = P_4$, otherwise the two process pairs must be synchronous. Indeed, two writers and two readers sharing a common object form an access cycle.

   From the guard,$(x!v)_1 \leadsto (x?v')_2$, we deduce the following:

- from case (i), the synchrony of (P_1, P_2) implies that $(x!v)_1 \rightarrow .. \rightarrow (op)_2 \rightarrow (x?v')_2$ where $(x!v)_1 \rightarrow ..(op)_2$ is a synchronous path. This implies, from well-ordering and PO (virtual cycle) respectively, that $end_(x!v)_1 \mapsto end_(op)_2$ and $end_(op)_2 \mapsto end_(x?v')_2$.
- from case (ii), the asynchrony of (P_1, P_2) implies that P_1 and P_2 are connected by a single and direct edge in which case we have from PO (virtual cycle and single edge) that $(x!v)_1 \rightarrow (y!u)_1 \rightarrow (y?u)_2 \rightarrow (x?v)_2 \Rightarrow end_(x!v)_1 \mapsto end_(y!u)_1 \mapsto end_(y?u)_2 \mapsto end_(x?v')_2$.

Hence in both cases, we have $end_(x!v)_1 \mapsto end_(x?v')_2$.

Now, suppose GV already contains $(x!v')_3 \rightsquigarrow (x?v)_2$. By the same reasoning, we conclude $end_(x!v')_3 \mapsto end_(x?v)_2$. The above two derived happens-before relations contradict CO(3).

2. $(x!v)_1 \rightsquigarrow (x?v')_2 \Rightarrow (x!v)_1 \rightarrow (x!v')_3$.

Analogously to the preceding case, we have $end_(x!v)_1 \mapsto end_(x?v')_2$. Suppose $(x!v')_3 \rightsquigarrow (x!v)_1$. Then we get also $end_(x!v')_3 \mapsto end_(x!v)_1$ by similar arguments. These contradict CO(3).

Since Phase III does not introduce any new synchronous path in GV, the preserving of happens-before order in a synchronous path follows from Lemma 9. Hence GV is acyclic. ■

Thus after Phase III, the resulting GV is well-ordered and no further augmentation under Rule 1 is possible. However, GV may still contain unordered conflicts between $(x!v)_1$ and $(x!v')_3$ which involve a single reader, say P_2. We show that we can order these conflicts successively and arbitrarily without introducing cycles until an acyclic possible view is formed.

Phase IV: Ordering of asynchronous concurrent writes

At this time, GV may still contain asynchronous conflicts that are unordered as Rule 1 cannot be applied to them, i.e., $x!v \rightarrow x?v'$ does not exist in GV. In Phase IV, take any two unordered $(x!v)_1$ and $(x!v')_3$ and augment GV with

$(x!v)_1 \rightarrow (x!v')_3$ and $(x?v)_2 \rightarrow (x!v')_3$ where P_2 is an arbitrary process that reads the value v. If this augmentation creates $(y!u) \rightsquigarrow (y!u')$, then order also $(y?u) \rightarrow (y!u')$. We call this arbitrary ordering. This is repeated until unordered conflicts do not exist in GV.

Lemma 11 : *The processes involved in Phase IV must be asynchronous.*

Proof: This immediately follows as a consequence of Phases I and II. ∎

Lemma 12 : *The arbitrary ordering in Phase IV preserves the well-ordering in the resulting GV.*

Proof: Consider each step of the arbitrary ordering. Since $(x!v)_1$ and $(x!v')_3$ are unordered to start with, ordering them does not introduce a cycle in GV. Now consider the introduction of $(x?v)_2 \rightarrow (x!v')_3$ that follows. Since Rule 1 cannot be applied before the ordering, we cannot have $(x!v')_3 \rightsquigarrow (x?v)_2$ before the augmentation. Hence GV remains acyclic. Now we need also to show that Rule 1 cannot be applied after the augmentation step. Suppose we get the following after the augmentation:

(i): $(y!u)_i \rightsquigarrow (x!v)_1/(x?v)_2 \rightarrow (x!v')_3 \rightsquigarrow (y?u')_j$.

This is impossible, otherwise we have traced an access cycle with these events.

(ii): $(y!u)_i \rightsquigarrow (x!v)_1/(x?v)_2 \rightarrow (x!v')_3 \rightsquigarrow (y!u')_j$

This is possible under asynchronous edges when $P_i = P_1$, $P_j = P_3$ and the only reader of y is P_2. Otherwise, an access cycle is traced through object y. A consequence of case (i) is that GV cannot contain $(y!u')_3 \rightsquigarrow (y!u)_2$. Hence, augmenting it with $(y?u)_2 \rightarrow (y!u')_3$ in the step would preserve acyclicity. The preserving of the happens-before order in a synchronous path follows from Lemma 9, since Phase III does not introduce any new synchronous path in GV. This completes the proof.

∎

Theorem 2 *NI implements sequential consistency.*

Proof: From Lemma 12, at the end of Phase IV, the resulting GV is well-ordered. Moreover, it does not contain any unordered conflicts. Hence it is an acyclic possible view. Hence the claim. ■

Based on the above abstract results, we can focus on designing a protocol that synchronizes all virtual cycles while maintaining that every read or write of an object is locally atomic (to the object). To illustrate this aspect, let us consider the simple access graph in Figure 16. In this graph, there are only two access cycles. Suppose process P_1 performs the following sequence of operations: $x!1; y!2; z?v; t?v'$... Then Lemma 6 asserts that $x!1$ does not delay $y!2$ but must be completed before $z?v$ as the two writes are in two different access cycles and cannot form a view cycle. Hence $end_(x!1) \mapsto end_(y!2)$ does not have to be enforced. Similarly, $end_(y!2) \mapsto end_(z?v)$ does not have to be enforced.

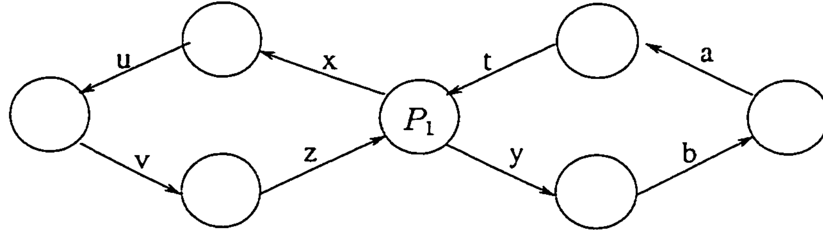


Figure 16: Access graph with 2 independent access cycles

Access Cycle Based Synchronous (Neighbor) Protocol

In this protocol, synchronization on asynchronous edges is done implicitly by assuming the presence of FIFO channels. To synchronize synchronous edges, objects are separated into 2-phase and 3-phase objects respectively. Single reader objects are synchronized using a 2-phase protocol. Others are synchronized using a 3-phase protocol. Synchronization delay between operations in a process is confined to operations that lie in a same virtual cycle. Operations which are not lying in a virtual cycle are not synchronized and do not incur delay between them (other than the FIFOness requirement of the underlying channels). The serialization of concurrent writes in

3-phase objects is achieved by using logical timestamp augmented with the process identifier. It is assumed that a process contains a sequence of memory operations to be invoked. This sequence is the program order of these operations. The invocation of a memory operation spawns a child thread from the parent process thread. The end of an operation is delayed if a preceding operation in a common virtual cycle has not ended. When an operation finishes, the child thread disappears. In addition, there is a kernel thread that is responsible for receiving and updating the values of objects readable by that process. The details of the protocol are:

```
Process i:
   Suppose ti is the logical clock shared by all the threads in process i and txi is the times-
   tamp/version number of the most recent version of x established at process i. We assume that
   each statement in the protocol is atomically executed in a thread and a parent thread spawns
   child threads in program order.

(i) (x!v) thread:
   case of
      3-phase write:
         procedure write_thread_3(x,v,ts)*;
            local:  ts, wait;
            ti, txi, ts:= inc(ti);                                         (st_(x!v))
            broadcast update3(x,v,i,ts) to all readers of x;
            wait := {j| process j is a reader of x};
            repeat until (wait = empty or killed(x,ts)) and
                  (all active older brother threads in a same virtual cycle have returned)
               {upon receipt of ack(x,i,ts) from process j do
                        wait := wait - {j} };
            if ¬killed(x,ts)
               then broadcast commit(x,i,ts) to all readers of x;         (end_(x!v))
         return;
      2-phase write:
         procedure write_thread_2(x,v);
            repeat until all active older brother threads in a same virtual cycle have returned;
            send update2(x,v) to the reader;                               (st_(x!v))
            wait for acknowledge from the reader;
         return;

*For simple presentation, it is assumed here that a writer is also a reader.

(ii) (x?v) thread:
   procedure read_thread(x);
      repeat until all older brother threads in a same virtual cycle have returned;
      case of
         3-phase object:
            repeat until readable(x);
            return value(x);                                  (st/end_(x?v))
         2-phase object:
            return value(x);                                  (st/end_(x?v))
```

```
(iii) Kernel thread:
   repeat forever
      case of
         receipt of update3(x,v,j,ts') from process j:
            ti:  = max(ti,ts');
            if txi < ts' then
               readable[x] := false;  value[x] := v;
               txi := ts';
               send ack(x,j,ts');
               for all active write_thread(x,v,ts) and ts' > ts do
                  killed[x,ts] := true;
         receipt of commit(x,j,ts'):
            if txi = ts' then readable[x] := true;
         receipt of update2(x,v) from process j:
            value[x] := v;
            send acknowledge(x,v) to process j;       (end_(x!v))(2-phase write)
   end repeat forever
```

Lemma 13 : *The implementation of the synchronous protocol satisfies conditions* **PO** *and* **CO**.

Proof: The details of the correctness proof involve showing the causal relationship among the ending of all operations. We proceed by showing that the protocol indeed implements all the happens-before relations required by PO and CO:

- PO - $op_1 \xrightarrow{p} op_2 \Rightarrow end_(op_1) \mapsto end_(op_2)$

  Program order among two operations (events) op_1 and op_2 are ensured by the loop that delays the ending of the actions in the read and the two (2-phase and 3-phase) writes operations. Hence, $end_(op_1) \mapsto end_(op_2)$. This delay is enforced only if the operations lie on a same virtual cycle. Program order is also ensured by the presence of FIFO channels.

- CO - Conflict order

  (1) $x!v \rightarrow x?v$: There are two write operations to consider. In the 3-phase write, the variable "readable" ensures that the new value is not read before the write ends. Hence, $end_(x!v) \mapsto end_(x?v)$. In the 2-phase write, the write ends as soon as the variable

is updated at the reader node. Hence, $end_(x!v) \mapsto end_(x?v)$. This ensures conditions CO(1).

(2) $x!v \rightarrow x!v'$: A complete ordering of all the write operations is implemented by the use of the timestamp `ts`. So, if there are two concurrent writes, one of them will be killed according to the timestamp. This ensures $end_(x!v) \mapsto end_(x!v')$. This ensures conditions CO(2b)

(3) $x!v \rightarrow x?v'$: As in case (2), since the writes are totally ordered, we also have $end_(x!v) \mapsto end_(x?v')$. This ensures conditions CO(2a)

(4) $x?v \rightarrow x!v'$: By construction, in our protocol, the read operation is virtually atomic since $st_(x?v)$ and $end_(x?v)$ are both associated with the same operation. So, if $x!v'$ is a 2-phase write, then $end_(x?v) \mapsto end_(x!v')$ as the local update and the read are both locally atomic operations. If $x!v'$ is a 3-phase write, then $end_(x?v) \mapsto end_(x!v')$ as the update of the variable `readable` and the read are both locally atomic operations. This ensures condition CO(2a).

(5) CO(3) : CO(3) assumes that $end_(x!v)_1 \mapsto end_(x?v')_2$. Then, from the total ordering of write operations ensured by the timestamps, we know that $end_(x!v)_1 \mapsto end_(x!v')_3$ (either $x!v$ really ends before $x!v'$ or it was killed because of its smaller timestamps). Then, the fact that $end_(x!v)_1 \mapsto end_(x?v)_2$, the atomicity of local updates, and the persistence of memory information ensure that $\neg(end_(x!v')_3 \mapsto end_(x?v)_2 / end_(x!v)_1)$ is always satisfied.

■

3.2.3 Flush Protocol

The flush protocol is a novel protocol designed to eliminate the tight synchronization imposed by PO and CO. In particular, we wish to allow conflicting operations to proceed asynchronously. So, in the flush, we enhance concurrency by permitting all operations in different edges in a virtual cycle, except a specific one called the *flush* operation, to execute asynchronously. When the flush operation is to be executed, it synchronizes all processes in the virtual cycle such that all events that have started must have completed before the flush operation can complete. The correctness of the flush protocol is based on the following conditions, particularly on the flush order condition that must be enforced between an arbitrary operation (*op*) and a flush operation (fop) in a same virtual cycle. We use *st_(op)* to represent the starting event of *op*, as marked explicitly in the detailed protocol earlier.

Condition 3 : **RPO** - *Relaxed Program Order*

If op_i *and* op_j *are two instances of operations by a process on a same edge or on a same virtual cycle then we require:* $op_i \xrightarrow{p} op_j \Rightarrow st_(op_i) \mapsto st_(op_j)$.

Hence, program order is preserved only in the form of start-event order for every potential view cycle.

Condition 4 : **RCO** - *Relaxed Conflict Order*

In any case, a write $x!v$ *must have started before any read returning the value* v *can end. Hence, we require:* $st_(x!v) \mapsto end_(x?v)$.

Condition 5 : **WO** - *Write order*

For any $x!v$ *and* $x!v'$*, we require:*

1. $end_(x!v) \mapsto end_(x!v')$ *or* $end_(x!v') \mapsto end_(x!v)$.

2. $end_(x!v) \mapsto end_(x!v') \Rightarrow end_(x?v) \mapsto end_(x!v')$

Condition 6 : **EO** - *Edge Ordering (FIFOness on a single edge)*

Suppose $x!v$, $x!v'$, $x?v$, $x?v'$, $y!u$ *and* $y?u$ *are instances of operations in a same edge from a process. We require the following "FIFOness":*

1. $st_(x!v) \mapsto st_(x!v') \mapsto st_(y!u) \Rightarrow \neg(st_(y?u) \mapsto st_(x?v))$ *and* $\neg(st_(x?v') \mapsto st_(x?v))$

2. $st_(x!v) \mapsto st_(x?v') \Rightarrow \neg(st_(x?v') \mapsto st_(x?v))$

EO is used to avoid view cycles that can occur on a single edge.

Condition 7 : **FO** - *Flush Order in a virtual cycle:*

Each virtual cycle has a flush operation fop such that for every instance of operation op in the virtual cycle, we have

1. $st_(op) \mapsto st_(fop)$ *or* $st_(fop) \mapsto st_(op)$,

2. $st_(op) \mapsto st_(fop) \Rightarrow end_(op) \mapsto end_(fop)$.

3. $st_(fop) \mapsto st_(op) \Rightarrow end_(fop) \mapsto end_(op)$.

Conditions 3 to 7 are called flush invariants and we now show that they are sufficient to ensure sequential consistency (SC).

Let us consider the augmentation of GV with the happens-before order among writes to each object, and their respective reads according to WO. We call this view graph TGV (Temporal Global View).

Lemma 14 : *TGV is acyclic.*

Proof: Suppose otherwise. Take a cycle in TGV consisting of a cyclic sequence of critical events in $P_1; P_2; ...; P_1$ (Recall critical events are the first and last events in a process in the cyclic sequence.). As in earlier proofs (see Lemmas 3 and 4) these critical events trace either (a) a virtual cycle or (b) a single edge of the access graph traversed twice.

(a) virtual cycle case

Since each access cycle must contain a flush operation, one of the critical events must be a flush. Without loss of generality, take the first/last operation (i.e., op_1^1) in the cyclic sequence of critical events $[op_1^1; op_2^1; ..; op_2^k; op_1^1]$ as the flush operation. The ordering between $op_2^i \rightarrow op_1^{(i+1)}$ in this sequence is one of the following:

(i) $x!v \to x?v$: From RCO, $st_(x!v) \mapsto end_(x?v)$.

(ii) $x!v \to x!v'$: This comes from TGV, and hence $end_(x!v) \mapsto end_(x!v')$.

(iii) $x?v \to x!v'$: This comes from TGV, and hence $end_(x?v) \mapsto end_(x!v')$.

Consider $op_1^1 = fop$. From conditions RPO and FO, we have $st_op_1^1 \mapsto st_(op_2^1)$ and $end_(fop) \mapsto end_(op_2^1)$. The next relation in the given sequence of critical events is an instance of (i), (ii) or (iii). Consider $op_2^1 \to op_1^2$ is $x!v \to x!v'$ (the same holds for $x!v \to x?v$ and $x?v \to x!v'$). Hence $st_(fop) \mapsto st_(x!v) \mapsto end_(x!v) \mapsto end_(x!v')$. From FO, we get $end_(fop) \mapsto end_(x!v) \mapsto end_(x!v')$. This in turn leads to $st_(fop) \mapsto st_(x!v')$. From this deduction, for each occurrence of (i), (ii) or (iii), we can extend $st_(fop) \mapsto st_(op_1^i)$ where op_1^i is the second event of (i), (ii) or (iii). Hence $end_(op_1^1) \mapsto end_(op_j^i)$ for every event op_i^j in the given sequence, including op_1^1 itself. This is a contradiction.

(b) single edge case

Since it is a single edge, the cyclic sequence of critical events concerns only two processes and mimics:

- $[(x!v)_1; (x!v')_1; (x?v')_2; (x!v)_1]$

  However, this sequence implies the sequence $(x?v')_2 \xrightarrow{p} (x?v)_2$ in P_2. But, from RPO we must then have $st_(x!v)_1 \mapsto st_(x!v')_1 \mapsto st_(x?v')_2$ and $st_(x?v')_2 \mapsto st_(x?v)_2$, This contradicts EO(3).

- $[(x!v')_1; (y!v'')_1; (y?v'')_2; (x?v)_2; (x!v')_1]$

  However, this sequence of events implies the sequence $(x!v)_1; (x!v')_1; (y!v'')_1$ and $(y?v'')_2 \xrightarrow{p} (x?v)_2$ in P_2. From RPO(1), we must then have $st_(x!v)_1 \mapsto st_(x!v')_1 \mapsto st_(y!v'')_1 \mapsto st_(y?v'')_2$ and $st_(y?v'')_2 \mapsto st_(x?v)_2$. This contradicts EO(2).

- $[(x!v')_1; (x?v')_2; (x?v)_2; (x!v')_1]$

  However, this sequence implies the sequence $(x?v')_3 \rightsquigarrow (x?v)_4$ in some processes P_3 and P_4. But this also creates the cycle $[(x!v')_1; (x?v')_2; (x!v)_i;$

$(x?v')_3]$ which lies in a virtual cycle. Therefore, there is a flush operation in the sequence such that $[end_(x!v')_1 \mapsto \ \ldots \ \mapsto end_fop \mapsto end_(x?v')_3 \mapsto end_(x!v')_1]$. Hence a contradiction.

■

Lemma 15 : *TGV satisfies R1.*

Proof: Suppose otherwise. There is an instance of $x!v \leadsto x?v'/x!v'$ but there is some $x?v$ such that $x?v \rightarrow x!v'$ does not hold.

1. $x!v \leadsto x!v'$ and there exists $x?v \not\rightarrow x!v'$:

   From WO, we have $end_(x!v') \mapsto end_(x?v)$. This in turn implies $end_(x!v') \mapsto end_(x!v)$. From TGV, we must have $x!v' \rightarrow x!v$. This implies TGV is cyclic and hence is a contradiction.

2. $x!v \leadsto x?v'$ and there exists $x?v \not\rightarrow x!v'$:

   From WO, we must have $end_(x!v') \mapsto end_(x?v)$ and hence $end_(x!v') \mapsto end_(x!v)$. This in turn implies $end_(x?v') \mapsto end_(x!v)$. From TGV, we have $x?v' \rightarrow x!v$. Again, we have a cyclic TGV and a contradiction.

■

Lemma 16 : *TGV is a possible view.*

Proof: It is immediate from Lemma 15 and the fact that TGV does not contain any concurrent writes to a same object (they are all serialized via WO). ■

Theorem 3 : *The flush invariants ensure SC.*

Proof: This follows from Lemmas 14 to 16. ■

So, in terms of the synchronous protocol described in Section 3.2.2, it means a child thread no longer waits for its old brother threads in a same virtual cycle to return before proceeding with its termination operation. The parent thread can start each child thread of a virtual cycle in program order. These child threads move asynchronously with respect to their brothers as well as with respect to other threads in

other processes. The only synchronization imposed is atomically triggered whenever a special write operation in each virtual cycle is invoked by a parent thread. This special write operation is called a *flush_write*. The *flush_write* serves to synchronize all asynchronous operations in a virtual cycle that can potentially lead to a view cycle. Intuitively a view cycle involving the operations in a given virtual cycle cannot be formed unless every operation in the virtual cycle has been invoked since the last (flush) synchronization. Hence synchronization is completely hidden/ignored until the *flush_write* occurs. When it does, each writer process in the same virtual cycle is checked so that all operations in the virtual cycle that have started before the flush must have ended before the flush is allowed to end. The theory allows the use of a read operation as a flush, but for our design and simulation, we use a write for this purpose. The details of such a protocol are given below. It is assumed that the protocol described in Section 3.2.2 is used with the removal of all delays (waiting) caused by the return of older brother threads (those lines marked in italics). This removal concerns only events in a same virtual cycle. The changes to the base protocol then include the case of a *flush_write* in the write thread and the case of receipt of *flush_write* in the kernel thread:

```
(x!v) thread:
   case of
      :::
      flush_(x!v):
         procedure flush_(x!v);
            broadcast flush_(x!v) to each process in the virtual cycle;
            repeat until receipt of flush_ack(x,v) from each process;      (st_flush)
            broadcast commit_flush(x,v) to each process;                   (end_flush)
         return;


Kernel thread:
   case of
      :::
      receipt of flush_(x!v) from process j;
         atomically perform
            wait until
               all child threads in the same virtual cycle as the flush operation have returned;
            send flush_ack(x,v) to process j;
            if process i is a reader of x then value[x]:= v;
```

In addition, the parent thread is changed so that it delays spawning a child thread if the latter is in the same virtual cycle of an ongoing *flush_write* which has not yet committed, i.e., the *commit_flush* message has not been received.

Lemma 17 : *The preceding implementation of the flush protocol satisfies the flush invariants.*

Proof: The details of its correctness proof involve showing the causal relationship among the ending of all operations. In particularly, we have to show that if an operation in the virtual cycle starts before the flush operation, then it ends before the flush. This condition is locally enforced in the protocol by requiring that each earlier operation ends before acknowledging the flush operation. Moreover, since these actions are done atomically and the start of other threads is delayed by the parent, no other operations can start before the flush ends.

As for the neighbor protocol, WO is enforced by timestamps and EO is enforced by FIFO channels. ■

3.2.4 Multiple Flush Protocols

The flush invariants presented earlier can be considered as "single flush" invariants since every cycle is synchronized by a single flush operation. For scalability considerations, it is possible to use multiple flush operations in a long cycle. Protocols using multiple flush operations inside a single virtual cycle may be useful when we have a long virtual cycle since it reduces the number of participants for each broadcast and hence reduces the time for the 3-phase flush protocol.

When multiple flush operations are used, each flush operation synchronizes a subset of the operations involved in the virtual cycles. According to the organization of the operations, these subsets must possess some specific characteristics.

Definition 22 : *A* **group** *is a set of operations* $\{op_1, op_2, ..., op_k\}$ *such that for all* i, j, op_i *and* op_j *are operations of the same virtual cycle.*

Definition 23 : *Two operations* op_i *and* op_j *are* **consecutive** *in a virtual cycle if they are emanating from the same process on two separate edges of the cycle or from two distinct processes on the same edge of the cycle.*

Definition 24 : *A* **segment** *is a group such that, for each* i*,* op_i *and* op_{i+1} *are consecutive operations in the virtual cycle.*

As an example, consider the virtual cycle $P_1 \overset{o_1}{\rightarrow} P_2 \overset{o_2}{\rightarrow} P_3 \overset{o_3}{\rightarrow} P_4 \overset{o_4}{\rightarrow} P_1$. This virtual cycle can be divided into the two following segments, each containing four operations: $\{(o_1!v_1)_1, (o_1?v_1)_2, (o_2!v_2)_2, (o_2?v_2)_3\}$ and $\{(o_3!v_3)_3, (o_3?v_3)_4, (o_4!v_4)_4, (o_4?v_4)_1\}$. The number k of operations in each group may differ. As an example, the preceding virtual cycle can also be divided into two groups or segments containing respectively three and five operations.

When a virtual cycle is divided into groups or segments, we can provide a distinct flush operation for each group or segment. The choice of the flush operation for each group or segment produces many possible organizations and hence multiple flush conditions. In this section, we present three different multiple flush conditions. These flush conditions, combined with conditions EO, WO and RPO described earlier, produce different multiple flush invariants.

Linear Flush Operations

Consider a virtual cycle which is divided into n segments $s_1 = \{op_1^1, op_2^1, ..., op_k^1\}$, ..., $s_n = \{op_1^n, op_2^n, ..., op_{k'}^n\}$ where op_i^j identifies the ith operation of segment s_j. The segments are consecutive, which means that the last operation of segment s_j, op_k^j, and the first operation of segment s_{j+1}, op_1^{j+1}, are consecutive events.

When we use a single flush, the choice of the operation in a virtual cycle is simple. However, with multiple flush, the choice becomes more complex. Indeed, a virtual cycle can produce two view cycles, one in each "direction", and so each "direction" must be covered. This is not a problem with the single flush since it covers all operations in both directions. With linear flush protocols, each segment must be covered by two flush operations, one at each end. A simple strategy is to use the

first and the last event of a segment as flush. Hence, the first flush operation fop_1^i of segment i is op_1^i and the second flush operation fop_2^i is the last event of the segment. The flush operations, fop_1^i and fop_2^i, must themselves be synchronous respectively with the first operation of the next segment or the last operation of the preceding segment. The condition for the linear flush that replaces FO in the flush invariants is:

Condition 8 : **LFO** - *Linear Flush Order in a virtual cycle:*

1. op^i *and* fop_l^i *(for* $l = 1$ *or* 2*) in a same segment* s_i *of a virtual cycle* $\Rightarrow st_(op^i) \mapsto st_(fop_l^i)$ *or* $st_(fop_l^i) \mapsto st_(op^i)$,

2. $st_(op^i) \mapsto st_(fop_l^i)$ *in a same segment* $\Rightarrow end_(op^i) \mapsto end_(fop_l^i)$,

3. $st_(fop_l^i) \mapsto st_(op^i)$ *in a same segment* $\Rightarrow end_(fop_l^i) \mapsto end_(op^i)$, *and*

4. $st_(fop_1^i) \mapsto st_(op_k^{i-1}) \Rightarrow end_(fop^i) \mapsto end_(op_1^{i+1})$ *where* op_k^{i-1} *is the last event of the segment* s_{i-1} *or* $st_(fop_2^i) \mapsto st_(op_1^{i+1}) \Rightarrow end_(fop^i) \mapsto end_(op_1^{i+1})$.

Conditions 1 to 3 ensure that the flush operations synchronize every operation in their segment. Condition 4 synchronizes the flush operations with the adjacent segments.

When we combine conditions RPO, RCO, EO, WO and LFO, we obtain the linear flush invariants. Showing that these invariants are still sufficient to ensure sequential consistency is similar to the proof showing that the single flush invariants guarantee sequential consistency. So, we assume that we still have the view graph TGV. We can show that TGV remains acyclic under the linear flush invariants.

Lemma 18 : *TGV is acyclic under the linear flush invariants.*

Proof: Suppose otherwise. Take a cycle in TGV consisting of a cyclic sequence of critical events in $P_1; P_2; ..P_1$. As in earlier proofs these critical events trace either (a) a virtual cycle and (b) a single edge of the access graph traversed twice.

(a) virtual cycle case

Since each virtual cycle is divided into segments and each segment contains a flush operation, the last critical event of the segment is the flush operation. This means that the view cycle that is formed by a set of operations regrouped into m segments of the form:

$$s_1 \rightarrow s_2 \rightarrow \dots s_m \rightarrow s_1$$

where each segment s_i contains a sequence of operations (events) of the form:

$$op_1^i \rightarrow op_2^i \rightarrow op_3^i \dots \rightarrow op_n^i$$

where op_j^i is the jth critical event of segment s_i and op_n^i is the flush operation (fop_1^i) of that segment.

The ordering between two events $op_j^i \rightarrow op_{(j+1)}^i$ in this sequence is either enforced by program order or one of the following:

(i) $x!v \rightarrow x?v$: From RCO, $st_(x!v) \mapsto end_(x?v)$.

(ii) $x!v \rightarrow x!v'$: This comes from TGV, and hence $end_(x!v) \mapsto end_(x!v')$.

(iii) $x?v \rightarrow x!v'$: This comes from TGV, and hence $end_(x?v) \mapsto end_(x!v')$.

If $op_{(n-1)}^i \xrightarrow{p} op_n^i$ than from RPO and LFO, we have $st_(op_k^i) \rightarrow st_(op_n^i) = st_(fop_2^i)$ and hence $end_(op_k^i) \mapsto end_(fop_2^i)$. The next relation in the given sequence of critical events is an instance of (i), (ii) or (iii). Consider the case, $op_j^i \rightarrow op_{(j+1)}^i$ is $x!v \rightarrow x!v'$ (the same holds for $x!v \rightarrow x?v$ and $x?v \rightarrow x!v'$). From WO we have $end_(x!v) \mapsto end_(x!v') \mapsto end_(fop_2^i)$. This in turn leads to $st_(x!v) \mapsto st_(fop_2^i)$. From this deduction, for each occurrence of (i), (ii) or (iii), we can extend $end_(op_2^i) \mapsto end_(fop_2^i)$ for each event in segment s_i. From LFO, we also deduce that $end_(fop_2^{(i-1)}) \mapsto end_(op_2^i) \mapsto end_(fop_2^i)$. Hence, $end_(fop_2^{(i-1)}) \mapsto end_(fop_2^i)$ for all segments s_i involved in the view cycle. The same reasonning can be used to show that the other flush operation fop_1^i ensures the same ordering in the other direction. Hence, this is a contradiction.

(b) single edge case

This case is the same as in Lemma 14 and is not repeated here.

■

Theorem 4 : *The linear flush invariants ensure SC.*

Proof: This follows from Lemmas 16 and 18. ■

When we use a single flush, the choice of the operation in a virtual cycle is simple. However, with multiple flush, the choice becomes more complex. Indeed, since a virtual cycle can produce two view cycles, one in each "direction", each segment must be covered by two flush operations, one at each end. A simple strategy is to use the first and the last event of a segment as flush. However, the best solution is to use a single flush operation for two contiguous segments. Each segment then overlaps with its neighbor segments. The overlapping concerns only one operation which is used to flush the two segments. With that solution, we have fewer flush operations and the segments are still covered both ways.

Consecutive Flush Operations

Consider a virtual cycle which is divided into n groups $g_1 = \{op_1^1, op_2^1, ..., op_k^1\}$, ..., $g_n = \{op_1^n, op_2^n, ..., op_{k'}^n\}$ where op_i^j identifies the ith operation of group g_j, and one segment fs. All the operations of the unique segment, called the flush segment, are used as flush operations. Each operation of the flush segment is used to "flush" all the operations of one or many of the groups. The operations of the flush segment are noted $\{fop^1, fop^2, ..., fop^m\}$. The operations of the flush segment must all be synchronized as specified by the neighbor protocol. Hence they form a synchronized segment. Again, as in the linear flush order, care must be taken to cover the virtual cycle in both directions. In this case, this affects only the flush segment. The conditions for the consecutive flush order are:

Condition 9 : **CFO** - *Consecutive Flush Order in a virtual cycle:*

1. op_i^j *of a group* g_j *flushed by* fop^j *in a particular virtual cycle* $\Rightarrow st_(op_i^j) \mapsto st_(fop^j)$ *or* $st_(fop^j) \mapsto st_(op_i^j)$,

2. $st_(op_i^j) \mapsto st_(fop^j) \Rightarrow end_(op_i^j) \mapsto end_(fop^j)$,

3. $st_(fop^j) \mapsto st_(op_i^j) \Rightarrow end_(fop^j) \mapsto end_(op_i^j)$,

4. *for two consecutive flush operations,* fop^j *and* fop^{j+1}*, we have* $st_(fop^j) \mapsto st_(fop^{j+1})$ *or* $st_(fop^{j+1}) \mapsto st_(fop^j)$,

5. $st_(fop^j) \rightarrow st_(fop^{j+1}) \Rightarrow end_(fop^j) \mapsto end_(fop^{j+1})$ *or* $st_(fop^{j+1}) \rightarrow st_(fop^j) \Rightarrow end_(fop^{j+1}) \mapsto end_(fop^j)$, *and*

6. $st_(fop^m) \mapsto st_(op_1^1) \Rightarrow end_(fop^m) \mapsto end_(op_1^1)$ *where* op_1^1 *is consecutive to* fop^m *in the view cycle and* $st_(fop^1) \mapsto st_(op_i^j) \Rightarrow end_(fop^1) \mapsto end_(op_i^j)$ *where* op_i^j *is consecutive to* fop^m *in the view cycle.*

Conditions 1 to 3 are normal flush requirements. Conditions 4 and 5 ensure that all the operations in the flush segment are synchronous. Finally, condition 6 synchronizes the flush operations at both ends of the flush segments with operations outside the segment. When we combine CFO with conditions RPO, RCO, EO and WO, we obtain the consecutive flush invariants. As before, to show that these invariants are sufficient to ensure sequential consistency, we must show that TGV remains acyclic under these invariants.

Lemma 19 : *TGV is acyclic under the consecutive flush invariants.*

Proof: Suppose otherwise. Take a cycle in TGV consisting of a cyclic sequence of critical events in $P_1; P_2; ..P_1$. As in earlier proofs these critical events trace either (a) a virtual cycle and (b) a single edge of the access graph traversed twice.

(a) virtual cycle case

Each virtual cycle is divided into n groups and one flush segment fs that contains all flush operations. This means that a view cycle, formed by a set of operations regrouped into n groups and one flush segment fs, is of the form:

$$op_1 \rightarrow op_2 \rightarrow op_3 \rightarrow \ldots \rightarrow fs \rightarrow op_1$$

where each segment op_i is a member of a single group g_j and flushed by a single flush operation fop^j of segment fs.

From RPO and WO, we know that for all events, op_i and op_{i+1} (flush events or not) either $st_(op_i) \mapsto st_(op_{(i+1)})$ or $end_(op_i) \mapsto end_(op_{(i+1)})$.

Hence from CFO, we have for every flush operation, fop^i and $fop^{(i+1)}$ in fs, $end_(fop^i) \mapsto end_(fop^{(i+1)})$. Also from CFO, we have that from each event op_i which is flushed by some fop^k, $st/end_(op_i) \mapsto st/end_(op_{(i+1)}) \ldots$ $st/end_(op_l) \mapsto st/end_(fop^1) \mapsto \ldots \mapsto end_(fop^k)$ where $st/end \mapsto st/end$ means $st \mapsto st$ or $end \mapsto end$. So, by transitivity, we have for all non flush events op_i, $end_(op_i) \mapsto end_(fop^k) \mapsto end_(fop^m)$. In particular, $end_(op_1) \mapsto end_(fop^l) \mapsto end_(fop^m)$. However, from CFO, we also have $end_(fop^m) \mapsto end_(op_1)$. The same argument can be used to show the same ordering in the other direction. Hence there is a contradiction.

(b) single edge case

The single edge case is the same as in Lemma 14 and is not repeated here.

■

Theorem 5 : *The consecutive flush invariants ensure SC.*

Proof: This follows from Lemmas 16 and 19. ■

The advantage of this protocol is that the operations in a group can be non-consecutive. However, the operations of the flush segment must be consecutive in the virtual cycle. The problem of this approach is the formation of the groups. Since it is not required to put consecutive operations in groups, other criteria must be established to create each group.

Hierarchical Flush Operations

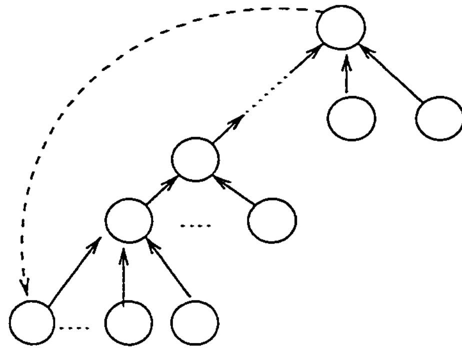
Consider a virtual cycle which is divided into n groups $g_1 = \{op_1^1, op_2^1, ..., op_k^1\}$, ..., $g_n = \{op_1^n, op_2^n, ..., op_{k'}^n\}$ where op_i^j identifies the ith operation of the group g_j.

In the hierarchical flush, all flush operations are organized in a tree-like hierarchy. The root node contains one operation fop^r that flushes all its children nodes. Each intermediary node contains an operation that flushes all its children nodes. The leaf nodes contain only normal operations. All nodes having the same parent represent a group. All the operations of a specific group g_j must appear before their flush operation (fop^j) in the view graph, i.e., for all op_i^j, $op_i^j \leadsto fop^j$ in the view. Again, the root operation must be synchronized with its consecutive operation.

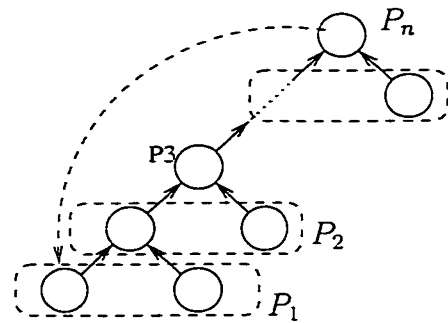
The detailed description and proof of the hierarchical flush invariants are omitted here since they are very similar to the descriptions and proofs of the other multiple flush invariants. However, this particular organization of the flush operations is the most flexible. Indeed, the definition of the flush permits many hierarchical organizations. Figure 17 shows some possible organizations for the hierarchical flush operations. In fact, any rooted tree provides a possible hierarchical organization for the flush operations. Figure 17(a) and (b) show two possible trees. Figure 17(b) shows a hierarchy where each flush operation flushes two operations of the same process. This avoids the use of broadcasting primitives since each flush communicates only with one process. It is interesting to note that CFO can be used to recreate many protocols. Figure 17(c) shows a hierarchy that represents exactly the organization of the neighbor protocol where each operation flushes the preceding one in the view. Figure 17(d) shows a hierarchy that represents the single flush protocol.

3.3 Conclusion

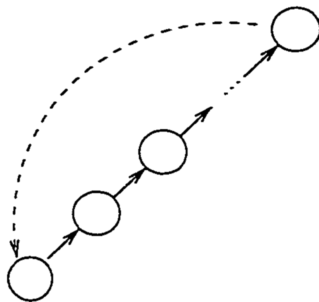
In this chapter, we have presented new protocols that implement sequential consistency. Two of these protocols, the neighbor and flush protocols, were evaluated and compared with other more conventional algorithms. The results of this evaluation are reported in Chapter 8 and in [27]. The multiple flush protocols were not evaluated.



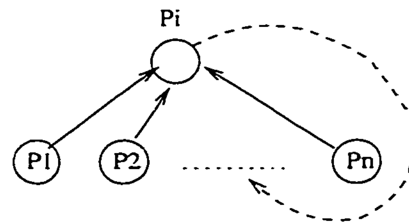
a) General organisation



b) Organisation for process oriented protocol



c) Organisation for the neighbor protocol



d) Organisation for single flush protocol

Figure 17: Possible organizations for hierarchical flush events

More work must be done to complete the implementation and evaluation of all these protocols. Further work must also be done to compare the flush protocols with protocols used in *relaxed consistency models* such as release consistency. We think that these protocols and the flush protocols are intrinsically related.

Chapter 4

DSM Consistency Models Based on Global Views

The view concept is a very powerful and flexible tool to represent different DSM consistency models. In the preceding chapter, we have used it to represent principally sequential consistency. In this chapter, using the global view as a seed, we define hierarchies of weak consistency models. A first minimal consistency model is defined by requiring the global view to be acyclic. Other stronger consistency models are then defined by the use of augmentation rules which enforce additional orderings on operations. These orderings lead to two distinct hierarchies of consistency models. Moreover, another hierarchy is obtained by requiring the acyclic global view to satisfy some additional properties. This last hierarchy, related to causal memory, requires the absence of some bad orderings instead of enforcing new ones.

4.1 Minimal Consistency Based on Global View

From the definition of global view, it follows that a global view can be regarded as a directed graph. When the graph is acyclic, then the global view is a partial order. A minimal consistency model can than be defined.

Definition 25 : *An execution satisfies* minimal consistency *iff its global view is acyclic.*

Process P_i			
$x!v$	:	x.v := v; broadcast (x,v);	$end_(x!v)$
$x?v$	:	returns x.v;	$end_(x?v)$
$receive(x,v)$	:	x.v := v;	update x.v

Figure 18: Asynchronous Update Protocol (AUP)

Intuitively, minimal consistency is achieved when all program orders and value orders do not contradict one another "globally" among the processes. Unfortunately, minimal consistency is a very weak memory model to be used in programming, although it is easily implementable. One such protocol is the asynchronous update protocol (AUP) shown in Figure 18.

Lemma 20 : *The asynchronous update protocol AUP (Figure 18) implements minimal consistency.*

Proof: We use Lamport's happens-before relation (denoted by $\mapsto$) in the proof. We observe that under AUP, $op_1 \rightarrow op_2$ in GV also implies that $end_(op_1) \mapsto end_(op_2)$ (in Figure 18). Specifically, op_1 must have ended before op_2 can end. Events under the happens-before relation must be partially ordered. Hence GV must be acyclic. ■

The asynchronous protocol, since it does not try to order the operations, is fast because a process is not stalled at all by either read or write operations. Hence message communication or network delays do not contribute to memory latency.

4.2 Augmentation Rules and Consistency Models

As minimal consistency is too weak to be of general use, we will focus on its gradual strengthening to form various consistency hierarchies. From here onward, it is assumed that all executions are minimally consistent, i.e., their global views are partially

ordered (acyclic). This assumption is largely based on the ease with which minimal consistency can be guaranteed, such as by the AUP as established in Lemma 20.

4.2.1 Reachability Relations

To formulate our strengthening strategy, let us restrict our attention to the read and write operations of two distinct values of a variable, say x. We use $X?v$ to denote the entire set of read operations that return the value v, i.e., $\{x?v\}$, and $x?v$ to denote a particular element in $X?v$. There are exactly nine possible reachability relations in GV involving an element in $\{x!v, x?v, X?v\}$ and another element in $\{x!v', x?v', X?v'\}$ as follows.

$$\begin{array}{lll} R_9 : X?v \rightarrow x!v' & R_8 : X?v \rightarrow X?v' & R_7 : X?v \rightarrow x?v' \\ R_6 : x?v \rightarrow x!v' & R_5 : x?v \rightarrow X?v' & R_4 : x?v \rightarrow x?v' \\ R_3 : x!v \rightarrow x!v' & R_2 : x!v \rightarrow X?v' & R_1 : x!v \rightarrow x?v' \end{array}$$

In the above notation, for example in R_9, $X?v \rightarrow x!v'$ means that in GV every read of v in x can reach the write of v' in x, i.e., there is a path from the former to the latter in the directed graph GV. We use $R_i > R_j$ to mean that if R_i holds in GV, then R_j also holds in GV. The following lemma holds.

Lemma 21 : *The following order exists among the nine reachability relations:*

$$\begin{array}{lll} R_9 > R_8 > R_7; & R_6 > R_5 > R_4; & R_3 > R_2 > R_1; \\ R_9 > R_6 > R_3; & R_8 > R_5 > R_2; & R_7 > R_4 > R_1. \end{array}$$

Proof: We will show $R_9 > R_8$. The rest is proved similarly.

With $X?v \rightarrow x!v'$ (R_9) and $x!v' \rightarrow X?v'$ (the definition of global view), we can immediately conclude $X?v \rightarrow X?v'$ (R_8). ■

From Lemma 21, a natural hierarchy of reachability relations among the nine relations has surfaced as shown in Figure 19. In this hierarchy an arrow between a relation R_i and R_j means that $R_i > R_j$. This hierarchy can be used later as the basis of defining rules to augment GV, from which stronger consistency models are defined.

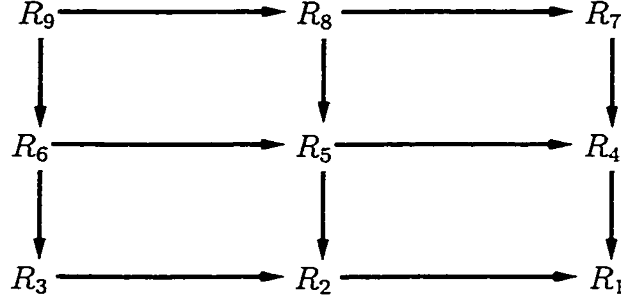


Figure 19: Hierarchy of relations

Some of these reachability relations can be limited to program-order, in which case they are noted R_i^p where the superscript p means *program-order*. Obviously the hierarchy defined earlier between the nine relations can be extended to reachability relations (R_i^p) that are program-ordered.

4.2.2 Augmentation Rules

In this section, we introduce various augmentation rules which, once applied on the basic global view, are used to define weak consistency models. Augmentation rules, as presented in Chapter 2, are used to infer new precedence relations from existing ones. They are guarded commands of the form $R_i \Rightarrow R_j$ where R_i and R_j are reachability relations. An augmented view obtained from the rule $R_i \Rightarrow R_j$ is noted GV_{ij}. A consistency model C_{ij} is definable by requiring the global view GV_{ij} to be acyclic.

If program-ordered reachably relations are used as guards ($R_i^p \Rightarrow R_j$), we have a local augmentation rule. The augmented global view and the consistency models obtained by the use of a local augmentation rule are respectively noted GV_{ij}^p and C_{ij}^p. These local rules are used to define even weaker consistency models than the more *global* augmentation rules.

The well-ordered property of the relations R_1 through R_9 presented in the preceding section were used in [42] to define ordering properties among the consistency models. So, an execution that satisfies C_{ij} will also satisfy C_{ik} if R_j and R_k are two

relations such that $R_j > R_k$. Hence, a memory protocol that implements C_{ij} also implements C_{ik}. Similarly to the hierarchy established among the relations, a hierarchy can be established among all the consistency models obtainable from augmentation rules that use the same guard. So, a distinct hierarchy of consistency models can be obtained by using different guards.

In the following sections, we use $op_1 \xrightarrow{p} op_2$ to represent two events op_1 and op_2 which are program ordered, i.e., op_1 can reach op_2 by traversing only events from the same process. In general, $op_1 \rightsquigarrow op_2$ represents the reachability relation between op_1 and op_2 in GV (a directed path exists from op_1 to op_2 in GV).

4.2.3 Consistency Models Based on Local Augmentation Rules

R_1, R_3, R_4 and R_6 are the only relations that can involve operations in the same process. The remaining five reachability relations may involve reads from multiple processes. We identify these relations as R_1^p, R_3^p, R_4^p and R_6^p respectively. The hierarchy among the relations is still valid among their local counterparts.

R_i^p can be used as the basis of augmenting a global view. In particular, whenever R_i^p holds in a global view GV, we can augment GV so that it satisfies R_j. For example, whenever $x!v \rightarrow x?v'$ holds in program order (written as $R_1^p : x!v \xrightarrow{p} x?v'$), then we augment GV with $x!v \rightarrow x!v'$. In other words, a directed edge is added to GV connecting $x!v$ and $x!v'$. Obviously the latter write operations may belong to different processes. This augmentation rule is stated in the form of a guarded command: $R_1^p \Rightarrow R_3$, or in general, $R_i^p \Rightarrow R_j$. The left hand part of the rule is called the guard and the right hand side is called the consequence of the rule.

Definition 26 : *GV_{ij}^p (i =1,3,4,6) is the augmented global view obtained by applying the local augmentation rule $R_i^p \Rightarrow R_j$. Whenever R_i^p holds in GV, GV is augmented to become GV_{ij}^p such that R_j also holds in GV_{ij}^p. Specifically, if $R_j = op_1 \rightarrow op_2$ and some $op_1 \rightarrow op_2$ does not hold, then O will be augmented with that instance of (op_1, op_2). Since it would be vacuous to augment GV using $R_i^p \Rightarrow R_j$ if $R_i > R_j$, we*

will consider only $R_i < R_j$ in an augmentation rule.

From this definition, we derive the following augmented view:

- Augmented views based on the local guard R_1^p.

  From the guard R_1^p, i.e., $x!v \xrightarrow{p} x?v'$, and from Lemma 21, we can deduce five different augmented global views: GV_{12}^p, GV_{13}^p, GV_{17}^p, GV_{18}^p, GV_{19}^p. The cases of R_4, R_5 and R_6 (hence GV_{14}^p, GV_{15}^p, GV_{16}^p) cannot arise because $x?v$ is not identifiable from the guard R_1^p in these cases. Figures 20 shows GV_{13}^p ($x!v \xrightarrow{p} x?v' \Rightarrow x!v \rightarrow x!v'$) and GV_{19}^p ($x!v \xrightarrow{p} x?v' \Rightarrow X?v \rightarrow x!v'$) of the example execution respectively. In the drawings, we use $op_1 \dashrightarrow op_2$ to denote a new relation introduced by an augmentation rule.

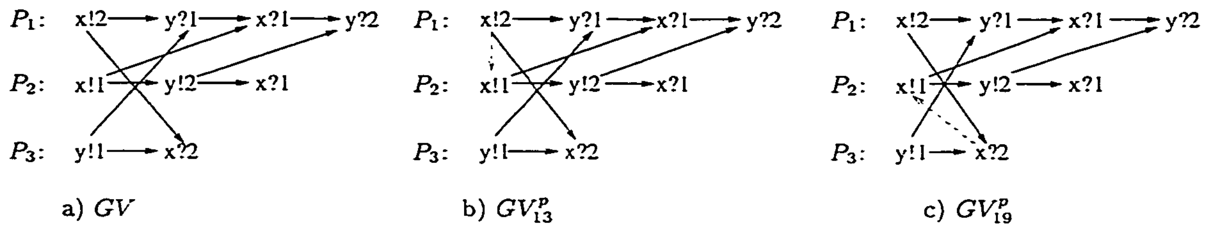


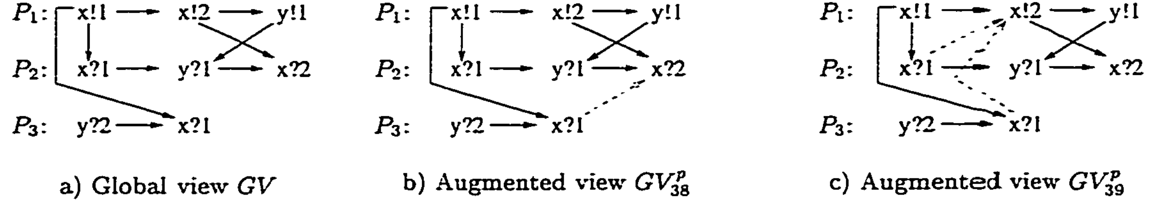
Figure 20: GV_{13}^p and GV_{19}^p

- Augmented views based on the local guard R_3^p.

  From the guard R_3^p, i.e., $x!v \xrightarrow{p} x!v'$, and from Lemma 21, we can deduce only two different augmented global views: GV_{39}^p and GV_{38}^p. The augmentations using R_4, R_5, R_6 and R_7 are unusable since they require knowledge about a specific read, and all the others are equivalent to GV according to Lemma 21. Figure 21 shows some examples of augmented views.

- Augmented views based on the local guard R_4^p.

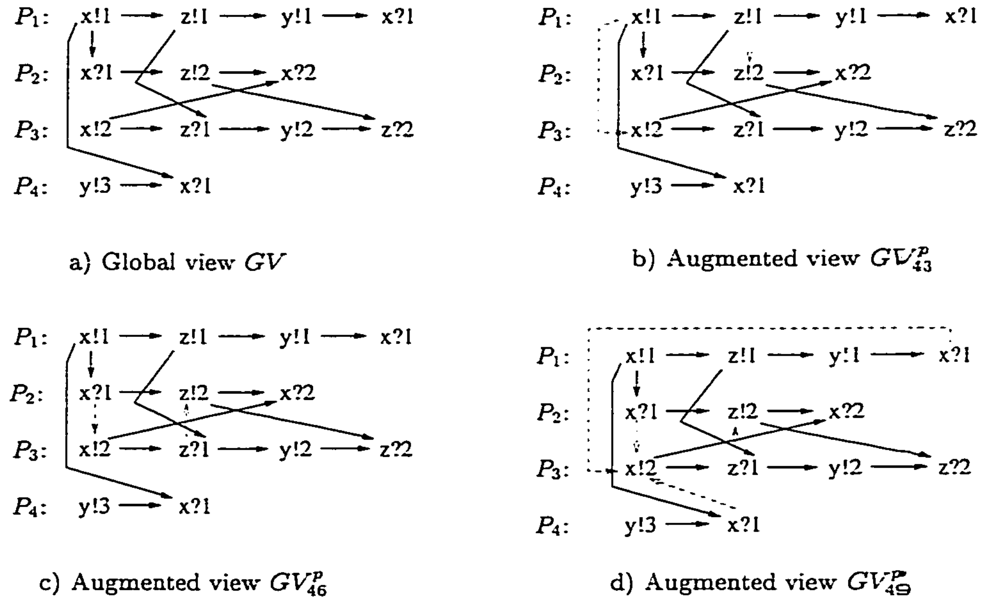
  Again, from the guard R_4^p, i.e., $x?v \xrightarrow{p} x?v'$, and from Lemma 21, we can deduce seven different augmented global views: GV_{49}^p, GV_{48}^p, GV_{47}^p, GV_{46}^p, GV_{45}^p, GV_{43}^p



a) Global view GV b) Augmented view GV_{38}^p c) Augmented view GV_{39}^p

Figure 21: Augmented views based on the guard R_3^p

and GV_{42}^p. The cases of GV_{44}^p and GV_{41}^p are redundant as they are equivalent to GV, according to definition and Lemma 21 respectively. Figure 22 shows some examples of augmented views.



a) Global view GV b) Augmented view GV_{43}^p

c) Augmented view GV_{46}^p d) Augmented view GV_{49}^p

Figure 22: Augmented views based on the guard R_4^p

- Augmented views based on the local guard R_6^p.

From the guard R_6^p, i.e., $x?v \xrightarrow{p} x!v'$, and from Lemma 21, we can deduce two different augmented global views: GV_{69}^p and GV_{68}^p. The augmentation using R_7 is unusable since it requires the knowledge about a specific read, and all the others are equivalent to GV according to Lemma 21. Figure 23 shows some examples of augmented views.

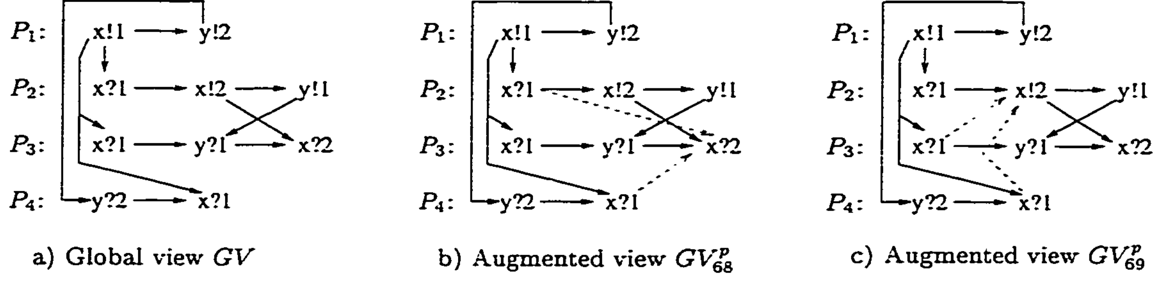


Figure 23: Augmented views based on the guard R_6^p

Given an acyclic GV, it is not guaranteed that an augmentation rule preserves acyclicity in GV_{ij}^p. For example, Figure 24 shows a cyclic GV_{13}^p with an acyclic GV.

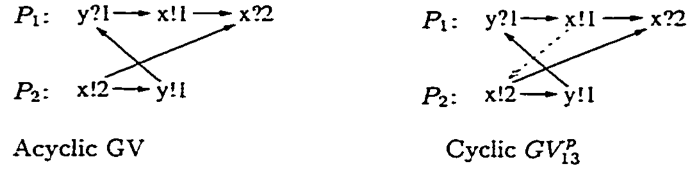


Figure 24: Acyclic global view with a cyclic augmentation

A consistency model can be defined corresponding to each GV_{ij}^p that is acyclic. In other words, C_{ij}^p is the memory consistency model that requires every GV_{ij}^p to be acyclic (and hence consistent under the given augmentation). For all the consistency models derivable from a particular guard R_i^p, a simple consistency hierarchy results, based on the following theorem.

Theorem 6 : *Consider two consistency models C_{ij}^p and C_{ik}^p such that $R_j > R_k$. An execution that satisfies C_{ij}^p must also satisfy C_{ik}^p.*

Proof: From Lemma 21 and the definition of GV_{ij}^p, GV_{ij}^p must contain GV_{ik}^p. Hence acyclic GV_{ij}^p also implies acyclic GV_{ik}^p. ∎

Hence a shared memory protocol that ensures C_{ij}^p will also ensure C_{ik}^p. We denote this ordering relationship $C_{ij}^p > C_{ik}^p$. For the five consistency models based on R_1^p, the following hierarchy is derivable from Theorem 6.

$C_{19}^p > C_{18}^p > C_{17}^p$; $C_{13}^p > C_{12}^p$; $C_{19}^p > C_{13}^p$; $C_{18}^p > C_{12}^p$.

Table 1 provides all the consistency models derivable from R_1^p, R_3^p, R_4^p and R_6^p as the basis of augmentation and summarizes the consistency hierarchies obtainable from these guards. Since R_i^p and R_j^p are not related in general, the four consistency hierarchies based on program-ordered augmentation are unrelated in general.

Guard (R_i^p)	Consistency model	Hierarchy	
$R_1^p : x!v \xrightarrow{p} x?v'$	$C_{19}^p, C_{18}^p, C_{17}^p, C_{13}^p,$ C_{12}^p.	$C_{19}^p > C_{18}^p > C_{17}^p$; $C_{19}^p > C_{13}^p$;	$C_{13}^p > C_{12}^p$; $C_{18}^p > C_{12}^p$.
$R_3^p : x!v \xrightarrow{p} x!v'$	C_{39}^p, C_{38}^p.	$C_{39}^p > C_{38}^p$.	
$R_4^p : x?v \xrightarrow{p} x?v'$	$C_{49}^p, C_{48}^p, C_{47}^p, C_{46}^p,$ $C_{45}^p, C_{43}^p, C_{42}^p$.	$C_{49}^p > C_{48}^p > C_{47}^p$; $C_{43}^p > C_{42}^p$ $C_{49}^p > C_{46}^p > C_{43}^p$;	$C_{46}^p > C_{45}^p$; $C_{48}^p > C_{45}^p > C_{42}^p$.
$R_6^p : x?v \xrightarrow{p} x!v'$	C_{69}^p, C_{68}^p.	$C_{69}^p > C_{68}^p$.	

Table 1: Consistency models and hierarchy using program-ordered guards

All these augmented views are obtained by applying a unique augmentation rule. Other views can be obtained by applying multiple augmentation rules. As an example, we can combine rules $R_1^p \Rightarrow R_9$ and $R_4^p \Rightarrow R_9$, which produce the rule $x!v/x?v \xrightarrow{p} x?v' \Rightarrow X?v \rightarrow x!v'$, to augment GV and obtain another weak consistency model. A more complex rule can be obtained by the combination of $R_4^p \Rightarrow R_9$ ($x?v \xrightarrow{p} x?v' \Rightarrow X?v \rightarrow x!v'$) and $R_1^p \Rightarrow R_3$ ($x!v \xrightarrow{p} x?v' \Rightarrow x!v \rightarrow x!v'$). Many other combinations are possible and are not discussed in detail here.

All these augmentation rules are used to generate a series of consistency models that we identify under the class of *local consistency models* since they are based on local augmentation rules. Based on the rules, we have identified the following family of local consistency models:

- Local read consistency models (LRC)

  Local read consistency models are all based on local augmentation rules that use relations R_1^p and R_4^p as guards ($x?v/x!v \xrightarrow{p} x?v'$). For examples, C_{13}^p is one possible LRC consistency model as well as the combination of C_{13}^p and C_{46}^p.

- Local write consistency models (LWC)

Local write consistency models are all based on local augmentation rules that use relations R_3^p and R_6^p as guards $(x?v/x!v \xrightarrow{p} x!v')$. For example, C_{36}^p is one possible LWC models and the combination of C_{36}^p and C_{69}^p is another one.

- Local consistency models (LC)

  General local consistency models are all based on combined local augmentation rules. The guards can generally take the form $x?v/x!v \xrightarrow{p} x?v'/x!v'$. So any combination of LRC and LWC models can be used to specify LC models.

The above leads to many consistency models. Not all of them are useful or efficiently implementable. However, in Chapter 6 we give algorithms that implement some of these models.

4.2.4 Consistency Models from Global Augmentation Rules

The previous section uses R_i^p as the basis of augmentation. The program-ordered restriction of the guard can be relaxed to generate more augmentations and hence stronger consistency models. This section presents these results.

A major difference arises when the guard R_i is not required to be in program order: now all cases of R_i (instead of just the four cases in Section 4.2.3) can be used as the guard for further augmentation. This is because a global view can meaningfully satisfy every R_i. For example, R_8 $(X?v \rightarrow X?v')$ is satisfied in GV when every occurrence of $x?v$ can reach every occurrence of $x?v'$ in GV. Augmented global view GV_{ij} and consistency model C_{ij} can be defined similar to the ones defined in Section 4.2.3.

Table 2 gives the consistency models derivable from all the global rules. Using R_1, R_3, R_4, and R_6, we derive respectively five (GV_{12}, GV_{13}, GV_{17}, GV_{18} and GV_{19}), two (GV_{38} and GV_{39}), seven (GV_{42}, GV_{43}, GV_{45}, GV_{46}, GV_{47}, GV_{48} and GV_{49}) and two (GV_{68} and GV_{69}) augmented views which are used to define the consistency models presented in Table 2. These models are similar to the one deduced from the local counterpart of these guards. Using R_2, R_5, R_7 and R_8 we deduce respectively three (GV_{23}, GV_{28} and GV_{29}), four (GV_{53}, GV_{56}, GV_{58} and GV_{59}), four (GV_{72}, GV_{73}, GV_{78} and GV_{79}) and two (GV_{83} and GV_{89}) new augmented views. Figure 25, 26, 27 and

28 show some of these augmented views. All the other cases either use a specific operation that cannot be meaningfully applied with the guard, or are equivalent to GV.

Guard	Consistency models	Hierarchies based on the guard	
R_1	$C_{12}, C_{13}, C_{17}, C_{18}, C_{19}$	$C_{19} > C_{18} > C_{17}$; $C_{19} > C_{13}$;	$C_{13} > C_{12}$; $C_{18} > C_{12}$.
R_2	C_{23}, C_{28}, C_{29}	$C_{29} > C_{28}$;	$C_{29} > C_{23}$.
R_3	C_{38}, C_{39}	$C_{39} > C_{38}$.	
R_4	$C_{42}, C_{43}, C_{45}, C_{46}, C_{47}, C_{48}, C_{49}$	$C_{49} > C_{48} > C_{47}$; $C_{43} > C_{42}$; $C_{49} > C_{46} > C_{43}$;	$C_{46} > C_{45}$; $C_{48} > C_{45} > C_{42}$.
R_5	$C_{53}, C_{56}, C_{58}, C_{59}$	$C_{59} > C_{58}$;	$C_{59} > C_{56} > C_{53}$.
R_6	C_{68}, C_{69}	$C_{69} > C_{68}$.	
R_7	$C_{72}, C_{73}, C_{78}, C_{79}$	$C_{79} > C_{78}$; $C_{79} > C_{73}$;	$C_{73} > C_{72}$; $C_{78} > C_{72}$.
R_8	C_{83}, C_{89}	$C_{89} > C_{83}$.	

Table 2: Consistency models using global guards

a) GV

b) GV_{13}

c) GV_{19}

Figure 25: Augmented views based on Rule R_1

A hierarchy of consistency models can be defined with Theorem 6 extended as follows:

Corollary 1 : *If* $R_j > R_k$ *then* $C_{ij} > C_{ik}$.

Table 2 summarizes the eight hierarchies of consistency models derivable from global augmentation rules. Following the same reasoning as used in deriving the various consistency models of a hierarchy using R_i as guard, C_{ij} will be a consistency model only if under the guard R_i, R_j can be meaningfully applied to a global view

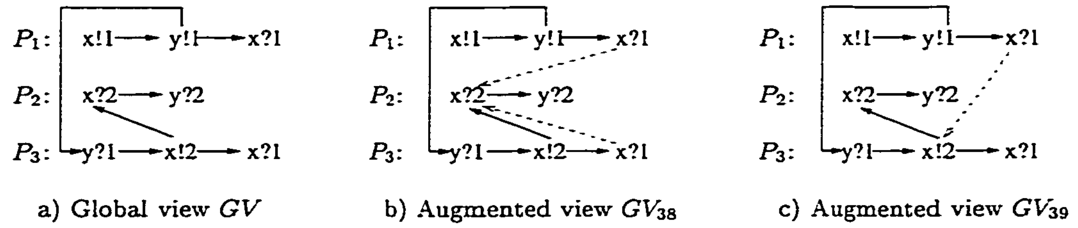


Figure 26: Augmented views based on Rule R_3

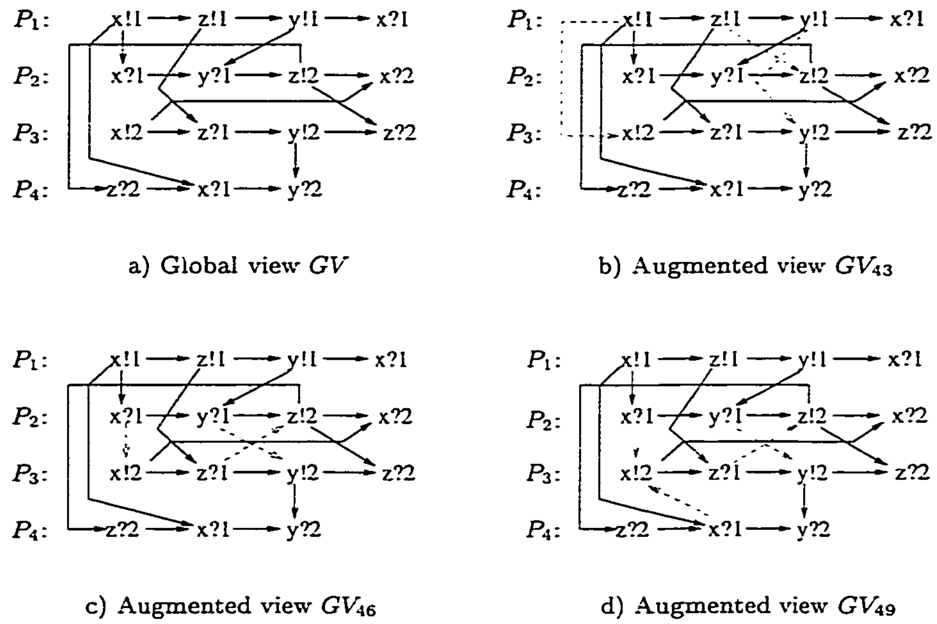


Figure 27: Augmented views based on Rule R_4

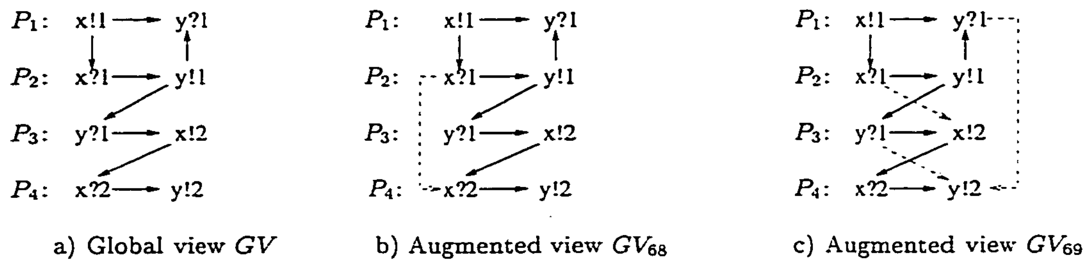


Figure 28: Augmented views based on Rule R_6

GV, i.e., without introducing inconsistency. For example, C_{19} is a consistency model obtained using $R_1(x!v \rightarrow x?v') \Rightarrow R_9(X?v \rightarrow x!v')$: whenever an $x!v$ can reach an occurrence of $x?v'$ in GV, we augment GV so that every occurrence of $x?v$ can reach the occurrence of $x!v'$. However, not all C_{ij} are consistency models. For example, C_{14} is not a consistency model obtainable: $R_1(x!v \rightarrow x?v') \Rightarrow R_4(x?v \rightarrow x!v')$ cannot be meaningfully applied given R_1; given R_1, the instance of $x?v$ is not uniquely identifiable in GV for the augmentation. Because of this, many of the hierarchies shown in Table 2 consist of fewer model candidates. Table 2 also does not show consistency models C_{ij} which are indistinct from the minimal consistency given by an acyclic GV, according to the following theorem.

Theorem 7 : *A consistency model C_{ij} is equivalent to minimal consistency (acyclic GV) whenever $R_i > R_j$.*

Proof: This follows trivially from Lemma 21. Whenever $R_i > R_j$, then GV that satisfies R_i also satisfies R_j. Hence the claim. ■

Unlike hierarchies obtained using program-ordered guards, the eight hierarchies in Table 2 are related. Indeed together they actually form a single composite hierarchy due to the following theorem.

Theorem 8 : *If $R_i > R_j$ and C_{ik} and C_{jk} are both defined, then $C_{ik} > C_{jk}$.*

Proof: This also follows from Lemma 21. If $R_i > R_j$, and both augmentations are defined, then every augmentation in GV_{jk} can be found in GV_{ik}. Hence if GV_{ik} is acyclic then GV_{jk} will also be acyclic, or equivalently $C_{ik} > C_{jk}$. ■

Using Theorem 8, we can integrate the eight hierarchies in Table 2 to form a single hierarchy under the ordering relation ">" among consistency models. The details are not further elaborated. Table 3 shows the hierarchy based on Theorem 8 with all the models listed in Table 2 and Figure 29 shows the single hierarchy formed by all the models.

Again, even if no example is given, it is possible to combine multiple augmentation rules to specify more complex consistency models.

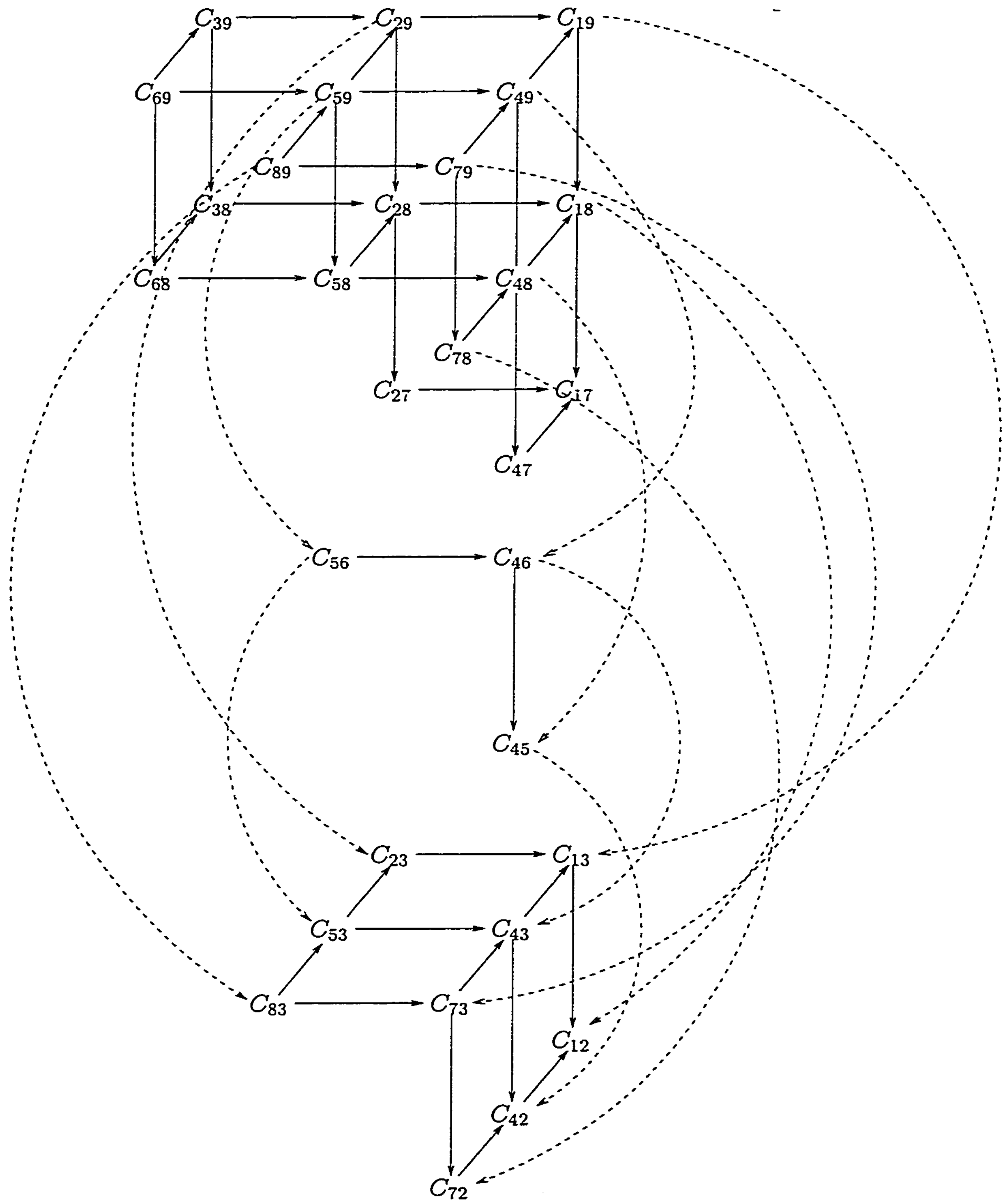


Figure 29: Global hierarchy

Consequence R_i	Consistency Hierarchies based on R_i		
$R_9 : X?v \rightarrow x!v'$	$C_{89} > C_{79}$;	$C_{69} > C_{59} > C_{49}$;	$C_{39} > C_{29} > C_{19}$;
	$C_{69} > C_{39}$.	$C_{89} > C_{59} > C_{29}$;	$C_{79} > C_{49} > C_{19}$;
$R_8 : X?v \rightarrow X?v'$	$C_{68} > C_{58} > C_{48}$;	$C_{38} > C_{28} > C_{18}$;	
	$C_{68} > C_{38}$.	$C_{58} > C_{28}$;	$C_{48} > C_{18}$;
$R_7 : X?v \rightarrow x?v'$	$C_{47} > C_{17}$;		
$R_6 : x?v \rightarrow x!v'$	$C_{56} > C_{46}$.		
$R_3 : x!v \rightarrow x!v'$	$C_{83} > C_{73}$;	$C_{53} > C_{43}$;	$C_{23} > C_{13}$;
	$C_{83} > C_{53} > C_{23}$;	$C_{73} > C_{43} > C_{13}$;	
$R_2 : x!v \rightarrow X?v'$	$C_{72} > C_{42} > C_{12}$;		

Table 3: Consistency hierarchies using consequences

All these augmentation rules are used to generate a series of consistency models that we identify under the class of *global consistency models* since they are based on global augmentation rules. Based on the rules, we have identified the following family of global consistency models:

- Global read consistency models

  Global read consistency models use as guards the global augmentation rules R_1, R_2, R_4, R_5, R_7 and R_8.

- Global write consistency models

  Global write consistency models use as guards the global augmentation rules R_3 and R_6.

- Global consistency models

  Global consistency models include all the models mentioned earlier. The more restrictive of these global consistency models is based on the necessary view. So, we call this particular consistency model, Necessary Consistency.

4.3 Weak Consistency Models Without Augmentation

An acyclic global view can be used to define other consistency hierarchies without applying augmentation rules to GV. Alternately, a consistency model can be defined by asserting some additional property to be satisfied in GV. This section will introduce one such interesting hierarchy.

Suppose a given global view GV satisfies some R_i. In Section 4.2, augmentation is used to produce GV_{ij} (for some $R_j > R_i$). Here, instead of changing GV, we define a new consistency model by requiring that GV does not violate R_j. The non violation of a reachability relation is expressed by the converse of the reachability relation denoted $R_{\bar{j}}$. We have nine "non-reachability" relations:

$$R_{\bar{9}} : x!v' \nrightarrow X?v \quad R_{\bar{8}} : X?v' \nrightarrow X?v \quad R_{\bar{7}} : x?v' \nrightarrow X?v$$
$$R_{\bar{6}} : x!v' \nrightarrow x?v \quad R_{\bar{5}} : X?v' \nrightarrow x?v \quad R_{\bar{4}} : x?v' \nrightarrow x?v$$
$$R_{\bar{3}} : x!v' \nrightarrow x!v \quad R_{\bar{2}} : X?v' \nrightarrow x!v \quad R_{\bar{1}} : x?v' \nrightarrow x!v$$

The hierarchy introduced in Lemma 21 is also still valid on their reverse counterpart.

Lemma 22 : *If $R_i > R_j$ in the hierarchy then $R_{\bar{i}} > R_{\bar{j}}$*

Proof: The proof is similar to the one of Lemma 21 and is not shown here. ■

So, similarly to the ordering established among the reachability relations R_9 through R_1, the following order exists among these non-reachability relations :

$$R_{\bar{9}} > R_{\bar{8}} > R_{\bar{7}}; \qquad R_{\bar{6}} > R_{\bar{5}} > R_{\bar{4}};$$
$$R_{\bar{3}} > R_{\bar{2}} > R_{\bar{1}}; \qquad R_{\bar{9}} > R_{\bar{6}} > R_{\bar{3}};$$
$$R_{\bar{8}} > R_{\bar{5}} > R_{\bar{2}}; \qquad R_{\bar{7}} > R_{\bar{4}} > R_{\bar{1}}.$$

where $R_{\bar{i}} > R_{\bar{j}}$ means that if a relation does not exist between two operations in $R_{\bar{i}}$ then it also does not exist in $R_{\bar{j}}$.

As for the reachability relations, these "non-reachability" relations can also be limited to program-order and are then noted $R^p_{\bar{i}}$. Obviously, the hierarchy defined earlier between the nine non-reachability relations also applies to their local counterparts.

The additional property to be satisfied in GV is expressed by a guarded command: $R_i \Rightarrow R_{\bar{j}}$ where R_i is the guard and $R_{\bar{j}}$ the restriction. A consistency model $C_{i\bar{j}}$ is definable by requiring GV to be acyclic and to satisfy the rule $R_i \Rightarrow R_{\bar{j}}$. These weak consistency models are all related to *causal memory*.

As an example, consider $C_{4\bar{9}}$ (weak consistency model with R_4 as guard and the converse of R_9 as the restriction). Under $C_{4\bar{9}}$, we require that whenever an instance R_4 exists in GV, then $R_{\bar{9}}$ must hold in GV. Specifically, if $x?v \rightarrow x?v'$ in GV (i.e., R_4), then GV cannot contain any other instance of $x?v$ such that $x!v' \rightarrow x?v$. Figure 30 illustrates an example of a global view that violates $C_{4\bar{9}}$. Intuitively, $C_{4\bar{9}}$ requires that whenever an instance of $x?v$ is ordered before an instance of $x?v'$, then $x!v'$ cannot be ordered before any other instance of $x?v$ in GV. The latter is asserted by taking $R_{\bar{9}}$ involving the write $x!v'$ and an arbitrary instance of $x?v$. So through value-order or program-order, the ordering of "values" in an object is "consistent" in GV.

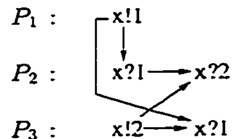


Figure 30: An execution that violates $C_{4\bar{9}}$

Some $C_{i\bar{j}}$ is redundant: an acyclic GV already ensures it. For example, $C_{4\bar{1}}$ is redundant: given $x?v \rightarrow x?v'$ in GV, $x?v' \rightarrow x!v$ cannot be in GV without creating a cyclic GV. Hence minimal consistency already includes $C_{4\bar{1}}$. On the other hand, some $C_{i\bar{j}}$ is undefined: given the operations in R_i, we cannot assert additional properties of GV using R_j. For example, $C_{1\bar{4}}$ is undefined: given $x!v \rightarrow x?v'$ (R_1) in GV, it is not meaningful to associate a particular $x?v$ not referenced in R_1 but which is required in R_4. A different example is $C_{2\bar{4}}$ which is also undefined. Indeed, given $x!v \rightarrow X?v'$ (R_2) in GV, it is not meaningful to associate a particular $x?v'$ not referenced in R_2

but which is required in R_4. In the rest of the thesis, every reference of $C_{i\bar{j}}$ is assumed to be defined and meaningful.

Theorem 9 : *If $R_j > R_k$, then $C_{i\bar{j}} > C_{i\bar{k}}$ and $C_{k\bar{i}} > C_{j\bar{i}}$.*

Proof: We show that every execution allowed in $C_{i\bar{j}}$ is also allowed in $C_{i\bar{k}}$. Suppose otherwise and that there exists a converse instance of R_j. But it would also be a converse instance of R_k ($R_j > R_k$). Hence $C_{i\bar{j}} > C_{i\bar{k}}$. Similarly we can prove that $C_{k\bar{i}} > C_{j\bar{i}}$. ■

As for the reachability relations, the non-reachability relations can also be limited to program-order (local relations) and are then noted $R^p_{\bar{i}}$. As before, when limited to program order, R^p_1, R^p_3, R^p_4 and R^p_6 are the only relations usable as guard. However for the non-reachability relations limited to program order, $X?v$ is interpreted as all the read operations in a single process. For example, $R^p_{\bar{1}} : x?v' \stackrel{p}{\not\rightarrow} x!v$ is interpreted as a particular read of value v' must not precede the write of value v, while $R^p_{\bar{2}} : X?v' \stackrel{p}{\not\rightarrow} x!v$ is interpreted as no read of v' from process P_i must precede the write of v by P_i. Obviously, the hierarchy defined earlier between the nine non-reachability relations also applies between their local counterparts. The local and global reachability and non-reachability relations can be used to create different types of restriction each of which being able to define consistency models based on:

1. Global properties obtained with global guards and global restrictions ($R_i \Rightarrow R_{\bar{j}}$). Such models are noted $C_{i\bar{j}}$.

2. Mixed properties obtained with local guards and global restrictions ($R^p_i \Rightarrow R_{\bar{j}}$). Such models are noted $C^p_{i\bar{j}}$.

3. Mixed properties obtained with global guards and local restrictions ($R_i \Rightarrow R^p_{\bar{j}}$). Such models are noted $C^{-p}_{i\bar{j}}$.

4. Local properties obtained with local guards and local restrictions ($R^p_i \Rightarrow R^p_{\bar{j}}$). Such models are noted $C^{pp}_{i\bar{j}}$.

Again, the basic rules presented in the next sections can be combined to produce more complex rules. These rules produce other consistency models that can also be part of the family of causal memory. This part is not elaborated here.

4.3.1 Global Properties ($R_i \Rightarrow R_{\bar{j}}$)

Relations R_1 through R_8 are used as guards in these relations to produce the consistency models $C_{i\bar{j}}$ presented in Table 4 and the hierarchies obtainable according to Theorem 9. The similarities between this hierarchy and the ordering hierarchy formed by R_i and the augmentation rules is noteworthy.

R_i	Consistency models	Consistency Hierarchy	
R_1: $x!v \rightarrow x?v'$	$C_{1\bar{9}}, C_{1\bar{8}}, C_{1\bar{7}}, C_{1\bar{3}}, C_{1\bar{2}}$	$C_{1\bar{9}} > C_{1\bar{8}} > C_{1\bar{7}}$; $C_{1\bar{9}} > C_{1\bar{3}}$;	$C_{1\bar{3}} > C_{1\bar{2}}$ $C_{1\bar{8}} > C_{1\bar{2}}$.
R_2: $x!v \rightarrow X?v'$	$C_{2\bar{9}}, C_{2\bar{8}}, C_{2\bar{3}}$	$C_{2\bar{9}} > C_{2\bar{8}}$;	$C_{2\bar{9}} > C_{2\bar{3}}$.
R_3: $x!v \rightarrow x!v'$	$C_{3\bar{9}}, C_{3\bar{8}}$	$C_{3\bar{9}} > C_{3\bar{8}}$.	
R_4: $x?v \rightarrow x?v'$	$C_{4\bar{9}}, C_{4\bar{8}}, C_{4\bar{7}}, C_{4\bar{6}}$, $C_{4\bar{5}}, C_{4\bar{3}}, C_{4\bar{2}}$	$C_{4\bar{9}} > C_{4\bar{8}} > C_{4\bar{7}}$; $C_{4\bar{3}} > C_{4\bar{2}}$; $C_{4\bar{9}} > C_{4\bar{6}} > C_{4\bar{3}}$;	$C_{4\bar{6}} > C_{4\bar{5}}$; $C_{4\bar{8}} > C_{4\bar{5}} > C_{4\bar{2}}$.
R_5: $x?v \rightarrow X?v'$	$C_{5\bar{9}}, C_{5\bar{8}}, C_{5\bar{6}}, C_{5\bar{3}}$	$C_{5\bar{9}} > C_{5\bar{8}}$;	$C_{5\bar{9}} > C_{5\bar{6}} > C_{5\bar{3}}$;
R_6: $x?v \rightarrow x!v'$	$C_{6\bar{9}}, C_{6\bar{8}}$	$C_{6\bar{9}} > C_{6\bar{8}}$.	
R_7: $X?v \rightarrow x?v'$	$C_{7\bar{9}}, C_{7\bar{8}}, C_{7\bar{3}}, C_{7\bar{2}}$	$C_{7\bar{9}} > C_{7\bar{8}}$; $C_{7\bar{9}} > C_{7\bar{3}}$;	$C_{7\bar{3}} > C_{7\bar{2}}$; $C_{7\bar{8}} > C_{7\bar{2}}$.
R_8: $X?v \rightarrow X?v'$	$C_{8\bar{9}}, C_{8\bar{3}}$	$C_{8\bar{9}} > C_{8\bar{3}}$.	

Table 4: Weak consistency models and hierarchies based on global properties

As before, these basic rules can be combined to obtain more complex rules and then different consistency models.

These weak consistency models are related to causality of the reads and writes via the global view. It is interesting to note that the causal memory, as defined by Ahamad et al. [7] and used in [5, 8, 10, 36, 50, 49], is equivalent to the weak consistency model that satisfies the property $x!v/x?v \rightarrow x!v'/x?v' \Rightarrow x!v'/x?v' \not\rightarrow X?v$. This intuitively means that if a write or a read of a value v precedes the write or read of the same object with another value v', then the latter operation cannot precede any read of the former value. Another slightly less restrictive form of causal memory was introduced in [9]. This causal memory is different since the causality is

defined only relative to write operations. This "write" causal memory corresponds to the weak consistency involving $x!v/x?v \rightarrow x!v' \Rightarrow x!v' \nrightarrow X?v$. This causal memory is less restrictive than the preceding one and the authors have provided a non-blocking implementation based on timestamps and remote writes and local reads.

4.3.2 Mixed Properties ($R_i^p \Rightarrow R_{\bar{j}}$)

Relations R_1^p, R_3^p, R_4^p and R_6^p are used as guards in these properties to produce the consistency models $C_{i\bar{j}}^p$. Table 5 shows the possible mixed properties and the hierarchies obtainable according to Theorem 9. It should be noted that these models are strictly weaker than their global counterpart, and so form a sub-hierarchy of the latter.

R_i	Consistency models	Consistency Hierarchy
R_1^p: $x!v \xrightarrow{p} x?v'$	$C_{1\bar{9}}^p, C_{1\bar{8}}^p, C_{1\bar{7}}^p, C_{1\bar{3}}^p, C_{1\bar{2}}^p$	$C_{1\bar{9}}^p > C_{1\bar{8}}^p > C_{1\bar{7}}^p$; $C_{1\bar{3}}^p > C_{1\bar{2}}^p$ $C_{1\bar{9}}^p > C_{1\bar{3}}^p$; $C_{1\bar{8}}^p > C_{1\bar{2}}^p$.
R_3^p: $x!v \xrightarrow{p} x!v'$	$C_{3\bar{9}}^p, C_{3\bar{8}}^p$	$C_{3\bar{9}}^p > C_{3\bar{8}}^p$.
R_4^p: $x?v \xrightarrow{p} x?v'$	$C_{4\bar{9}}^p, C_{4\bar{8}}^p, C_{4\bar{7}}^p, C_{4\bar{6}}^p,$ $C_{4\bar{5}}^p, C_{4\bar{3}}^p, C_{4\bar{2}}^p$	$C_{4\bar{9}}^p > C_{4\bar{8}}^p > C_{4\bar{7}}^p$; $C_{4\bar{6}}^p > C_{4\bar{5}}^p$; $C_{4\bar{3}}^p > C_{4\bar{2}}^p$; $C_{4\bar{9}}^p > C_{4\bar{6}}^p > C_{4\bar{3}}^p$; $C_{4\bar{8}}^p > C_{4\bar{5}}^p > C_{4\bar{2}}^p$.
R_6^p: $x?v \xrightarrow{p} x!v'$	$C_{6\bar{9}}^p, C_{6\bar{8}}^p$	$C_{6\bar{9}}^p > C_{6\bar{8}}^p$.

Table 5: Weak consistency models and hierarchies based on local guards

As before, these basic rules can be combined to obtain more complex properties and derive other consistency models.

4.3.3 Mixed Properties ($R_i \Rightarrow R_{\bar{j}}^p$)

Relations R_1 through R_8 are again used as guards in these properties to produce the consistency models $C_{i\bar{j}}^{-p}$. Since we can use all the non-reachability relations as restrictions, we deduce many consistency models. Table 6 shows the possible mixed properties and the hierarchies obtainable according to Theorem 9. In this case, given R_i in GV, the absence of $R_{\bar{j}}$ in GV is restricted to program-order. For example, $GV_{3\bar{9}}^{-p}$: $x!v \rightarrow x!v' \Rightarrow x!v' \overset{p}{\nrightarrow} X?v$ in GV means that if the write of the value v

precedes the write of value v', then the write of v' must never precede any read of v in the process that issues the write.

It should be noted that these models are strictly weaker than their global counterpart and so form a sub-hierarchy of the latter.

R_i	Consistency models	Weak Consistency Hierarchy
R_1: $x!v \rightarrow x?v'$	$C^{-p}_{1\bar{9}}, C^{-p}_{1\bar{8}}, C^{-p}_{1\bar{7}}, C^{-p}_{1\bar{3}}, C^{-p}_{1\bar{2}}$	$C^{-p}_{1\bar{9}} > C^{-p}_{1\bar{8}} > C^{-p}_{1\bar{7}}$; $C^{-p}_{1\bar{3}} > C^{-p}_{1\bar{2}}$ $C^{-p}_{1\bar{9}} > C^{-p}_{1\bar{3}}$; $C^{-p}_{1\bar{8}} > C^{-p}_{1\bar{2}}$.
R_2: $x!v \rightarrow X?v'$	$C^{-p}_{2\bar{9}}, C^{-p}_{2\bar{8}}, C^{-p}_{2\bar{3}}$	$C^{-p}_{2\bar{9}} > C^{-p}_{2\bar{8}}$; $C^{-p}_{2\bar{9}} > C^{-p}_{2\bar{3}}$.
R_3: $x!v \rightarrow x!v'$	$C^{-p}_{3\bar{9}}, C^{-p}_{3\bar{8}}$	$C^{-p}_{3\bar{9}} > C^{-p}_{3\bar{8}}$.
R_4: $x?v \rightarrow x?v'$	$C^{-p}_{4\bar{9}}, C^{-p}_{4\bar{8}}, C_{4\bar{7}}, C^{-p}_{4\bar{6}}, C^{-p}_{4\bar{5}}, C_{4\bar{3}}, C^{-p}_{4\bar{2}}$	$C^{-p}_{4\bar{9}} > C^{-p}_{4\bar{8}} > C^{-p}_{4\bar{7}}$; $C^{-p}_{4\bar{6}} > C^{-p}_{4\bar{5}}$; $C^{-p}_{4\bar{3}} > C_{4\bar{2}}$; $C^{-p}_{4\bar{9}} > C^{-p}_{4\bar{6}} > C^{-p}_{4\bar{3}}$; $C^{-p}_{4\bar{8}} > C^{-p}_{4\bar{5}} > C^{-p}_{4\bar{2}}$.
R_5: $x?v \rightarrow X?v'$	$C^{-p}_{5\bar{9}}, C^{-p}_{5\bar{8}}, C^{-p}_{5\bar{6}}, C^{-p}_{5\bar{3}}$	$C^{-p}_{5\bar{9}} > C^{-p}_{5\bar{8}}$; $C^{-p}_{5\bar{9}} > C^{-p}_{5\bar{6}} > C^{-p}_{5\bar{3}}$;
R_6: $x?v \rightarrow x!v'$	$C^{-p}_{6\bar{9}}, C^{-p}_{6\bar{8}}$	$C^{-p}_{6\bar{9}} > C^{-p}_{6\bar{8}}$.
R_7: $X?v \rightarrow x?v'$	$C^{-p}_{7\bar{9}}, C^{-p}_{7\bar{8}}, C^{-p}_{7\bar{3}}, C^{-p}_{7\bar{2}}$	$C^{-p}_{7\bar{9}} > C_{7\bar{8}}$; $C^{-p}_{7\bar{3}} > C^{-p}_{7\bar{2}}$; $C^{-p}_{7\bar{9}} > C^{-p}_{7\bar{3}}$; $C^{-p}_{7\bar{8}} > C^{-p}_{7\bar{2}}$.
R_8: $X?v \rightarrow X?v'$	$C^{-p}_{8\bar{9}}, C^{-p}_{8\bar{3}}$	$C^{-p}_{8\bar{9}} > C^{-p}_{8\bar{3}}$.

Table 6: Weak consistency models and hierarchies based on local restrictions

As before, these basic rules can be combined to obtain more complex properties and derive other consistency models.

4.3.4 Local Properties ($R^p_i \Rightarrow R^p_{\bar{j}}$)

Relations R^p_1, R^p_3, R^p_4 and R^p_6 are used as guards in these properties to produce the consistency models $C^{pp}_{i\bar{j}}$. Table 7 shows the consistency models derivable from these properties and the associated hierarchies. Because of the program order restriction, this hierarchy of models is weaker than the previous ones without full program restriction in both the guard and the consequence.

Many consistency models are not included in the table because they are redundant with the well ordering restrictions imposed on admissible local views. Indeed, the rules $C^{pp}_{1\bar{2}}$, $C^{pp}_{1\bar{3}}$, $C^{pp}_{1\bar{7}}$, $C^{pp}_{3\bar{9}}$, $C^{pp}_{4\bar{5}}$, $C^{pp}_{4\bar{6}}$,$C^{pp}_{4\bar{7}}$ and $C^{pp}_{6\bar{9}}$ are redundant with the concept of admissible local view. For example, $C^{pp}_{1\bar{3}}$ is derived from the property $x!v \xrightarrow{p} x?v' \Rightarrow x!v' \stackrel{p}{\not\rightarrow} x!v$

which means that a read must return that most recent value written (locally). All the other models are very weak but useful. For example, $C^{pp}_{\overline{38}}$, derived from the property $x!v \xrightarrow{p} x!v' \Rightarrow X?v' \overset{p}{\not\rightarrow} X?v$, may seem very weak but it can be used to specify object based FIFO channels which are similar to slow memory [33].

R_i	Consistency models	Weak Consistency Hierarchy
R^p_1: $x!v \xrightarrow{p} x?v'$	$C^{pp}_{\overline{19}}, C^{pp}_{\overline{18}}$	$C^{pp}_{\overline{19}} > C^{pp}_{\overline{18}}$;
R^p_3: $x!v \xrightarrow{p} x!v'$	$C^{pp}_{\overline{38}}$	
R^p_4: $x?v \xrightarrow{p} x?v'$	$C^{pp}_{\overline{49}}, C^{pp}_{\overline{48}}, C^{pp}_{\overline{43}}, C^{pp}_{\overline{42}}$,	$C^{pp}_{\overline{49}} > C^{pp}_{\overline{48}}$; $C^{pp}_{\overline{43}} > C^{pp}_{\overline{42}}$; $C^{pp}_{\overline{49}} > C^{pp}_{\overline{43}}$; $C^{pp}_{\overline{48}} > C^{pp}_{\overline{42}}$.
R^p_6: $x?v \xrightarrow{p} x!v'$	$C^{pp}_{\overline{68}}$	

Table 7: Weak consistency models and hierarchies on locals guards and restrictions

As before, these basic rules can be combined to obtain more complex properties and derive different consistency models.

4.4 Conclusion

In this chapter. we have presented multiple hierarchies of consistency models using the global view as a seed. All the models are derived from augmentation rules or properties asserted on the global view.

Other consistency models may be derivable from the global view by applying subset restrictions as presented in Chapter 2. Many subset restrictions (projections) can be performed on the basic global view or on any augmented global view, including the necessary and possible views. Some examples of projections are:

1. $GV|_o$ is the global view projected on object o;

2. $GV|_p$ is the global view projected on process p;

3. $GV_{ij}|_o$ is some augmented global view projected on object o;

4. $GV_{ij}|_p$ is some augmented global view projected on process p;

However the usefulness of these projected views is questionable from the perspective of removing cycles in the original view. For example, to remove a cycle in a view, all the objects referenced in the cycle have to be removed in the projection operation. We leave this exploration to future work.

Chapter 5

DSM Consistency Models Based on Partial Views

In the preceding chapter, we have presented many consistency models derivable from global views. It is possible to define consistency models based on partial views that contain only a subset of the events. In this chapter, we present some of these more restrictive models. They are weaker because of the partiality of views used.

5.1 Restricted Views

Restricted views involve only a subset of the events in an execution. They are selected by restricting the choices on (i) process, or (ii) object, or (iii) type of event. Such "projection" operations [28] applied on local views result in restricted local views.

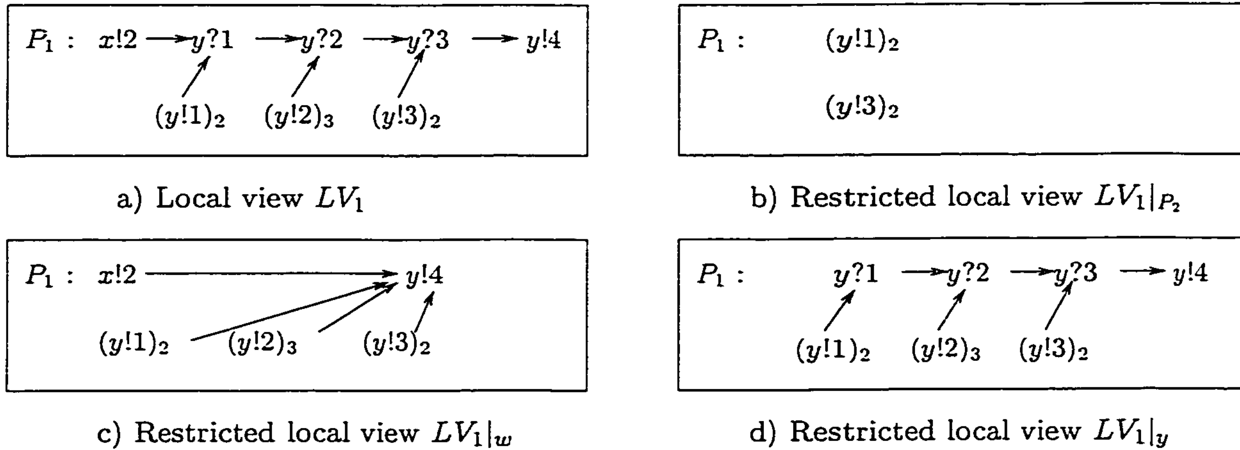
5.1.1 Restricted Local Views

A restricted local view can be obtained by applying subset restriction (event projection) on the local view. Restriction can be based on one of the three criteria: (i) subset of objects, (ii) subset of processes, or (iii) type of operation. Unfortunately, if we apply restrictions directly to local views and later compose these views to form a partial (global) view, too many ordering relations may be lost in the restriction.

Such a partial view may not lead to interesting consistency models. Figure 31 and 32 illustrate this. Figure 31 shows an execution. The local view of process P_1, LV_1, is shown in Figure 32(a). If we apply process restriction to LV_1, in particular, restricting LV_1 to events in process P_2, we get $LV_1|_{P_2}$ which is shown in Figure 32(b). $LV_1|_{P_2}$ shows two concurrent writes from P_2. In comparing LV_1 with $LV_1|_{P_2}$, we observe that even though the two writes from P_2 are still concurrent in LV_1, these two writes are indirectly related by the reads in P_1 and their ordering (consistency) can still be inferred later in the use of LV_1. But $LV_1|_{P_2}$ has completely removed this valuable piece of information, and significantly weakens itself from being useful in consistency requirement. Figure 32(c) and (d) shows the restriction of LV_1 to write events, and to events involving object y, respectively.

P_1 :	$x!2$; $x?1$; $y?1$; $y?2$; $y!3$
P_2 :	$x!1$; $y!2$; $x?1$
P_3 :	$y!1$

Figure 31: Sequence of memory operations by three processes



a) Local view LV_1

b) Restricted local view $LV_1|_{P_2}$

c) Restricted local view $LV_1|_w$

d) Restricted local view $LV_1|_y$

Figure 32: Local view LV_1 and some restricted local views

To overcome the above deficiency of a restricted local view, we propose to augment a local view before applying the restriction. The augmentation used is the direct correspondence of the rule used to produce the necessary global view in Chapter 4.

The only difference is that it is applied to a local view instead. For this distinction, we call it Rule 2. Our strategy is to augment local views, to become necessary local views, before restricting them to a subset of events. We can then take the union of these restricted necessary local views to form partial (global) views.

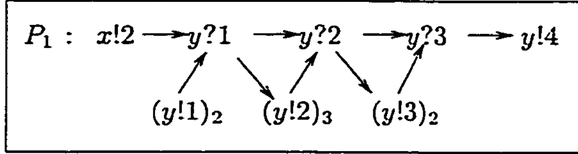
Rule 2 : $(x!v)_i/(x?v)_i \xrightarrow{p} (x?v')_i \Rightarrow (x!v)_i/(x?v)_i \rightarrow (x!v')_j$

This rule is a combination of the two augmentation rules $R_1^p \Rightarrow R_3$ and $R_4^p \Rightarrow R_6$ as presented in Chapter 4. It orders every read of x that returns the value of v, or the single write of the new value v, to appear before the external write of v' into x, if the write of the former is ordered in process P_i before some read of the latter. It corresponds to the maximal ordering among the events directly "perceived" by the process P_i.

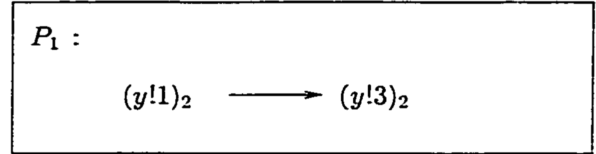
Definition 27 : *A local view LV_i augmented with Rule 2 is called a* **necessary local view** NLV_i.

We use the name necessary local view because the local view cannot be consistent if it does not satisfy it. Indeed, Rule 2 is just the localized version of Rule 1. Since external writes are not ordered, the necessary local view is a partial order. Figure 33(a) shows the necessary local view of process P_2 from the example execution of Figure 31. Figure 33(b), (c) and (d) shows the restricted local views obtained from the necessary local view. We use restrictions on necessary local views rather than local views as the former preserve all ordering perceived by a process and are more useful in defining consistency models.

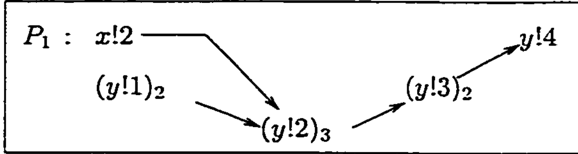
Applying this rule before or after the formation of the global view produces exactly the same augmented view. Hence, $(\cup_i NLV_i) = GV^p_{(13,46)}$ where $GV^p_{(13,46)}$ represents the global view augmented with the rules $R_1^p \Rightarrow R_3$ and $R_4^p \Rightarrow R_6$. This means that every restricted local view derived from the necessary local view may be used to specify a consistency model strictly weaker than the consistency model $C^p_{(13,46)}$ which is derived from the augmented global view $GV^p_{(13,46)}$. This consistency model is one



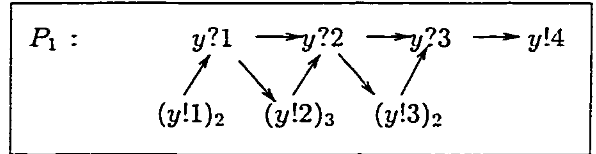
a) Necessary local view NLV_1



b) Restricted necessary local view $NLV_1|_{P_2}$



c) Restricted necessary local view $LV_1|_w$



d) Restricted necessary local view $NLV_1|_y$

Figure 33: Necessary local view and its restricted views

of the models included in the class of local read consistency models introduced in the Chapter 4.

Instead of Rule 2, other restricted views are possible by applying $R_1^p \Rightarrow R_3$ or $R_4^p \Rightarrow R_6$. These new augmented local views can then be used to produce consistency models weaker than those obtainable from GV_{13}^p (C_{13}^p) and GV_{46}^p (C_{46}^p). However, these models are not explored in this thesis.

5.2 Consistency Models Based on Partial Views

A restricted necessary local view is one obtained by selecting a subset of the events in the necessary local view. A partial (global) view is obtained by taking the union of some restricted necessary local views. To derive consistency models that are based on partial views, we use partial views involving restrictions on a subset of necessary local views. A consistency model is definable by requiring a partial view to remain acyclic. In this section, we present these consistency models obtained from partial views. They can be classified into three types:

- object relative consistency: they are obtained by restricting necessary local views to selected objects.

- process relative consistency: they are obtained by restricting necessary local views on a subset of processes.
- mixed consistency models: they are obtained by a combination of the above.

5.2.1 Consistency Models using Object Restriction

Some interesting **object relative consistency** models are given below:

1. **Pairwise Single Object Consistency** (PSO)

   Suppose $PV_{ij}^{x} = (NLV_i|_x \cup NLV_j|_x)$.

   Definition 28 : *If, for every process pair P_i and P_j and for each object x, PV_{ij}^{x} is acyclic then the execution satisfies* **Pairwise Single Object** *(PSO) consistency.*

   This model asserts that every pair of processes must agree on the ordering of conflicting events on every object that they share.

2. **Single Object Consistency** (SO)

   Suppose $PV^{x} = (\cup_i \; NLV_i|_x)$.

   Definition 29 : *If, for every object x, PV^{x} is a partial order, then the execution satisfies* **Single Object** *(SO) consistency.*

   This model asserts that all processes must agree on the ordering of all conflicting events on every object. This condition is often called coherence [29].

3. **Object Local Consistency** (OL)

   This consistency is derived from $GV_{(13,46)}^{p}|_x = (\cup_i \; NLV_i)|_x$. This augmented global view can be restricted to a single object x to produce the restricted global view $GV_{(13,46)}^{p}|_x$.

Definition 30 : *If for all objects x, $GV^p_{(13,46)}|_x$ is a partial order, then the execution satisfies* **Object Local** *(OL) consistency.*

This model asserts that all processes must agree on the ordering of all write events on every object x individually. However they must agree even in the presence of indirect ordering relations induced by other objects. This consistency model is provably equivalent to model $C^p_{(13,46)}$.

These models form the following hierarchy: PSO $<$ SO $<$ OL $\equiv C^p_{13,46}$. This means that any protocol that implements $C^p_{13,46}$ also implements the other models.

Figure 34 shows three examples. In Figure 34(a), the execution is not PSO consistent. In Figure 34(b), the execution is PSO consistent but not SO consistent. Finally, in Figure 34(c), the execution is PSO and SO consistent but not OL consistent.

P_1 :	$x!1$; $x!2$
P_2 :	$x?2$; $x?1$
P_3 :	$x?1$; $x?2$

(a) No object consistency.

P_1 :	$x!1$
P_2 :	$x?1$; $x!2$
P_3 :	$x?2$; $x?1$

(b) PSO consistent but not SO consistent.

P_1 :	$x!1$; $y!1$
P_2 :	$y?1$; $z!1$
P_3 :	$z?1$; $x?0$; $x?1$

(c) SO consistent but not $C^p_{(13,46)}$ consistent.

Figure 34: Examples of object relative consistency

5.2.2 Process Relative Consistency

The only **process relative consistency** model is **Single Process** (SP) consistency. Suppose $PV^j_{i,j} = (NLV_i|_{P_j} \cup NLV_j|_{P_j})$.

Definition 31 : *If, for every process pair P_i and P_j, $PV_{i,j}^j$ is a partial order, then the execution satisfies* **Single Process** *(SP) consistency.*

This model asserts that every process agrees with the ordering of all the write operations performed by another process. This model is equivalent to the **PRAM** model [44] and is useful in defining FIFO delivery.

Definition 32 : **Process FIFO delivery** *(PRAM [44])*

An execution satisfies process FIFO delivery iff for every writer P_j and every reader P_i of a shared object, PV_{ij}^j is a partial order.

Process FIFO delivery ensures that the order in which a writer changes any object will not be contradicted by a reader.

It is noteworthy that single process consistency is equivalent to requiring that the partial view obtained by the union of all restricted necessary local views ($\cup_i NLV_i|_{P_j}$) is acyclic. This is enforced by the fact that the local view of a process P_j, $NLV_j|_{P_j}$, contains all the events and their relative order.

The following hierarchy exists between this model and $C^p_{(13,46)}$: SP $<$ $C^p_{(13,46)}$.

Figure 35 illustrates SP. In Figure 35(a), the execution is not SP consistent. In Figure 35(b), the execution is SP consistent but not locally consistent.

P_1 :	$x!1$; $y!1$; $z!1$
P_2 :	$y?1$; $x?0$; $x?1$
P_3 :	$x?1$; $y?1$; $z?1$

(a) No process consistency.

P_1 :	$x!1$; $y!1$
P_2 :	$y?1$; $z!1$
P_3 :	$z?1$; $x?0$; $x?1$

(b) SP Consistent but not $C^p_{(13,46)}$ consistent.

Figure 35: Examples of process relative consistency

5.2.3 Other Consistency Models

It is possible to obtain a consistency model by applying restrictions on both object and process. This leads to the following mixed consistency models:

1. Suppose $PV_{ij}^{j,x} = (NLV_i|_{P_j,x} \cup NLV_j|_{P_j,x})$.

   Definition 33 : *If, for any pair of processes P_i and P_j and for every object x, $PV_{ij}^{j,x}$ is a partial order, then the execution satisfies* **Single Process/Object** *(SPO) consistency.*

   This model asserts that every process agrees with the order of operations by a writer process P_i on every object x. This is equivalent to object FIFO delivery and **Slow memory** [33]. It is also very similar to $C_{3\bar{8}}^{pp}$.

   Definition 34 : **Object FIFO delivery** *(Slow Memory [33])*
   An execution satisfies object FIFO delivery iff for every object x, writer P_j and reader P_i, $(NLV_i|_{P_j,x} \cup NLV_j|_{P_j,x})$ is a partial order.

   Object FIFO delivery ensures that the order in which a writer changes a variable x will not be contradicted by a reader in its reads. It is a weaker condition than process FIFO delivery. So, We have the hierarchy:

   SPO $<$ SO $<$ OL $\equiv C_{13,46}^{p}$ SPO $<$ SP $< C_{13,46}^{p}$

   It is noteworthy that process and object relative consistencies are not comparable. So they form two distinct hierarchies.

2. As we have done with some augmented views in Chapter 4, it is possible to combine some of these partial views to define stronger consistency models. As an example, a consistency model can be defined on both object and process consistency. This model would be very similar to **processor consistency** [6, 29].

Definition 35 : Processor Consistency

An execution respects the processor consistency constraints if it is single process (SP) consistent ($\forall\ P_i, P_j\ \ PV_{ij}^j$ *is acyclic) and single object (SO) consistent or coherent* (PV^x *is acyclic).*

5.3 Conclusion

In this chapter, we have introduced some weak consistency models based on restricted and partial views. Other models may be possible using other restrictions and augmentations but we have limited ourselves to those that seem more interesting.

Chapter 6

Exact Implementability of Weak Consistency Models

One of the reasons for our deriving the taxonomy of consistency hierarchies is to create a medium to analyze the consistency support required by various types of application programs, and that can also be used to determine as precisely as possible which model a particular protocol implements. Using these hierarchies as a basic tool one can determine the consistency models implemented by some protocols, and even derive new protocols. For some models, it is possible to come up with an exact implementation. An exact implementation involves a protocol that implements the required model but not any stronger model in the hierarchy.

In this chapter, we first define the concept of exact implementation. Then, we introduce many protocols and show they implement, or exactly implement, some models of our hierarchies. These protocols are grouped into four types: asynchronous protocols, semi-synchronous protocols, almost synchronous protocols and synchronous protocols. It is noteworthy that many protocols based on timestamps implement some form of causal memory $C_{u\overline{v}}$. Indeed, since these models are based on absence of certain bad orderings ($R_{\overline{v}}$), in the presence of some ordering R_i in an augmented GV, and GV intrinsically preserves causality, an obvious strategy to implement $C_{u\overline{v}}$ is to maintain causal knowledge among processes in all message exchanges in such a way as to avoid those ordering relations not allowed under $C_{u\overline{v}}$. This is an avoidance

strategy rather than the assurance strategy required in handling C_{ij}. As could be expected, avoidance can be done asynchronously, unlike assurance which often requires handshaking in the form of 2-phase or 3-phase protocols. Timestamps are generally useful for avoidance of contradicting causal knowledge in the computation, in contrast to global knowledge required in assurance requirements.

6.1 Exact Implementation

Since a stronger consistency model normally costs more (in space and/or time) to implement, we need to determine as precisely as possible the model implemented by a particular protocol. In this section, the notion of exact implementation is introduced, to capture the tightness of an implementation with respect to a consistency model.

Definition 36 : *A* shared memory protocol *implements a consistency model* C^p_{ij} *(*C_{ij}*) iff every execution allowed in the protocol has an acyclic* GV^p_{ij} *(*GV_{ij}*). A protocol implements* exactly C^p_{ij} *(*C_{ij}*) iff it implements* C^p_{ij} *(*C_{ij}*) but no* C^p_{ik} *(*C_{ik}*) such that* $C^p_{ik} > C^p_{ij}$ *(*$C_{ik} > C_{ij}$*).*

In the above definition, an exact implementation is restricted to a hierarchy generated from a common augmentation guard. Intuitively, it is possible to take the entire hierarchy formed by the eight guards in Table 2, integrated via the $' >'$ relation from Theorem 8, as illustrated in Figure 29. Moreover, as a direct consequence of the hierarchies, we can state the following corollary.

Corollary 2 : *A protocol that implements* C_{ij} *also implements all weaker models in the hierarchies.*

6.2 Asynchronous Protocols

We use the term *asynchronous protocol* to design a protocol that uses non-blocking read and write operations. Write operations atomically modify the local copy and

broadcast the new value to all readers. In Chapter 4, we have presented one such protocol, called AUP, that implements the minimal consistency model. In this section, we present many asynchronous protocols that use FIFO channels and/or timestamps to implement stronger consistency models. We analyze the implementation guarantees for each of these protocols.

6.2.1 Multiple Vector Timestamps Protocol (MVTP)

This protocol is very weak since it does not assume the presence of FIFO channels. To provide the illusion of FIFO channels, it uses a vector of version numbers `VN` associated with each object x. So, `x.VN[i]` represents the last version number that process P_i used to update object x. This protocol implements slow memory or Object FIFO delivery as defined earlier. Simply said, it provides a distinct FIFO channel for each object.

Process i		

$x!v$	:	`x.VN[i] := x.VN[i] + 1;`
		`x.v := v;`
		broadcast`(x,v,x.VN[i],i)`

$x?v$	:	**return**`(x.v);`
$receive(x, v, vn, j)$	:	**if** `x.VN[j] < vn` `x.VN[j] := vn; x.v := v;`

Protocol MTVP

Lemma 23 : *Protocol MVTP implements single process/object consistency (which is also named object FIFO delivery or* $C_{3\bar{8}}^{pp}$*).*

Proof: *Assume the contrary. This means we have* $x!v \xrightarrow{p} x!v'$ *and* $x?v' \xrightarrow{p} x?v$*. These in turn imply* $\mathtt{VN}(x!v) < \mathtt{VN}(x!v')$ *and* $\mathtt{VN}(x!v') < \mathtt{VN}(x!v)$ *respectively. An obvious contradiction is reached.* ■

Lemma 24 : *Protocol MTVP implements exactly* $C_{3\bar{8}}^{pp}$*.*

Proof: In order for MTVP to implement exactly $C^{pp}_{3\overline{8}}$, we must show that it does not implement $C^{-p}_{3\overline{8}}$, $C^{p}_{3\overline{8}}$, $C^{pp}_{3\overline{9}}$ and $C^{pp}_{6\overline{8}}$. The executions shown in Figure 36 are all permitted by MTVP but each violates one of these models. ■

P_1 : x!1 ; y!1
P_2 : y?1 ; x!2
P_3 : x?2 ; x?1

Execution that violates $C^{-p}_{3\overline{8}}$

P_1 : x!1 ; x!2
P_2 : x?2 ; y!1
P_3 : y?1 ; x?1

Execution that violates $C^{p}_{3\overline{8}}$

P_1 : x!1 ; x!2 ; y!1
P_2 : y?1 ; x?1

Execution that violates $C^{pp}_{3\overline{9}}$

P_1 : x!1
P_2 : x?1 ; x?2
P_3 : x?2 ; x?1

Execution that violates $C^{pp}_{6\overline{8}}$

Figure 36: Legal executions for protocol MTVP

6.2.2 Disjoint Version Number Protocol (DVNP)

This protocol also uses the concept of version number. However, it associates with each object a single version number which is updated whenever a process modifies the object.

Process i		

$x!v$	:	`x.VN := x.VN + 1;`
		`x.v := v;`
		broadcast`(x,v,x.VN,i)`

$x?v$	:	**return**`(x.v);`
$receive(x, v, vn, j)$	:	**if** `x.VN < vn` `x.VN := vn; x.v := v;`

Protocol DVNP

Lemma 25 : *Protocol DVNP implements a restricted form of local write consistency,* $C^{pp}_{3\overline{8}}$ *and* $C^{pp}_{6\overline{8}}$ $((x?v)/(x!v) \xrightarrow{p} (x!v') \Rightarrow (X?v') \not\xrightarrow{p} (X?v))$.

Proof: Assume the contrary. This means that we have $x!v/x?v \xrightarrow{p} x!v'$ and $x?v' \xrightarrow{p} x?v$. These in turn imply $\texttt{VN}(x!v) < \texttt{VN}(x!v')$ and $\texttt{VN}(x!v') < \texttt{VN}(x!v)$. An obvious contradiction is reached. ■

Lemma 26 : *Protocol DVNP implements exactly* $C^{pp}_{6\bar{8}}$.

Proof: In order for DVNP to implement exactly $C^{pp}_{6\bar{8}}$, we must show that it does not implement $C^{-p}_{6\bar{8}}$ and $C^{p}_{6\bar{8}}$. The executions shown in Figure 37 are all permitted by DVNP but each violates one of these models. ■

P_1 : x?1 ; x!2	P_1 : x?1 ; y!1
P_2 : x?2 ; y!1	P_2 : y?1 ; x!2
P_3 : y?1 ; x?1	P_3 : x?2 ; x?1
Execution that violates $C^{p}_{6\bar{8}}$	Execution that violates $C^{-p}_{6\bar{8}}$

Figure 37: Legal executions for protocol DVNP

6.2.3 Totally Ordered DVNP (TODVNP)

As in protocol DVNP introduced earlier, this protocol associates with each object a single version number. This version number is updated every time a process modifies the object. Moreover, when two operations have the same version number, process identifiers are used to order them. A new field x.pid is associated with each object x. It contains the identifier of the last process that modified object x.

```
                               Process i
x!v                 :  x.VN := x.VN + 1;
                       x.pid := i; x.v := v;
                       broadcast(x,v,x.VN,i)

x?v                 :  return(x.v);

receive(x,v,vn,j)   :  if [x.VN,x.pid] < [vn,j]
                              x.VN := vn; x.pid := j;
                              x.v := v;
```

Protocol TODVNP

Lemma 27 : *Protocol TODVNP implements some restricted form of local consistency* $C^{pp}_{u\bar{v}}$ *where u and v can take any valid value for these models* ($x?v/x!v \xrightarrow{p} x?v'/x!v' \Rightarrow X?v'/x!v' \not\xrightarrow{p} X?v/x!v$).

Proof: Suppose the contrary. This implies that there exists $x?v/x!v \xrightarrow{p} x?v'/x!v'$ as well as $x?v'/x!v' \xrightarrow{p} x?v/x!v$. However, according to the total ordering of the protocol, this means that $\mathtt{VN}(x!v) < \mathtt{VN}(x!v')$ as well as $\mathtt{VN}(x!v') < \mathtt{VN}(x!v)$. Hence a contradiction is reached. ■

Lemma 28 : *Protocol TODVNP implements exactly $C^{pp}_{u\overline{v}}$.*

Proof: In order for TODVNP to implement exactly $C^{pp}_{u\overline{v}}$, we must show that it does not implement $C^{-p}_{u\overline{v}}$ and $C^{p}_{u\overline{v}}$. The executions shown in Figure 38 are all permitted by TODVNP but each violates one of these models. ■

P_1 :	x?1 ; y!1	P_1 :	x?1 ; x!2
P_2 :	y?1 ; x!2	P_2 :	x?2 ; y!1
P_3 :	x?2 ; x?1	P_3 :	y?1 ; x?1

Execution that violates $C^{-p}_{3\overline{8}}$ Execution that violates $C^{p}_{3\overline{8}}$

Figure 38: Legal executions for protocol TODVNP

6.2.4 Extended Asynchronous Update Protocol (EAUP)

This protocol is an extended version of the AUP protocol, in which we assume that the underlying communication system provides FIFO channels. This assumption is sufficient to enforce single process consistency or $C^{pp}_{3\overline{8}}$.

Lemma 29 : *Protocol EAUP implements $C^{pp}_{3\overline{8}}$ ($x!v \xrightarrow{p} x!v' \Rightarrow X?v' \not\xrightarrow{p} X?v$).*

Proof: This immediately follows from the use of FIFO channel and hence the preservation of the arrival ordering of (x,v) and (x,v') at a reader site. Consequently, the asynchronous update prevents $X?v' \xrightarrow{p} X?v$. ■

Lemma 30 : *EAUP implements exactly $C^{pp}_{3\overline{8}}$.*

Proof: EAUP does not implement $C^{p}_{3\overline{8}}$ and neither $C^{-p}_{3\overline{8}}$. Figure 39 shows executions allowed by EAUP that violate $C^{p}_{3\overline{8}}$ and $C^{-p}_{3\overline{8}}$ respectively. ■

P_1:	x!1 ; x!2 ;	P_1:	x!1 ; y!1 ;
P_2:	x?2 ; y!1 ;	P_2:	y?1 ; x!2 ;
P_3:	y?1 ; x?1 ;	P_3:	x?2 ; x?1 ;

Violates $C^{p}_{3\overline{8}}$ Violates $C^{-p}_{3\overline{8}}$

Figure 39: Executions that violate $C^{p}_{3\overline{8}}$ and $C^{-p}_{3\overline{8}}$

6.2.5 Logical Time Protocols (LTP)

For this protocol, timestamps extracted from a logical clock are recorded at the local copy of an object in each reader process. Naturally a logical clock is maintained among the distributed processes. There are a few versions of such protocols, leading to different results.

Logical Read Time Protocol (LRTP)

The Logical Read Time Protocol (LRTP) is a novel protocol that allows a process to maintain the timestamp of the copy of a shared memory object locally. This timestamp indicates the (logical) time of the last read operation performed by the process on that object. An update received from another writer will be accepted only if it carries a timestamp larger than the last "time" that process read that object. Logical time is maintained among all processes through message passing. The full protocol is given below.

```
                              Process i
x!v                : LC := LC + 1;                increment logical clock
                     x.v := v;                    update value of x
                     broadcast(x,v,LC);           send update to readers

x?v                : LC := LC + 1;                increment logical clock
                     x.VN := LC;                  update timestamp of x
                     return x.v;

receive(x,v',LC')  : if x.VN < LC'                if larger timestamp
                             x.v := v';           accept the update and
                             LC := max(LC, LC');  update logical clock
```

Protocol LRTP

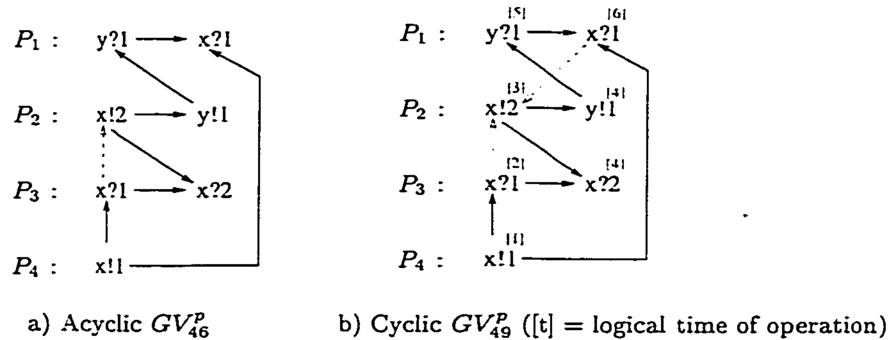
It is provable that LRTP implements C^p_{46}. The proof is based on the total ordering of logical time.

Lemma 31 : *LRTP implements C^p_{46}.*

Proof: Suppose VN[e] represents the logical time of a memory operation in GV^p_{46}. Given R^p_4 ($x?v \xrightarrow{p} x?v'$) in GV, LRTP ensures that VN[$x?v$] < VN[$x!v'$]; otherwise, the update of x from v to v' cannot happen and hence v' cannot be read in that process. Hence the augmentation using R_6 ($x?v \rightarrow x!v'$) satisfies also the logical time ordering, i.e., for every instance of $op_1 \rightarrow op_2$ in GV^p_{46}, we must also have VN[op_1] < VN[op_2]. From the total ordering of logical time, GV^p_{46} cannot contain any cycle. ■

Lemma 32 : *LRTP implements C^p_{46} exactly.*

Proof: From the definition of exact implementation and Table 1, we need to prove LRTP does not implement C^p_{49}. Figure 40 shows an example execution allowed by LRTP but acyclic GV^p_{46} and cyclic GV^p_{49}. Hence the claim. ■



a) Acyclic GV^p_{46} b) Cyclic GV^p_{49} ([t] = logical time of operation)

Figure 40: Acyclic GV^p_{46} and cyclic GV^p_{49}

Logical Write Time Protocol (LWTP)

In a similar manner, the timestamp maintained for an object may indicate the last time that process wrote into the object. When an update is received, it is accepted only if the timestamp of the update is larger than the last time the process wrote into the same object. The full protocol is given below:

Process i			
$x!v$	:	`LC := LC + 1;`	`increment logical clock`
		`x.v := v;`	`update local copy`
		`x.VN := LC;`	`record last-write time`
		`broadcast(x,v,LC);`	`send update to readers`
$x?v$	:	`LC := LC + 1;`	`increment logical clock`
		`return x.v;`	
$receive(x, v', LC')$	:	`if x.VN < LC'`	`if larger timestamp`
		`x.v := v';`	`accept the update and`
		`LC := max(LC, LC');`	`update logical clock`

Protocol LWTP

Lemma 33 : *LWTP implements* C^p_{13}.

Proof: The proof is exactly analogous to that of Lemma 31. In particular, the update protocol ensures that if $x!v \xrightarrow{p} x?v'$, then it must be that $\mathtt{VN}[x!v] < \mathtt{VN}[x!v']$. Hence any ordering in GV^p_{13} is also ordered in logical time. ■

Lemma 34 : *LWTP implements* C^p_{13} *exactly.*

Proof: As in the case of LRTP, LWTP does not implement C^p_{19} and hence LWTP implements C^p_{13} exactly. Figure 41 shows an example execution allowed by LWTP that has cyclic GV^p_{19}. ■

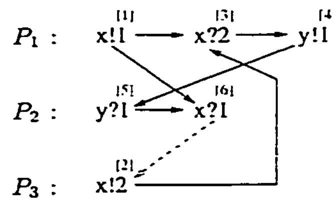


Figure 41: Cyclic GV^p_{19} allowed by LWTP

Both logical time protocols presented are basically asynchronous; every read or write operation is performed without explicit synchronization among the processes. Hence it can be expected that memory latency is small, as all operations involve local events and are wait-free. The consistency models they exactly implement are also

rather weak; both C_{13}^p and C_{46}^p are one of the weakest members of their respective consistency hierarchies.

Logical Time Protocol (LTP) and Local Read Consistency (LRC)

The previous protocols can be combined together to form a protocol that implements C_{13}^p and C_{46}^p. The composite updates the time of the last local operation by the process on an object.

Process i			

$x!v$	:	`LC := LC + 1;`
		`x.v := v;`
		`x.VN := LC;`
		`broadcast(x,v,LC);`
			`increment logical clock`
			`update local copy`
			`record last-write time`
			`send update to readers`

$x?v$	:	`LC := LC + 1;`
		`x.VN := LC;`
		`return x.v;`
			`increment logical clock`
			`record last-read time`

$receive(x, v', LC')$	:	`if x.VN < LC'`
		`    x.v := v';`
		`    LC := max(LC, LC');`
			`if larger timestamp`
			`accept the update and`
			`update logical clock`

Protocol LTP

Lemma 35 : *LTP implements exactly C_{13}^p and C_{46}^p.*

Proof: With `x.VN` updated to `LC` in both read and write operations in a process, the result follows from Lemmas 32 and 34. ■

C_{13}^p and C_{46}^p forms a LRC model as presented in Chapter 4; the augmentation rule applied become: $x!v/x?v \xrightarrow{p} x?v' \Rightarrow x!v/x?v \rightarrow x!v'$.

Intuitively, this asserts that when a process reads a local value different from the last value associated with its last operation on the object, then the latter must be "ordered" before the write of the new value (performed elsewhere).

6.2.6 Logical Clock Protocol (LCP)

This protocol uses a logical clock `LC` to associate a timestamp with each write operation. With this protocol there is no timestamp associated with an object. An

operation is accepted only if its timestamp is bigger than the current clock. The problem with this protocol is that many write operations may be discarded even if they do not cause any inconsistencies.

Process i		
$x!v$	:	LC := LC + 1; x.v := v; **broadcast**(x,v,LC)
$x?v$	:	return(x.v);
$receive(x, v, T)$	:	if LC < T x.v := v; LC := max(LC,T);

Protocol LCP

Lemma 36 : *This protocol implements* $C_{3\bar{9}}$ *and* $C_{6\bar{9}}$ $(x?v/x!v \leadsto x!v' \Rightarrow X?v' \not\leadsto x!v)$.

Proof: According to the protocol, the relation $x?v/x!v \leadsto x!v'$ implies that LC(x!v) < LC(x!v'). Again, according to the protocol, the relation $X?v' \leadsto x!v$ implies LC(x!v') < LC(x!v). Hence a contradiction. ■

6.2.7 Totally Ordered LCP (TOLCP)

This protocol is an extension of protocol LCP presented earlier. It uses a logical clock LC to associate a timestamp with each operation, which is then used as a version number $x.VN$. Moreover, as in the preceding protocol, process identifiers are used to resolve conflicting version numbers.

Process i		
$x!v$	:	LC := LC + 1; x.VN := LC; x.pid := i; x.v := v; **broadcast**(x,v,LC,i)
$x?v$	:	return(x.v);
$receive(x, v, T, j)$	:	if [LC,x.pid] < [T,j] x.VN := T; x.pid := j; x.v := v; LC := max(LC,T);

Protocol TOLCP

Lemma 37 : *Protocol TOLCP implements* C_{69} $(x?v \rightsquigarrow x!v' \Rightarrow X?v \rightarrow x!v')$.

Proof: According to the protocol, $x?v \rightsquigarrow x!v'$ implies that `LC(x!v) < LC(x!v')`. Again, according to the protocol, $x!v' \rightsquigarrow x?v$ implies `LC(x!v') < LC(x!v)`. Hence a contradiction. Moreover, all concurrent events are ordered by the process identifier. So a total order exists among them. ■

6.2.8 Causal Update Protocol (CUP)

This protocol associates with each object x a version number `x.VN`. The version number of object x is incremented each time it is updated. When process P_i executes a write operation, it broadcasts the entire object space `OS` to all processes. This guarantees that the information will always be causally up-to-date. CUP maintains the most recent version of object x causally among the processes. Updates will be performed only if the received version is more recent than the version known at the time.

```
                        Process i
x!v           :  OS[x].VN := OS[x].VN + 1;
                 OS[x].v := v;
                 broadcast(OS)

x?v           :  return(x.v);

receive(ROS)  :  for all objects x
                         if OS[x].VN < ROS[x].VN
                                 OS[x].v := ROS[x].v;
                                 OS[x].VN := ROS[x].VN;
```

Protocol CUP

Lemma 38 : *Protocol CUP implements* $C_{u\bar{v}} =$

$$(x?v)_i/(x!v)_i \rightsquigarrow (x?v')_j/(x!v')_j \Rightarrow (x?v')_k/(x!v')_k \not\rightarrow (x?v)_l/(x!v)_l.$$

Proof: Using the causal broadcast of OS and the update policy at a reader process, the latter is guaranteed to maintain the most recent version of every object it has perceived through updates it has received. If $x?v/x!v \rightsquigarrow x?v'/x!v'$ exists in GV, then it must be also true that $OS[x].VN < OS[x].VN'$. This in turn implies GV cannot have $x?v'/x!v' \rightarrow x?v/x!v$. ■

6.2.9 Vector Time Protocol (VTP)

LTP can be extended to involve a vector to record the most recent write operation of each object known to a process. With n objects in the system, the vector has n elements. As update messages are broadcast from a writer to all readers, naturally every process becomes aware of the most recent version `ts[j]` and value `v[j]` of object `j`. Recentness of an object `j` is established by using a tuple: the version number `ts[j].n` and the process identifier `ts[j].pid`. The process identifier is used to totally order the tuples. In the VTP protocol presented below, it is assumed that the objects are identified with `j = (1,...,n)`, the timestamps with `T = (ts[1], ts[2], ...,ts[n])` and `T' = (ts[1]', ...,ts[n]')`, `V = (v[1], ..., v[n])`, the value with `V' = (v[1]',...,v[n]')`, `ts[i] = (ts[i].n, ts[i].pid)` and that `max(T,T')` returns the component-wise larger timestamps of `T` and `T'`.

```
                                  Process i
j!v            :  LC := LC + 1;                       increment clock
                  ts[j] := (LC, i);                   update vector clock
                  v[j] := v;                          update local copy
                  broadcast (T,V);                    broadcast to readers

j?v            :  LC := LC + 1;                       increment clock
                  ts[j] := LC;                        record last-read time
                  return v[j];

receive(T',V') :  for all j do                        for all objects
                     if ts[j] < ts[j]'                if larger timestamp
                        v[j] = v[j]' ;                update the value,
                        T:=(max(T,T');                the vector clock and
                        LC:=max(LC,ts[1].n,...,ts[n].n)    logical clock;
```

Protocol VTP

In VTP, a process accepts an update only if its version is more recent than the present one. Hence the logical times (augmented with `pid`) of the two writes are also ordered in the same manner.

Lemma 39 : *Protocol VTP implements exactly* C_{13}.

Proof: We use `ts[e]` to denote the logical time `(ts[e].n,ts[e].pid)` of each operation. From VTP, we can easily establish:

1. if $op_1 \xrightarrow{p} op_2$ then $\mathtt{ts}[op_1] < \mathtt{ts}[op_2]$,

2. if $x!v \rightarrow x?v'$ then $\mathtt{ts}[x!v] < \mathtt{ts}[x?v']$, and $\mathtt{ts}[x!v] < \mathtt{ts}[x!v']$ (else the reader could not read v')

Hence the augmented view GV_{13} cannot be cyclic without violating the total ordering in (ts[e].n,ts[e].pid). Moreover, VTP does not implement C_{19}. Figure 42 is an example execution with cyclic C_{19} that is allowed under VTP. Hence the claim. ■

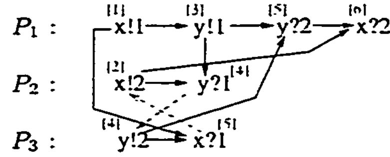


Figure 42: Cyclic GV_{19} allowed by VTP

Protocol VTP is interesting because it also implements $C_{1\bar{9}}$. Since $C_{1\bar{9}}$ is the strongest model in the weak consistency hierarchy, it follows, from Theorem 9, that VTP implements every model in this particular hierarchy.

Lemma 40 : *Protocol VTP implements $C_{1\bar{9}}$.*

Proof: Following the proof of Lemma 39, if $x!v \rightarrow x?v'$ holds in GV, then $\mathtt{ts}[x!v] < \mathtt{ts}[x!v']$. But if there exists an instance of $x?v$ such that $x!v' \rightarrow x?v$ in GV, then we also have $\mathtt{ts}[x!v'] < \mathtt{ts}[x!v]$. Hence we have a contradiction. ■

VTP can be slightly changed to become VRWTP (Vector Read/Write Time Protocol), which allows the timestamp of an object to change at every read operation (in addition to every write operation). Specifically, the "Read" function in VTP is modified as follows:

Process i			
$x?v$	:	LC := LC + 1;	increment logical clock
		ts[j] := LC;	record last-read time
		return v[j];	

With the above change, it is observable that if $x?v \to x?v'$ then $\mathtt{ts}[x?v] < \mathtt{ts}[x!v']$, else that reader will not be able to read v'. Hence we can prove that VRWTP implements C_{46}. For brevity, the details are omitted here.

VTP is expensive in that the message passing actually carries object values. The latter can be eliminated if a process delays a read operation until it has received the value corresponding to the most recent version causally known to the process. In other words, if other indirect channels form a faster path than the update channel to a reader, then the reader has to wait until the actual update is received.

6.2.10 Ahamad Vector Clock Protocol (AVCP)

Ahamad [9] has proposed the following protocol for causal memory.

Process i			
$x!v$	:	VC[i]:=VC[i] + 1;	increment clock
		x.v:=v;	update local copy
		broadcast(i,x,v,VC);	send to readers
$x?v$	:	return x.v;	
$receive(j, x, v', VC')$	:	wait until	
		VC[k] ≥ VC'[k] ∀ k≠j	if larger timestamp
		and VC'[j]=VC[j]+1;	
		x.v := v;	accept update and
		VC[j] := VC'[j]+1;	update logical clock

Protocol AVCP

It is assumed that the *receive* is multi-threaded. When one is blocked, other messages can be received. Moreover, when the *receive* is blocked, reads and writes can continue to be issued.

Lemma 41 : *Protocol AVCP implements* $C_{3\overline{9}}$ *and* $C_{6\overline{9}}$ $(x!v/x?v \to x!v' \Rightarrow x!v' \not\to X?v)$.

Proof: This was proved in [9] ■

This protocol does not implement $C_{4\overline{6}}$ $(x?v \to x?v' \Rightarrow x!v'/x?v' \not\to X?v)$. Figure 43 shows an execution which is permitted by the protocol AVCP but contradicts $C_{4\overline{6}}$.

P_1 : x!1 ;
P_2 : x!2 ;
P_2 : x?1 ; y!1
P_3 : y?1 ; x?2 ; z!1
P_4 : z?1 ; x?1

Figure 43: Legal execution for protocol AVCP that violates C_{46}

6.3 Semi-synchronous Protocols

Most of the asynchronous protocols implement very weak models and those that implement stronger model are very expensive in terms of space and bandwidth. To reduce this cost, we use blocking functions. In this new class of protocols, called semi-synchronous protocols, some but not all operations involve some blocking. In many cases, timestamps are used to determine if an operation or the process must be blocked.

6.3.1 2-phase Write Protocol (2WP)

Protocol 2WP uses a blocking 2-phase write protocol to block a writer until the write is acknowledged by every reader. Even if not written, we consider that concurrent writes are resolved by the use of process identifiers. This protocol does not assume FIFO channels as many of the timestamps or version number based protocols do.

Process i		
$x!v$	:	broadcast(x,v,i); wait ack(x,v) from all process;
$x?v$	:	return x.v;
$receive(x,v,j)$	:	x.v :=v; send(ack,x,v) to process P_j;

Protocol 2WP

Lemma 42 : *This protocol implements* C_{38}^{pp} $((x!v) \xrightarrow{p} (x!v') \Rightarrow (X?v') \not\xrightarrow{p} (X?v))$.

Proof: This condition is guaranteed since a previous write must end before the next one can start. So, when the new value is received by a process, the last value has already been received and is overwritten. ■

6.3.2 Invalidation Protocol

An invalidation protocol involves invalidation messages between readers and writers, so that an old value will not be read again. The following subsection presents two such invalidation protocols: a reader invalidation protocol (RIP) and a writer invalidation protocol (WIP).

Reader Invalidation Protocol (RIP)

The Reader Invalidation Protocol (RIP) requires a writer to wait until the value it last wrote has become invalid among all readers before the writer is allowed to read a new value. Hence synchronization latency occurs only when a writer reads a new value. In this protocol, the version number `VN` and the process identifier `pid` are used to resolve concurrent writes. The process identifier `pid` is also used to determine if a value to be read by process P_i is different from the last value it has written. In such a case, P_i must wait until the last value it has written is invalid among all potential readers. The variable `x.invalid` is used to maintain this information.

```
                                  Process i
x!v                      :  (x.VN,x.pid) := (x.VN + 1, i);
                            x.v := v;
                            broadcast (x, v, x.VN, i);

x?v                      :  if x.pid <> i
                            wait until x.invalid[j] for all reader Pj;
                            return x.v;

receive(x,v',T,j)        :  if (x.VN,x.pid) < (T,j)
                                  send(inv,x, x.VN, i) to Process x.pid;
                                  (x.VN,x.pid) := (T,j);
                                  x.v := v'
                            else send(inv,x,T,i) to Pj;

receive(inv,x,x.VN,j)    :  x.invalid[j] := true;
```

Protocol RIP

In RIP, a writer waits until the previous value it wrote has become invalid. Invalidation messages are sent from a reader to the writer whenever the former has received

a new version, or equivalently, the value written by the latter will not be readable by the former. Intuitively, this means C^p_{17} is implemented, as proved next.

Lemma 43 : *RIP implements C^p_{17} ($x!v \xrightarrow{p} x?v' \Rightarrow X?v \rightarrow x?v'$).*

Proof: Suppose $x!v \xrightarrow{p} x?v'$ (R^p_1) is used to augment GV with $X?v \rightarrow x?v'$ (R_7). From the description of RIP, a reader will send an invalidation message concerning $x = v$ to the writer only if it has received a more recent version of x, and hence v will never be readable again. In addition, the writer of v can read a new value v' only if it has received invalidation messages from all readers. Hence every $end_(x?v) \mapsto end_(x?v')$, or the augmentation preserves $\mapsto$ ordering among the related events. Using the same reasoning as in the proof of Lemma 50, we can similarly assert that GV^p_{47} is acyclic. ■

Writer Invalidation Protocol (WIP)

Protocol WIP associates to each object two version numbers: the version number of the current value, `x.VN`, and the version number of the last value read or written, `x.last`. `x.last` is transmitted with any new value and is used to invalidate old values. A boolean indicator is associated with each version number (`valid(x.VN)`) to indicate if the value associated with this version number was invalidated by some write operations.

Process i		

$x!v$	:	`x.last := x.VN ; x.VN := x.VN + 1;`
		`x.v := v;`
		`broadcast (x, v, x.VN, x.last);`

$x?v$	:	`wait until valid(x.VN);` `x.last := x.VN; return x.v;`

$receive(x, v, vn, last)$	:	`for all x.last ≤ last`
		`valid(x.last) := false;`
		`if vn > x.last`
		`x.v := v; x.VN := vn;`
		`valid(x.VN) := true;`

Protocol WIP

Lemma 44 : *Protocol WIP implements C^{pp}_{68} ($x?v/x!v \xrightarrow{p} x!v' \Rightarrow X?v' \not\xrightarrow{p} X?v$).*

Proof: This is obvious since any value read or written before the current value will be invalidated before the new value can be read. ■

6.3.3 Causal Read Protocol (CRP)

This protocol associates a version number with each object. All the version numbers are grouped into an Object Clock (OC). When an object x is updated, its corresponding version number in the object clock OC[x] is modified. OC is then broadcast with each write operation. Even if it is not specified here, process identifiers are used to resolve conflicting version numbers.

Process i		

$x!v$	:	OC[x] := OC[x] + 1;
		x.VN = OC[x];
		x.v := v;
		broadcast(x,v,OC)

$x?v$	:	**while** x.VN < OC[x] **do nothing**; **return**(x.v);
$receive(x, v, T)$	:	**if** x.VN < T[x] x.v := v; x.VN := T[x]; OC := max(OC,T);

Protocol CRP

Lemma 45 : *Protocol CRP implements* $C_{u\overline{v}} =$
$(x?v/x!v \rightsquigarrow x?v'/x!v' \Rightarrow x?v'/x!v' \not\twoheadrightarrow x?v/x!v)$.

Proof: The proof follows that of Lemma 38 and the fact that a reader is delayed until it has received the most recent version of an object through the causal broadcasts that it has received. ■

6.3.4 Asynchronous 3-phase Protocol (A3P)

Protocol A3P uses a 3-phase write protocol that does not block the writer. However, a reader may be blocked until the write has committed.

	Process i
$Init(x, v)$	: `x.old = -; ∀ x:  readable(x) := true;`
$x!v$	: `readable(x) = false;` `x.v = v; x.old = x.v` **broadcast**`(x,v);`

$x?v$	: **if** `((x.old ≠ -)` **and** `(x.old ≠ x.v))`
	        wait `readable(x);`
	`x.old = x.v;`
	`return(x.v);`

$receive(x, v)$	: `readable(x) := false;`
	`x.v := v;`
	`send(ack,x,v);`

$receive(ack, x, v)$	: **if** `ack received from all processes`
	        `broadcast(readable,x,v);`
	        `readable(x) := true;`

$receive(readable, x, v)$	: `readable(x) := true;`

Protocol A3P

Lemma 46 : *This protocol implements* $C_{4\bar{9}}$ $(x?v \rightarrow x?v' \Rightarrow X?v' \not\rightsquigarrow x!v)$.

Proof: The protocol ensures that a value is read only after the write has ended (so a write happens before a read in real time). If $x?v \rightarrow x?v'$, the protocol ensures that $x!v$ ends (x becomes readable) before $x!v'$. if $X?v' \rightsquigarrow x!v$, this means that $x!v'$ ends (x becomes readable) before $x!v$. Hence a contradiction. ■

6.3.5 Direct/Indirect Vector Clock Protocol (DIVCP)

This protocol associates with each process two vector clocks. The first vector clock, called `DVC`, contains the version numbers of each operation directly received from each process. The second vector clock, `IVC` contains the version number of the last operation of each process indirectly received from any other process. So, before reading any value, a process waits until `IVC` $\leq$ `DVC`.

Process i		

$x!v$	:	`DVC[i] := DVC[i] + 1;`
		`x.v := v; x.VN = DVC[i];`
		broadcast`(x,v,DVC[i],max(DVC,IVC),i)`

$x?v$	:	**while** `DVC` $\leq$ `IVC` **do nothing**; **return**`(x.v);`

$receive(x, v, T, VC, j)$	:	`DVC[j] := T; DVC[i] := max(DVC[i],T);`
		`IVC = max(IVC,VC);`
		if `(x.VN < T)`
		`x.VN := T; x.v := v;`

Protocol DIVCP

Lemma 47 : *Protocol DIVCP implements the same model as CRP ($C_{u\bar{v}}$).*

Proof: The proof is based on the fact that all causal events are captured by DVC and IVC and the fact that a process waits until the indirectly known events are received. ■

6.4 Almost Synchronous Protocols

The preceding protocols are semi-synchronous because at least one of the operations is always asynchronous. In this section we present a category of protocols in which all the operations are synchronous except when some conditions are satisfied.

6.4.1 Possibly Asynchronous Read (PAR)

Protocol PAR uses a blocking 3-phase write protocol that blocks the writer until the write is committed. The reader is also blocked unless the preceding committed value was never read. The field `x.old` is used to stored the value returned in the last read operation. The field `x.comm` contains the last value committed. If they are different, the read may execute asynchronously. As before, even if it is not specified, concurrent writes are resolved by the use of process identifiers.

```
                              Process i
Init                   :  x.old = x.comm = -;

x!v                    :  broadcast(x,v)
                          wait for all ack_j(x,v)
                          broadcast(commit,x,v)
                          x.v := v

x?v                    :  if (x.old = x.comm) and (x.old ≠ x.v)
                                wait committed(x)
                          x.old := x.v;
                          return x.v

receive(x,v)           :  x.v = v
                          committed(x) := false
                          send(ack,x,v)

receive(commit,x,v)    :  if (x.v = v) committed(x) := true
                                x.comm=x.v
```

Protocol PAR

Lemma 48 : *Protocol PAR implements* C_{49}^{p} $(x?v \xrightarrow{p} x?v' \Rightarrow x?v \rightarrow x!v')$.

Proof: When a value is read it may be either committed or not. If it is committed then since the write is blocking we have $x?v \rightarrow x!v'$ for any value v written before the value v'. If the value is not committed and we have $x?v \xrightarrow{p} x?v'$, this means there is another value v'', written more recently than v, which is committed. Then the value v cannot be read anymore. Hence $x?v \rightarrow x!v'$. ■

6.4.2 Possibly Asynchronous Write Protocol (PAWP)

Protocol PAWP mixes the use of the blocking 3-phase write protocol, that blocks the writer until the write is committed, and of an asynchronous write. The asynchronous write is used only if the last value read or written by this process is different from the last value committed. The reader is also blocked unless the asynchronous write is used. The field `x.old` is used to store the last value read or written. As before, even if it is not specified, concurrent writes are resolved by the use of process identifiers.

Process i	
Init	: `x.old = - ; readable(x) = true`

$x!v$	: `if ((x.old = -) or ((x.old <> x.v) and readable(x)))`
	`broadcast(write,x,v,async);`
	`else`
	`broadcast(write,x,v,sync);`
	`wait for all ack(x,v);`
	`broadcast(readable,x);`
	`x.old = x.v;      (to enforce x!v -> x!v');`
	`x.v = v;`

$x?v$	: `wait readable(x);`
	`x.old = x.v (for x?v->x!v');`
	`return x.v;`

$receive(x, v, type)$	: `x.v = v;`
	`if (type=sync)`
	`readable(x) = false;`
	`send(ack,x,v);`

$receive(readable, x)$	: `readable(x) = true;`

Protocol PAWP

Lemma 49 : *Protocol PAWP implements* $C^p_{\overline{39}}$ *and* $C^p_{\overline{69}}$, $x?v/x!v \xrightarrow{p} x!v' \Rightarrow X?v' \not\rightarrow x!v$.

Proof: When $x?v/x!v \xrightarrow{p} x!v'$, we know that $x!v$ ends before $x!v'$ starts in real time. If we have $X?v' \rightarrow x!v$, this implies that $x!v'$ starts in real time before $x!v$ ends. Hence a contradiction. ■

6.4.3 Fast-Read Three-Phase Protocol (FRTPP)

A traditional synchronous protocol involves a three-phase handshaking between a writer and the readers of a shared object: the writer broadcasts the new value to the readers, and waits for acknowledgements from the latter before broadcasting a commit

message to the latter, thereby allowing the value to be read. Concurrent writes are serialized by using version number and pid. Such a protocol is expensive: a memory operation may take extra delay before a new value is committed and readable. A reader must read in full synchrony with the rest of the processes.

A refined version of the 3-phase write-protocol is presented in this section. It is called Fast-Read Three-Phase Protocol (FRTPP). As the name implies, the protocol allows a reader to read asynchronously, when some locally decidable property holds.

```
                                   Process i

x!v                 :  x.t := x.t + 1; x.v := v; x.pid := i;
                       broadcast(x,v,x.t,x.pid);
                       suspend process i until
                           received ack(x,v,x.t,x.pid) from every reader;
                       broadcast(cmit,x,x.t,i);
                       x.commit := x.last := (x.t, x.pid);

x?v                 :  wait for x.ready or x.last <> x.commit;
                       x.last := x.commit;
                       return x.v;

receive(x,v',t',j)  :  if (t',j) > (x.t, x.pid)
                             (x.t, x.pid) := (t',j);
                             x.ready := false;
                       reply with ack(x,v',t',j);

receive(cmit,x,t',j) : if (x.t, x.pid) = (t',j)
                            x.commit := (t',j);
                            x.ready := true;
```

Protocol FRTPP

The above FRTPP differs from the traditional 3-phase protocol in only the read function: a reader is allowed to read without waiting for a value to be committed if `x.last <> x.commit`. Under such a condition, reader delay can be reduced.

FRTPP does not implement sequential consistency but implements both C^p_{49} and C^p_{19}, as proved next.

Lemma 50 : *FRTPP implements* C_{19}^p *and* C_{49}^p*.*

Proof: We will assume a memory operation begins/ends at the first/last statement in the respective functions of the given protocol. Hence $end_(x?v)$ occurs when the value v is returned, and similarly for $end_(x!v)$. The proof is constructed by showing that GV_{19} and GV_{49} cannot be cyclic. Specifically,

1. $op_1 \xrightarrow{p} op_2$: Obvious, $end_(op_1)$ happens before (denoted by $\mapsto$) $start_(op_2) \mapsto end_(op_2)$.

2. $x!v \rightarrow x?v$: Again, we get $start_(x!v) \mapsto end_(x?v)$.

3. $x?v \rightarrow x!v'$ (due to augmentation with $X?v \rightarrow x!v'$): Since some reader of $x!v'$ has x.commit (involving some $v*$) <> x.last (involving v), it follows also that all $end_(x?v) \mapsto end_(x!v*) \mapsto end_(x!v')/end_(x?v')$.

From the above, a cyclic GV_{19}^p or GV_{49}^p would create a cycle of events whose "ends" are ordered cyclically by the happens-before relation. This is a contradiction. ■

FRTPP can be extended so that a process causally keeps track of the last version of an object read globally by other processes (through the underlying message passing). This increases the complexity by a constant factor but the extended protocol will implement C_{19} and C_{49}, i.e., without restriction to program-ordered augmentation. The details are omitted here.

6.4.4 Ahamad's Owner Protocol for Causal Memory (OP)

Ahamad [8] and John [36] present the following owner based protocol that implements causal memory.

Process i		

$x!v$	:	LC := LC + 1;
		if owner(x) ≠ i
		send(write,x,v,LC,i) to owner(x);
		receive(reply,x,v,LC') from owner(x);
		LC = MAX(LC,LC');
		x.v := v; x.VN = LC;

$x?v$	:	if x= NIL;
		send(read,x,i) to owner(x);
		receive(reply,x,v',LC') from owner(x);
		LC = MAX(LC,LC');
		x.v := v; x.VN = LC;
		∀ y : y.VN < LC ⇒ y = NIL;
		v := x.v;

Owner		
receive(read, x, j)	:	send(reply,x,x.v,x.VN) to j;

receive(write, x, v, LC', j)	:	LC := MAX(LC,LC');
		x.v := v; x.VN := LC;
		∀ y : y.VN < LC ⇒ y = NIL;
		send(reply, x,x.v, x.VN) to j;

Ahamad's owner protocol

This protocol implements a stronger version of causal memory than the one they define. The exact causal memory implemented by this protocol can be defined by the property rule $x?v/x!v \rightarrow x?v'/x!v' \Rightarrow x!v' \not\rightarrow X?v$.

Lemma 51 : *OP implements* $C_{u\overline{9}}$ $(x!v/x?v \rightarrow x!v'/x?v' \Rightarrow x!v' \not\rightarrow X?v)$.

Proof: The Owner protocol uses vector clocks and invalidation to implement causality between write and read operations. Moreover, the owner concept ensures that all writes to a single object are totally ordered. So assume that $x!v/x?v \rightarrow x!v'/x?v'$. In such a case, causality and total ordering of operations ensure that $x.VN < x.VN'$. Suppose there exists a $x?v$ such that $x!v' \rightarrow x?v$. In such a case, causality and total ordering of write operations ensure that $x.VN > x.VN'$. This is a contradiction. ■

6.5 Synchronous Protocols

6.5.1 Possibly Asynchronous Protocol (PAP)

Protocol PAP uses a blocking write operation with a possibly non-blocking read. The read operation blocks if the last value read or written is different from the current value. This is implemented by the fields `x.last`, which contains the version number of the last value read or written locally, and `x.commit`, which contains the version number of the last value committed (write is completed). As in the previous protocol, variable `x.VN` contains the version number of the last value written, while variable `x.WVN` contains the version number of the last value written by P_i.

```
                                  Process i
x!v                  :  wait until x.commit > x.last or x.VN = x.last;
                        x.VN := x.WVN:= x.VN + 1; x.pid := i; x.v := v;
                        broadcast (x, v, x.VN, x.pid);
                        suspend process i until
                            received(ack,x.WVN,i,j) from every reader Pj
                                 or (x.VN,x.pid) > (x.WVN,i);;
                        broadcast(commit,x,x.WVN,i);
                        x.last:=x.commit:=x.VN;

x?v                  :  wait until x.commit > x.last or x.VN = x.last ;
                        return x.v; x.last := x.VN;

receive(x,v,vn,j)    :  if (x.VN,i) < (vn, j)
                             x.v := v;
                             x.pid := j; x.VN := vn;
                             reply(ack,x.VN, j, i);

receive(commit,x,vn,j):  if (x.VN,x.pid) = (vn, j)
                                 x.commit := vn;
```

Protocol PAP

Lemma 52 : *Protocol PAP implements* C^p_{u3} *and* C^p_{u4} *(which can also be defined with relations* $x?v/x!v \xrightarrow{p} x?v'/x!v' \Rightarrow X?v \to x?v'/x!v'$*).*

Proof: When a new value is read or written, the last value read or written is either committed or there is a more recent value v'' which is committed. If it is committed

then since the write is blocking, we have $x?v \rightarrow x!v'$ for any value v written before the value v'. If the value is not committed and we have $x?v/x!v \xrightarrow{p} x?v'/x!v'$, this means there is another value v'' written more recently than v which is committed. Then the value v cannot be read anymore. Hence $X?v \rightarrow x?v'/x!v'$. ∎

6.6 Conclusion

In this chapter, we have introduced the notion of exact implementation. Using this notion, we have studied implementations for some of the models introduced in Chapters 4 and 5. It is noteworthy that from the understanding of the consistency requirements, it is relatively easy to derive suitable protocols that implement them. Indeed, this chapter presents many protocols, but it is easy to derive even more protocols. For example, most of the protocols for models involving global augmentation rules require the use of a vector clock, rather than a logical clock, which keeps track of the most recent version of every memory object known to a process. So, it is possible to develop vectorized versions of many protocols such as LWTP and LRTP. Other examples are the following protocols, which were developed but are not described in detail here:

SVNP: This protocol uses a logical clock to associate a timestamp to each asynchronous operation.

TOSVNP: This protocol is the same as SVNP in which process identifiers are used to force a total order on all asynchronous operations.

CDP: This protocol uses a vector clock to order asynchronous operations.

VTP2: This protocol uses vector timestamps and waiting to order operations.

C2WP: This protocol combines timestamps and 2-phase write operations to provide a better ordering of the operations.

Table 8 gives a summary of the protocols presented in this chapter.

Protocol	Implements or implements exactly
MVTP, VTP2, 2WP and EAUP	$C^{pp}_{3\bar{8}}$
DVNP and C2WP	$C^{pp}_{3\bar{8}}$ and $C^{pp}_{6\bar{8}}$
SVNP	$C^{-p}_{3\bar{8}}$ and $C^{-p}_{6\bar{8}}$
LCP	$C_{3\bar{9}}$ and $C_{6\bar{9}}$
TODVNP	$C^{pp}_{u\bar{v}}$
TOSVNP	OC
LRTP	C^{p}_{46}
LWTP	C^{p}_{13}
LTP	C^{p}_{13} and C^{p}_{46}
TOLCP	C_{69}
CDP	$C^{-p}_{3\bar{8}}$
CUP	$C_{u\bar{v}}$
VTP	C_{13} and $C_{1\bar{9}}$
VRWTP	C_{46}
AVCP	$C_{3\bar{9}}$ and $C_{6\bar{9}}$
RIP	C^{p}_{17}, C^{p}_{47}
WIP	$C^{pp}_{6\bar{8}}$
CRP and DIVCP	$C_{u\bar{v}}$
A3P	$C^{p}_{4\bar{9}}$
PAR	C^{p}_{49}
PAWP	$C^{p}_{3\bar{9}}$ and $C^{p}_{6\bar{9}}$
FRTPP	C^{p}_{19} and C^{p}_{49}
OP (Owner Protocol)	C^{p}_{u9}
PAP	C^{p}_{u3} and C^{p}_{u4}
VWTP (vectorized version of LWTP)	C_{13}
VRTP (vectorized version of LRTP)	C_{46}

Table 8: Summary of some key examples

Chapter 7

Programmability of Weak Consistency Models

The underlying behavior of distributed applications may require a weaker consistency memory than sequential consistency for its correct execution. This may be caused by various reasons including (i) access patterns or restrictions to the shared objects, or (ii) use of specific synchronization mechanisms or patterns in the programming layer. Indeed, because of these, it is possible to execute programs and obtain sequentially consistent results on weaker memories. We make a purposeful distinction between a consistency model and a sequentially consistent result obtainable from sequential consistency. Because of program design, it is possible to obtain a sequentially consistent result on shared memory that is implementing a weaker consistency, such as the weak consistency models presented in the various hierarchies.

It is natural to expect a stronger consistency model in the hierarchy to cost more (in space and/or time) to implement. Hopefully, in return, a stronger consistency model is more easily programmable. So, part of the reasons for our deriving the taxonomy of consistency hierarchies is to create a medium to analyze the consistency support required by various types of application programs. In some sense, the augmentation or weak consistency hierarchies come naturally and pinpoint specific relationships in the global view whose consistency must be preserved. Bearing this in mind, one could scrutinize the actual consistency needs of various existing parallel

algorithms or use it to derive new algorithms. This has the obvious advantage of preserving the ease of programming associated with sequential consistency even if weaker consistency models are used.

The consistency needs of the applications may be determined in a number of ways. In this chapter, we do a static analysis of the access restrictions based on the number of readers/writers. This static analysis has the advantage of simplicity and, since these characteristics can be deduced automatically by some pre-processing tools, an appropriate consistency model can be chosen at compile time.

7.1 Readers/Writers based restrictions

Access restriction applied to a shared object can be asymmetrical. For example, a shared object can be written by a single process (single-writer) or read by a single process (single-reader). In this section, we scrutinize the actual consistency needs of various existing parallel algorithms by analysing their access restrictions. We determine the consistency model required for some application to produce sequentially consistent results. In particular, we show that under some access restriction, C_{39}^{p} and C_{46}^{p} are sufficient to guarantee sequentially consistent results.

7.1.1 Single-reader Variables and Algorithms

When applications use only single-reader variables, sequentially consistent results can be obtained under weaker memory models. The following theorem establishes the implications of single-reader algorithms and the relaxation of consistency requirements.

Theorem 10 : *C_{46}^{p} in a system involving only single-reader variables ensures sequential consistency.*

Proof: It suffices to show that GV_{46}^{p} is an acyclic possible view. (i) From the definition of C_{46}^{p}, we know that in the acyclic GV_{46}^{p}, $x?v \xrightarrow{p} x?v' \Rightarrow x?v \rightarrow x!v'$. Hence from (i) and from the fact that reads can only be issued by a single process,

GV_{46}^p contains the following well-ordering of writes and reads of a given variable: $x!v_1 \rightarrow X?v_1 \rightarrow x!v_2 \rightarrow X?v_2 \rightarrow \ldots \rightarrow x!v_k \rightarrow X?v_k \ldots$ Hence the claim. ■

Protocol LRTP presented in Section 6.2.5 is sufficient to implement sequential consistency with single-reader applications.

7.1.2 Single-writer Variables and Algorithms

The following theorem establishes the implications of single-writer algorithms and the relaxation of consistency requirements.

Theorem 11 : *C_{39}^p in a system involving only single-writer variables ensures sequential consistency.*

Proof: It suffices to show that GV_{39}^p is an acyclic possible view. (i) From the definition of C_{39}^p, we know that in the acyclic GV_{39}^p, $x!v \xrightarrow{p} x!v' \Rightarrow X?v \rightarrow x!v'$. Hence from (i), GV_{39}^p contains the following well-ordering of writes and reads of a given variable: $x!v_1 \rightarrow X?v_1 \rightarrow x!v_2 \rightarrow X?v_2 \rightarrow \ldots \rightarrow x!v_k \rightarrow X?v_k \ldots$ Hence the claim. ■

There are various well known algorithms that use single-writer variables.

Example 1: Bakery Algorithm

The classical bakery algorithm [12] employs shared variables, each writable by a single process, and readable by all processes (single-writer/multiple-reader variables). Hence, by default, all writes to a given variable are program ordered (i.e., $x!v \xrightarrow{p} x!v'$). From Theorem 11, the bakery algorithm can be executed correctly under C_{39}^p.

Example 2: Two-Process Mutual Exclusion

Consider the following two-process mutual exclusion example:

Process i

repeat
    flag[i] := true;
    while (flag[j]) **do** nothing;
    critical section
    flag[i] := false;
    non-critical section
until false

Since all variables are single-writer variables, we know from Theorem 11 that a protocol that implements C_{39}^{p} would correctly execute the above, and mutual exclusion would always be ensured. However, this algorithm suffers from possible deadlocks.

Example 3: Peterson's Two-process Mutual Exclusion [54]

```
Process i
repeat
    flag[i] := true;
    turn := j;
    while (flag[j] and turn=j) do;
    critical section
    flag[i] := false;
    non-critical section
forever
```

This algorithm involves single-writer variables *flag[i]* and *flag[j]*, and two-writer variable *turn*. The following is provable:

1. Mutual exclusion can be ensured by C_{39}^{p}.

2. Existence of an acyclic possible view (hence deadlock-freeness) can be ensured if runtime protocol serializes all writes and reads to "turn" according to the write-order, i.e., $turn!v1 \rightarrow TURN?v1 \rightarrow turn!v2 \rightarrow TURN?v2 \rightarrow turn!v3$ ...

Theorem 12 : *Assuming that all writes are serialized, the consistency models* C_{39}^{p} *and* C_{13}^{p} *ensure the correct execution of Peterson's algorithm.*

Proof:

A typical execution of Peterson's algorithm for two processes will provide the following sequence of events in both processes:

$P1:$ ... $flag[1]!0$ $(non-crit)$ ... $flag[1]!1;$ $turn!2;$ $flag[2]2?v1;$ $turn?v2$ ... $(C.S.)$ ...

$P2:$ ... $flag[2]!0$ $(non-crit)$ ... $flag[2]!1;$ $turn!1;$ $flag[1]?v3;$ $turn?v4$ ... $(C.S.)$ ...

Suppose there is an incorrect execution in the algorithm. This execution can only be produced by one of the following states for (v1,v2,v3,v4):

(0,-,0,-): In this state, both processes can enter critical section. However this situation causes the following cycle in the augmented view.

$(flag[1]!0)_1 \rightarrow (flag[1]?0)_2 \rightarrow (flag[1]!1)_1 \rightarrow (flag[2]?0)_1 \rightarrow$
$(flag[2]!1)_2 \rightarrow (flag[1]?0)_2$

(1,1,0,-): In this state, both processes can enter the critical section. However, the following cycle occurs in the augmented view:

$(flag[1]!1)_1 \rightarrow (turn!2)_1 \rightarrow (turn!1)_2 \rightarrow (flag[1]?0)_2 \rightarrow (flag[1]!1)_1$

(0,-,1,2): In this state, both processes can enter the critical section. However, this situation is identical to the preceding one, and causes a cycle in the augmented view.

(1,1,1,2): In this state, both processes enter the critical section. However, the following cycle occurs in the augmented view:

$(turn!2)_1 \rightarrow (turn!1)_2 \rightarrow (turn!2)_1$

(1,2,1,1): In this state, a deadlock occurs. There is no cycle in the augmented view. The program works if we guarantee a total order of writes on a single object.

No other state causes an incorrect execution.

■

7.1.3 Single-writer/Single-reader Variables and Algorithms

Even if it is very limited, it is possible to imagine a parallel application using only single-reader/single-writer variables. In such a case, from the single-writer and the single-reader cases mentioned earlier, we know that either the C_{46}^{p} or C_{39}^{p} memory model is sufficient to provide sequentially consistent execution for these applications.

7.2 Special Forms of Synchronization

Barrier synchronization is a special form of synchronization used in many parallel programs such as partial sum computation, matrix multiplication, iterative linear solver, sort, etc. In this section, we analyze the use of barrier synchronization in parallel algorithms and its implications for consistency requirements. It is conceivable

that similar results can be derived, for other types of synchronization which do not involve strict barriers to all processes, but we will not include such results in this thesis.

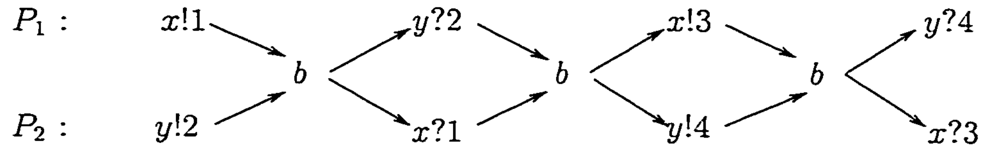
7.2.1 Modeling of Barrier Synchronization

Barrier synchronization can be modeled as an abstract event $[b]$ that occurs in all the processes simultaneously. Hence corresponding b events from the processes are merged to become a single (and same) atomic event.

In other words, suppose we have the following execution:

P_1 :	$x!1$; b ; $y?2$; b ; $x!3$; b ; $y?4$
P_2 :	$y!2$; b ; $x?1$; b ; $y!4$; b ; $x?3$

Then the global view is formed by merging the two b-events to become one:



Hence the above execution can be decomposed into a sequence of phases separated by b events:

Phase 1 ; b ; Phase 2 ; b ; Phase 3 ; b ; Phase 4 ...

A parallel algorithm based on barrier synchronization is often based on the previous "phased" structure.

7.2.2 Implementation of Barrier Synchronization

A common implementation of barrier synchronization is to use a centralized coordinator that handshakes with individual processes synchronously. In particular, process P_i and the coordinator C perform the following for each abstract b event:

P_i :	$r[i]!1$; $a[i]?v <> 1$; ... ; $a[i]?1$
C :	$r[1]?v <> 1$; ... ; $r[1]?1$; $r[2]?v <> 1$; ... ; $r[2]?1$; ... ; $r[n]?1$; $a[1]!1$; $a[1]!1$; ... ; $a[n]!1$

In the above coordinator code, it is conceivable that the coordinator may be made nondeterministic in the order in which it reads $r[i]$ although the writing of $a[i]$ can occur only after all the reads return a value of 1.

It is easy to verify that if we take $a[1]!1$ as the abstract event b, then the global view with all process events contains the abstracted global view in Section 7.2.1 involving other read and write events and b events only.

There exist many implementations without a centralized coordinator [12]. However, they all have the same consistency requirements.

7.2.3 Correctness Requirement (CR) of Barrier Synchronization Based Algorithms

An important correctness requirement (CR) in parallel algorithms based on barrier synchronization is that if $x?v$ occurs in phase i, then v must be associated with some $x!v$ in phase $j < i$ and $x!v'$ does not occur in any phase k satisfying $j < k < i$, assuming that conflicting operations cannot occur in a common phase. In other words, $x?v/x!v$ and $x!v'$ cannot occur in different processes in a common phase.

Theorem 13 : *$x!v/x?v \rightarrow b \rightarrow x!v' \Rightarrow X?v \rightarrow x!v'$ implements CR*

Proof: This is immediate. Suppose otherwise and we have a violation of CR: $x!v/x?v \rightarrow b \rightarrow x!v' \rightarrow b \rightarrow x?v$. Then the augmentation of global view will result in a cycle involving $x!v'$ and the latter $x?v$. ■

Theorem 14 : *Protocol EAUP implements CR.*

Proof: AUP with FIFO channels implements $x!v/x?v \rightarrow b \rightarrow x!v' \Rightarrow X?v \rightarrow x!v'$. Since conflicting operations cannot occur in a common phase, $X?v$ cannot occur in the same phase as $x!v'$. If some $x?v$ occurs after $x!v'$, then the reading process has received v after it has received v'. But this contradicts $x!v/x?v \rightarrow b \rightarrow x!v'$. Hence the claim. ■

Hence in general, synchronous algorithms involving barrier synchronization of the form modeled here can be correctly executed in a distributed system involving simply FIFO channels and asynchronous updates of local copies.

7.3 Other Applications

Many applications have been used to evaluate the performance of DSM [1, 11, 20, 22, 31, 32, 35, 37, 36, 39, 43, 48, 52, 56, 58]. The most used applications are :

- EP (embarrassingly parallel), MG (multigrid), 3d-FFT, IS (integer sort), CGM (conjugate gradient method) from the NAS benchmarks [15];
- Cholesky from the SPLASH benchmarks [55];
- Water, LU, Barnes and Ocean from the SPLASH-2 benchmarks [59];
- Matrix multiplication;
- SOR (successive over-relaxation);
- linear equation solver (simple approach);
- QS (Parallel quicksort);
- TSP (traveling salesman problem) ;
- Gauss.

Many of these applications are "whole applications" that use many different parallel algorithms. As an example, the Ocean application uses a SOR algorithm. These applications are quite complex and are very difficult to analyze. A complete study of the consistency requirements of these applications is beyond the scope of this thesis.

However, many of these applications strictly use barrier synchronization. In particular, applications such as IS, 3D-FFT, MG, CGM and Gauss [11] use only barrier synchronization. Hence, the consistency model required for barrier synchronization as established in the preceding section, can be applied to these applications. Moreover, many of the other applications, such as Water, QS and TSP, require mutual exclusion and in some cases mutual exclusion and barrier. In such cases, it is likely that the consistency requirements established for the mutual exclusion algorithm are applicable to those algorithms.

Chapter 8

Performance Evaluation of Neighbor and Flush Protocols

In Chapter 2, we have presented protocols that implement sequentially consistent memory systems. In this chapter, simulations are performed to evaluate the performance of the synchronous and flush protocols. The simulations are done to evaluate the efficiency of the two protocols under different operating conditions. The varying conditions for the simulation include (i) network communication delays, (ii) computation versus communication time in the user application, (iii) varying degrees of sharing modeled by access graphs, and (iv) typical application benchmarks.

Simulations are also performed on three other protocols. These protocols form a good basis for comparison.

8.1 The Simulator

The simulation is written in Java and built around a discrete event simulation package called Javasim [45]. It consists of three basic components: the shared memory kernel, the network simulator, and the application simulator. This section will present some relevant detail before analyzing the simulation results.

8.1.1 The Shared Memory Kernel Simulator

Five versions of different protocols are simulated as the kernel support. They are respectively:

- General 3-phase protocol:

  Each write operation is a 3-phase write that is broadcast to all processes, without taking advantage of the information of the access graph of the user application. Hence this is expected to be the worst performing protocol and serves as an upper bound of the execution time of the simulated system.

- Restricted-synchronous protocol:

  This is the synchronous protocol presented in Section 3.2 without making full use of neighbor information. In particular, the 3-phase handshaking is restricted to those processes actually sharing a given object. The writer always delays later operation until the current operation has ended.

- Synchronous (neighbor) protocol:

  This is the synchronous protocol presented in Section 3.2. It differs from the restricted-synchronous protocol in that a writer delays a later operation only if the latter lies in the same access cycle as one of its readers which has not yet acknowledged.

- Flush protocol:

  This is the flush protocol as described in Section 3.2.

- Asynchronous protocol:

  The asynchronous protocol is formed by removing all handshaking among readers and writers. In particular, a writer simply broadcasts its new value, and a reader reads the local copy at any time, asynchronously. This protocol obviously does not implement sequential consistency but the resulting performance will serve as the lower bound of the execution time of the simulated system.

8.1.2 The Network Simulator

The shared memory protocols in Section 8.1.1 are simulated on a simulated network environment. A single performance metric, the total execution time of the simulated application, is chosen to analyze system performance. Hence we do not need a detailed simulator such as that in [57]. To account for realistic communication delay, we use the same approach as that in [17, 38]. The sending and receiving of a message incurs a delay D. Hence a 2-phase handshake between a writer and a reader incurs a communication delay of 2D. A 3-phase operation involves the writer that broadcasts a message and receives n acknowledges before broadcasting a commit message. The total communication delay is therefore (n+2)D, assuming that an ethernet-like broadcast channel exists and that each broadcast incurs a delay of D. These assumptions are similar to those used in [17, 38] when the size of a packet is small. We make a simplifying assumption by ignoring congestion and retransmission. The latter could be modeled and simulated as well but is unlikely to make a difference in comparing the performance of different protocols.

8.1.3 The Application Simulator

The application simulator drives the shared memory kernel based on (i) the choice of protocol used, (ii) the behavior of the application being simulated, and (iii) the static access graph supplied with the application.

Behavior of the Application and Access Graph

The behavior of an application process consists of a sequence of read and write operations to be performed. These are either synthetically generated or derived from some known applications.

1. Synthetic applications

   A pure synthetic application generator is used to generate different behaviors to be tested. Each process repeatedly executes a computation phase followed by a

read/write operation chosen randomly. During a computation phase, a process can perform any operation except shared memory access. The duration of a computation phase is normally distributed with a mean of 5 time units. Figure 44 shows the various access graphs used in the synthetic applications.

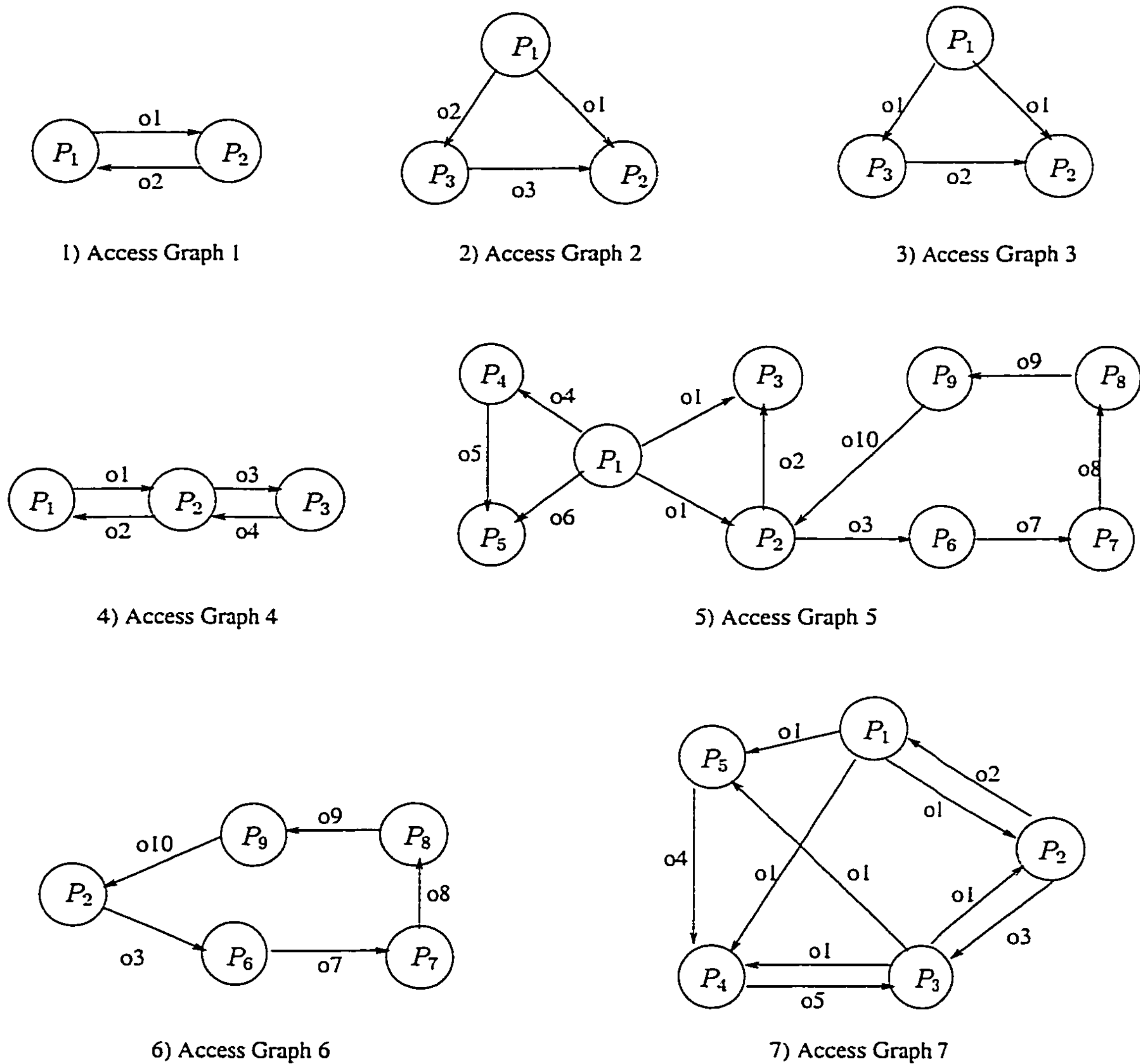


Figure 44: Synthetic application's access graphs

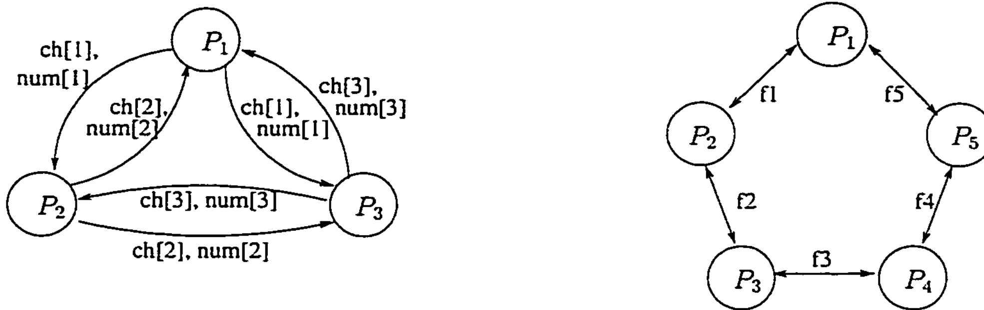
2. Mutual exclusion

Lamport's bakery algorithm [12] for critical sections is simulated here. In its general form, the access graph is a fully connected graph as shown in Figure 45(a). In the access graph, ch[i] represents the variable indicating that

process P_i is choosing his number, and num[i] represents the number picked up by process P_i. The variable parameters in this application include the computation delays associated with the critical and non-critical sections.

3. Dining philosopher

   A distributed mutual exclusion algorithm for the dining philosopher problem [12] is simulated. The access graph here is a single ring as shown in Figure 45(b). In the access graph, variables fi represent the forks of process P_i. As in the previous case, the computation delay of a process is a variable parameter in the simulation. By varying this, we achieve different degrees of concurrency in computation among the processes.



a) Access graph of Lamport's bakery algorithm b) Dining philosopher's access graph

Figure 45: Mutual exclusion's and dining philosopher's access graphs

8.2 Analysis of Results

The results of various simulation runs are illustrated in Figures 46, 47, 48 and 49. Figures 46 and 47 show the simulated performances of the protocols for the synthetic applications. Figure 48 shows a typical result for the mutual exclusion application and Figure 49 shows that of the dining philosopher application. In general, we expect the restricted-synchronous, synchronous (neighbor), and flush protocols to outperform the general 3-phase because of their avoidance of unnecessary synchronization and abilities to hide long access latency with the computation or among accesses.

In the case of the restricted-synchronous protocol, it reduces synchronization cost by restricting reader/writer synchronization among relevant processes. Hence each acknowledgment phase will be faster. In the synchronous neighbor protocol, two accesses from a process may overlap if they do not lie in the same access cycle. Hence synchronization delays of program-ordered accesses can overlap among themselves as well as with the computation phase of the process. In the flush protocol, not only is synchronization restricted to those processes that are related, but also each access does not delay subsequent accesses, except in the case of the flush. Hence, all synchronization delays except the flush are hidden. The cost of synchronization will surface in the latter case and it is localized to the access cycles that the flush operation controls.

The results of the synthetic applications more or less substantiate the above expectations. Seven different access graphs are simulated. Access graph 7 in Figure 44 contains more access cycles, whereas the rest of the access graphs are rather simple. Generally, the flush protocol outperforms the rest except in graph 6 which contains significantly more processes in a single access cycle. In that case, the synchronous neighbor protocol gives the best result, as synchronization between two neighbors is more effective than invoking a 3-phase flush involving a relatively large set of processes when that operation is performed. The effectiveness of the synchronous neighbor protocol is also noteworthy in Figure 46, demonstrating that using knowledge of access cycles to hide access latency is an effective strategy.

The mutual exclusion simulation is performed with some small changes from the synthetic simulation. As the access graph for this application is a complete graph, there is no difference between the general 3-phase protocol and the restricted-synchronous protocol. To ensure progress, the asynchronous protocol is not meaningful. So it is not included in the evaluation. Simulation is performed for different combinations of communication delays and number of processes. The results for different communication delays are very similar, and a typical comparison is plotted in Figure 48. In general, the flush protocol outperforms both the general 3-phase and the synchronous neighbor protocols, and the effects are more significant as the number of

processes increases. This is understandable as the frequency of synchronization, and program-ordered delays, become more significant with an increase in the number of processes.

The dining philosopher problem is in the opposite spectrum when compared with the general mutual exclusion problem as object sharing is more localized and precise access cycles exist. Hence, as the results in Figure 49 confirm, all three protocols that make use of the results of this paper perform well, and the synchronous neighbor protocol is the best of the three. The access graph of the dining philosopher problem contains cycles between two neighbors as well as global cycles involving all processes. Hence flush synchronization in various access cycles may result in more non-hidden delays than the synchronous neighbor protocol.

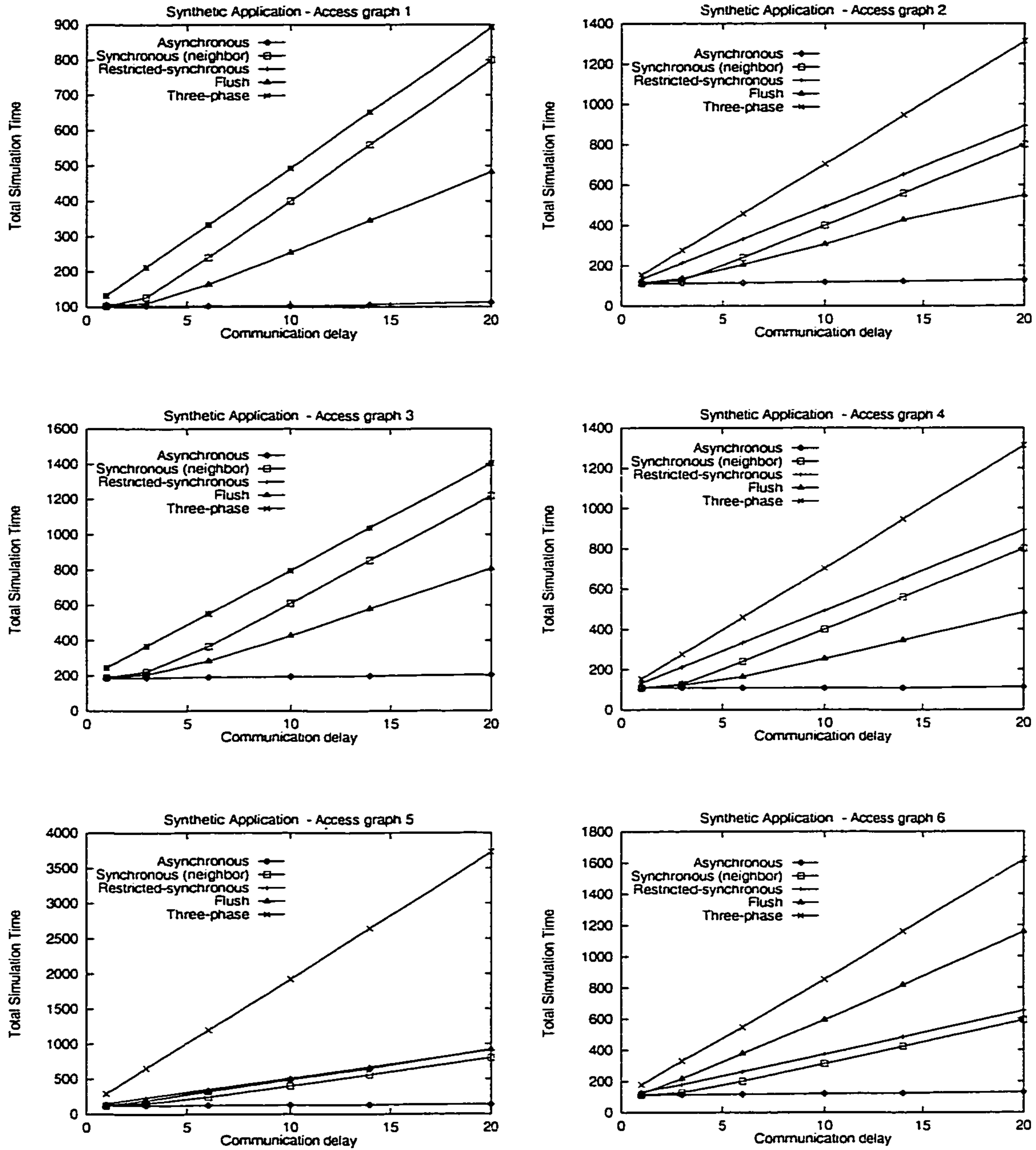


Figure 46: Synthetic application simulation results (access graphs 1-6)

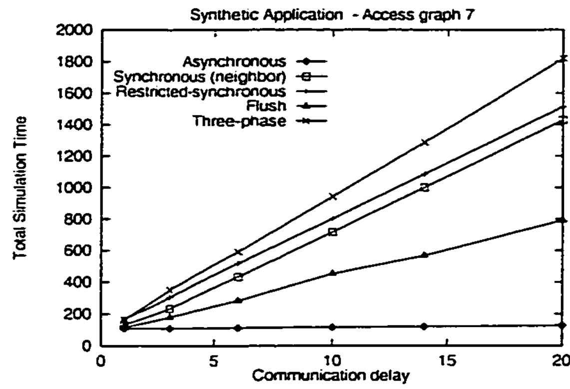


Figure 47: Synthetic application simulation results (access graph 7)

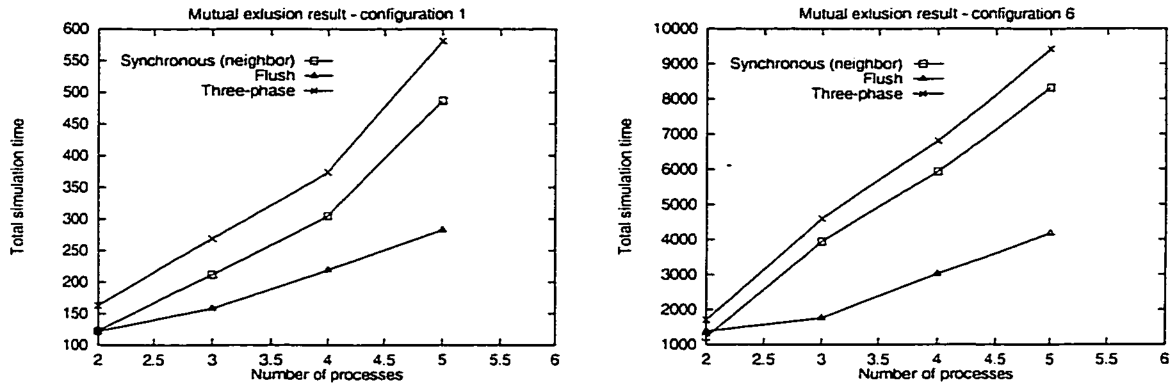


Figure 48: Mutual exclusion simulation results for two configurations

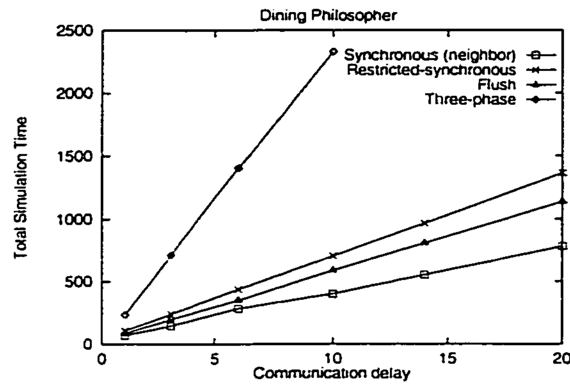


Figure 49: Dining philosopher simulation results

Chapter 9

Implementation Considerations

In Chapters 2 and 3 we have introduced and used the concept of virtual access graphs to develop efficient algorithms for sequentially consistent DSM. Chapter 8 has reported simulation experiments that show significant improvements in performance in our protocols, especially for the flush protocol.

For these preliminary evaluations, the construction as well as the analysis of the virtual access graphs of the chosen applications were done manually. The resulting synchronization was then manually coded in the simulator. In this chapter, we discuss how all the construction and analysis of the virtual access graph can be done automatically and how the proper synchronization can be provided by the run-time system.

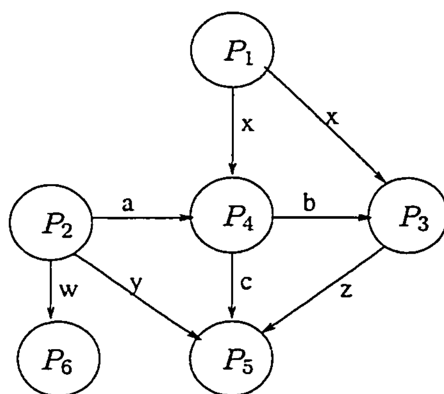
9.1 Virtual Access Graph Construction and Analysis

A possible procedure to implement the automatic treatment of virtual access graph and of synchronization could be for some pre-processor to add the proper synchronization directly into the application. In such a case, the DSM system does not provide any consistency model.

The pre-processor must first generate the virtual access graph. It then uses the

order relation introduced for the protocols (program order, flush order, ...) and the information provided by the virtual access graph to insert the necessary synchronization. The information provided by the virtual access graph can be extracted by a pre-processor, and inserted into some tables associated with the processes. Each entry of a table contains all the necessary information that must be known to avoid inconsistencies. Using these tables, the pre-processor can statically add synchronization operations into the application program and leave it to the run-time system to take care of the rest of the synchronization. A run-time system can use the table to dynamically determine the necessary actions to preserve consistency. However, the information needed from the virtual access graph may be different according to the protocol used. For example, the information required to implement the proper synchronization for the flush protocol is different than that for the neighbor protocol.

In order to illustrate the idea presented in this chapter, we use the example shown in Figure 50. It contains a virtual access graph and all the virtual cycles associated to it.



Virtual access graph

1. $P_2 \overset{a}{-} P_4 \overset{c}{-} P_5 \overset{y}{-} P_2$
2. $P_2 \overset{a}{-} P_4 \overset{b}{-} P_3 \overset{z}{-}] P_5 \overset{y}{-} P_2$
3. $P_2 \overset{a}{-} P_4 \overset{x}{-} P_1 \overset{x}{-} P_3 \overset{z}{-} P_5 \overset{y}{-} P_2$
4. $P_4 \overset{b}{-} P_3 \overset{z}{-} P_5 \overset{c}{-} P_4$
5. $P_4 \overset{x}{-} P_1 \overset{x}{-} P_3 \overset{b}{-} P_4$
6. $P_4 \overset{x}{-} P_1 \overset{x}{-} P_3 \overset{z}{-} P_5 \overset{c}{-} P_4$

Virtual cycles

Figure 50: Example of virtual access graph

9.1.1 Virtual Access Graph Construction

To construct the virtual access graph of a given distributed program (before or after compilation), we require that every process declare the set of shared variables used.

Static analysis is performed to identify whether an object is of the type read-only, write-only, or read/write. From this analysis, access tables can be constructed for each process. Using these tables, the complete virtual access graph is then built. Figure 51 shows the access table for the virtual access graph of Figure 50. Each line of the table contains all the objects written by a particular process and each column contains all the objects read by a particular process.

Process	P_1	P_2	P_3	P_4	P_5	P_6
P_1			x	x		
P_2				a	y	w
P_3					z	
P_4			b		c	
P_5						
P_6						

Figure 51: Access table

Once the virtual access graph is constructed, the proper synchronization information must be extracted from it. Since the synchronization is required only along virtual cycles, the extraction requires an analysis of the virtual cycles. This operation is done differently for each protocol.

Proper synchronization to avoid view cycles on a single edge can be easily implemented by the run-time system by using FIFO channels or timestamps without any other synchronization operations. Hence, the following discussion considers only the treatment of virtual cycles.

9.1.2 Neighbor Protocol

There are two types of savings in the neighbor protocol. First, synchronization traffic is reduced to involve accesses that lie on virtual cycles. Second, blocking is reduced inside a single process. Blocking of a later operation in a same process, say P_k, is deemed unnecessary unless the two operations lie in some virtual cycle. Essentially, the program order of the memory operations of a process is replaced by a partial-order; two instances of operations, say op_i and op_j (by processes P_i and P_j respectively), are

ordered $op_i < op_j$ iff they are in program order and lie on a common virtual cycle. This graph problem can be solved by checking if the nodes, P_i and P_j, connected to P_k via op_i and op_j are 1-connected (other than through P_k). This is done by checking if the removal of some other node will disconnect these two nodes. This is solvable by iteratively deleting all the other nodes and testing if the two nodes are still connected. Hence the problem possesses a polynomial time solution (complexity of $O(n^2)$). This algorithm is used to determine if two operations must be synchronized and also the processes involved in these operations.

To generate this partial order, the virtual access graph has to be processed so that for every pair of distinct accesses from a process, this dependence/independence can be recorded for runtime use. Specifically, by analyzing the virtual access graph, a table is constructed for process P_k such that (op_i, op_j) is in the table if op_i and op_j are two instances of operations by P_k such that they are associated with two edges that lie in a same virtual cycle. Moreover, the table contains the processes involved in the operations. The runtime system of a node executing P_k would block the commit of op_j if an earlier op_i has not yet committed. Alternately, we can directly tag this information in the code so that the partial order is explicitly specified in the program code, as in scope consistency [34].

For example, in Figure 50, process P_1 writes into object x which is read by processes P_3 and P_4. Since this object is involved in a cycle and has two readers, P_1 must use a (3-phase) write operation involving processes P_3 and P_4. This operation does not need to be synchronized with other operations. In the same example, process P_2 writes objects a, y and w. Object w is not involved in any cycle, while the other two objects are involved in cycles involving a single reader for each object. So, P_2 uses a write on w without having to wait for any acknowledgement, and (2-phase) writes on a and y that require each acknowledgements from processes P_4 and P_5 respectively. Since a and y are involved in the same cycle, any $a!v$ must end before a $y!v1$ can begin and vice versa. Figure 52 shows for each process a table containing all the necessary information for the neighbor protocol. Each line of these tables represents an object. Each column provides one item of the information as presented earlier. The first

column contains the object to consider. The second column, entitled "**Op**", contains the operation on that object. The third column, entitled "**Ack from**", contains the set of processes that must acknowledge the operation. When no acknowledgement is required, the operation is asynchronous. In a read operation, an acknowledgement indicates that the operation must wait for the arrival of the commit message. Finally, the last column, entitled "**Synchronized with**" contains the objects (and their respective operation) with which the operation on this object should be synchronized (program order must be enforced). So, in the table of process P_4, we know that $b!v$ must be acknowledged by process P_3 and must be synchronized with the operations $x?v1$, $a?v2$ and $c!v3$. In essence, this means for the neighbor protocol that the operations $x?v1$, $a?v2$ and $c!v3$ cannot end before the operation $b!v$ has been acknowledged by P_3.

9.1.3 Flush Protocol

The information needed for the flush protocol is quite different from that for the neighbor protocol. We need to find a set of flush operations that covers all the virtual cycles and the set of operations covered by each flush operation.

Savings in the flush protocols are obtained by the asynchronous nature of all non-flush operations. Hence, the careful choice of the flush operations is an important implementation issue and an interesting optimization problem. To minimize the use of flush-accesses, one strategy is to find a minimal set of operations (edges) whose deletion from the virtual access graph produces an acyclic graph. For example, in Figure 50, we can delete three edges, say a, z and b and make all writes into theses objects, by P_2, P_3 and P_4 respectively, as flush-writes, and allow all the other operations to be non-blocking (asynchronous). Figure 53 shows this particular choice of flush operation and all the operations covered by each flush operation. The write operations used in these tables are the same as the ones presented in Figure 52, i.e., they require the same acknowledgement. For now, we do not use read operations as flush. When a flush is executed, all operations it covers must end before the flush can continue and none of these covered operations can proceed until the flush ends

Process P_1

Object	operation	Ack from	Synchronized with
x	$x!v$	P_3, P_4	

Process P_2

Object	operation	Ack from	Synchronized with
a	$a!v$	P_4	$y!v1$
w	$w!v$		
y	$y!v$	P_5	$a!v2$

Process P_3

Object	operation	Ack from	Synchronized with
x	$a?v$	P_1	$b?v1$, $z!v2$
b	$b?v$		$x?v1$, $z!v2$
z	$z?v$		$b?v1$, $x?v2$

Process P_4

Object	operation	Ack from	Synchronized with
x	$x?v$	P_1	$a?v1$, $b!v3$, $c!v2$
a	$a?v$		$b!v_1$, $c!v2$, $x?v2$
b	$b!v$	P_3	$a?v1$, $c!v2$, $x?v3$
c	$c!v$	P_5	$a?v1$, $b!v2$, $x?v3$

Process P_5

Object	operation	Ack from	Synchronized with
c	$c?v$		$y?v1$, $z?v2$
y	$y?v$		$c?v1$, $z?v2$
z	$z?v$		$c?v1$, $y?v2$

Figure 52: Synchronization tables for the neighbor protocol

(commits). This means that, in our example, when the flush operation $(z!v)_3$ is executed, the operations $(c!v1)_4$ and $(x!v2)_1$ covered by the flush must end before the flush ends. It also blocks any new execution of $(c!v1)_4$ and $(x!v2)_1$ until the flush ends.

Process P_2

Flush operation	Operation flushed
$a!v$	$(c!v1)_4, (b!v2)_4, (y!v3)_2, (z!v4)_3, (x!v5)_1$

Process P_3

Flush operation	Operation flushed
$z!v$	$(c!v1)_4, (x!v2)_1$

Process P_4

Flush operation	Operation flushed
$a!v$	$(c!v1)_4, (z!v2)_3, (x!v3)_1$

Figure 53: Flush tables

The choice of a set of flush operations can be solved heuristically by repeatedly choosing an edge to delete until the acyclicity requirement is satisfied. Then for these sets of edges, we need to identify the edges that they flush. The algorithm works as follows:

1. Let G be the virtual access graph.
2. Randomly pick an edge a and decide $S(a)$ = set of edges which can be in some cycle with a. $S(a)$ is the set of edges flushed by a. Include a in F, the set of flush writes. Delete a from G and repeat 2 until the graph becomes acyclic.

According to this algorithm, it is possible for some edges to be flushed by multiple operations. But it is difficult to reduce the number of flush operations on a single edge. The complexity of the algorithm is polynomial since checking if two edges are on some cycle can be decided by asking if the deletion of some node in G (including the four edge nodes of the two edges) will disconnect some of the node(s) among of the four nodes. A problem answerable in $O(n^2)$ time. Moreover, checking if two nodes

lie on a cycle can also be decided by determining if the two nodes are 1-connected. A problem answerable in $O(n^2)$ time as well.

9.1.4 Choice of Synchronization

Detailed implementation of proper synchronization may involve different uses of asynchronous message passing (fast-read and fast-write), 2-phase or 3-phase synchronization, depending on the virtual access graph. 2-phase synchronization requires a process to send a message, and wait for the acknowledgement before completing the operation. On the receiver side, the operation is considered complete as soon as the acknowledgement is sent. 3-phase synchronization requires a process to send a message, waits for all required acknowledgements, and then sends a commit message before completing the operation. On the receiver side, the operation is considered complete only when the commit message is received. This implies blocking some operations in our implementations of the two protocols. Specifically, a synchronous remote write is delayed until its readers have acknowledged.

In general, operations not lying in any virtual cycle are implemented as asynchronous operations and hence are totally non-blocking. For other operations that lie in some virtual cycle, their implementation may vary. In the neighbor protocol, a single-reader object in a virtual cycle requires only 2-phase synchronization, while a multiple-reader object would require 3-phase synchronization. For the example of Figure 50, a write of a requires only 2-phase synchronization while a write of x requires 3-phase synchronization.

For the flush protocol, asynchronous operations are used except those labeled as flush. A flush operation requires 3-phase synchronization, involving all processes lying in the same virtual cycle "synchronized" by this flush. Hence once a flush operation has been heard by a process in the same virtual cycle, the latter cannot start any more operations until its previous operations have completed and it has acknowledged the flush.

The replacement of memory read and write operations by special calls using asynchronous read/write, 2-phase write, 3-phase write or flush can be automated by making use of the results outlined earlier.

9.2 Conclusion

In this chapter, we have presented simple techniques to implement the automatic construction and analysis of virtual access graphs. These techniques can be used by a compiler to automatically provide the proper synchronization for each application. These algorithms are not optimal. Possible optimizations are briefly introduced in Chapter 10.

Chapter 10

Conclusion

In recent years, much research has been done on software and hardware distributed shared memory. Numerous consistency models and their implementation have been proposed. In this chapter we summarize the contributions of this thesis to the field of DSM consistency models and their implementation. We also discuss some limitations of our approach that may open opportunities for future work.

Contributions

In this thesis, we have formalized a notion of local and global views that are useful for studying consistency requirements of shared memory and their subsequent implementations. A minimal consistency model was introduced which is interesting in its own right: it corresponds to simple asynchronous updates among the distributed processes. However, to successively generate stronger consistencies (and less programmed synchronization), a global view should remain consistent (acyclic) under different augmentation rules. These augmentation rules generate a hierarchy of consistency models. Besides acyclicity, some ordering ("causal") relations may also be asserted of a global view. The latter form the weak consistency hierarchy that is related to causal memory. By understanding these models from the perspective of views, it is possible to derive appropriate implementation protocols as well as to use them in programming.

A few attempts [19, 29, 39, 46, 51] have been suggested to define and compare shared memory consistency models. The view model presented in this thesis represents an original attempt that distinguishes itself from others by its use of logical order (view precedence) rather than time precedence that applies in the execution world. Hence, an important property of this model is that not all conflicting operations need to be ordered in the global or augmented views. This property gives the view concept a high level of flexibility in expressing consistency requirements rather than runtime ordering.

The results are attractive: the hierarchies so developed cover a broad spectrum of consistency models proposed to date and lead to new ones introduced in this thesis. Indeed, even though some comparisons and classifications were proposed in some earlier literature [19, 29, 33, 39, 47, 51], they were done only among existing models and the classifications were done in a rather ad hoc manner. Our approach is systematic and logical, and should lay a clear foundation in the study of all inter-related concepts in shared memory consistency, its use and implementation.

In this thesis, we have also presented two novel protocols that implement sequential consistency in a distributed memory system with replications in reader sites. Our protocols are update-based protocols but do not use atomic-broadcast or 3-phase on each update unlike the protocols introduced in [2, 4, 13, 14, 21].

To our knowledge, our protocols, particularly the flush protocol, are the only ones that provide as much asynchrony among operations in a process without using labeled operations as in relaxed consistency models [2, 3, 13, 25, 34]. In most protocols, all update operations are blocking. Some protocols [4, 46] allow asynchrony on write operations issued by the same process. Our flush protocol allows asynchronous operations not only between writes but also between write and read operations issued by the same process.

To increase asynchronicity in our protocols, we have used a new strategy to minimize synchronization cost and maximize the hiding of synchronization delays in a process. The strategy is based on the knowledge of spatial locality in the sharing of memory objects. An *access graph* is used to capture the sharing relationship among

processes via the shared objects.

Future Work

The new protocols presented in this thesis are interesting from a performance perspective. There is a need to implement these in a real DSM to validate the results obtained by simulation. In a real platform, much of the runtime and compilation support will have to be automated and optimized.

1. The automatic generation approach suggested in Chapter 9 uses a statically constructed access graph. It is however possible to imagine a dynamic access graph to which edges are inserted and deleted dynamically during execution. This is possible if the tables are used by a run-time system to implement the proper synchronization. In such a case, the run-time system can delete or insert information dynamically into the tables.

   So, a process knowing that it won't use a variable for some time can send a special message to all its neighbors in the virtual access graph related to that object telling them to "forget about me". The neighbors won't need to synchronize along this virtual cycle. When the process wants to use the shared variable, it must issue a synchronous operation saying "I am back", to inform its neighbors to recreate the arcs. This can be implemented with the access tables presented in Chapter 9. A process receiving a "forget about me" message deletes an entry in its access table corresponding to that particular process and the related object. The reverse is done when a process receives the "I am back" message.

   A process can decide to send the message "forget about me" based on some instruction count which can be provided by the compiler. The compiler may also detect some phases in the process and insert the necessary code to send the proper "forget about me" and "I am back" messages.

   When a process removes itself from a cycle, it eliminates the cycle. So it may

be interesting for the process to send a "forget about me" message to all the processes in the just broken virtual cycle (or the message can be forwarded by the neighbors). When a process is coming back, the same broadcasting must be done.

2. Normally, an object modified by a 3-phase write operation requires all read operations on the same object to be blocked until the write commits. We suggest an optimization in which a read operation can always execute asynchronously. For an object x modified by a 3-phase write, this means that a read can prematurely return a value even if it is not committed. However, the next operation, read or write, on any other object involved in a virtual cycle with x may not start before the value is committed. Hence, the next operation acts as a flush for the preceding write.

   We believe that a neighbor protocol using premature reads still guarantees sequential consistency. We intend to prove this claim in the future.

The work done on the view model is still incomplete. Even if we have designed hierarchies of consistency models, the synthesis of implementations for some of these consistency models is not done. Moreover, there are some consistency models that do not fall in our hierarchies. For example, the view model cannot be used to represent *relaxed consistency models* that use special operations to create a partial order in a process.

The extension of the view model to allow special operations is a future research topic. Our results lead us to believe that our approach can easily be extended. A basic characteristic of models that use special operations is that they require each program to be data race free. The special operations, often called *acquire* and *release*, can sometimes be interpreted as a *read-modify-write* operation (RMW) (for acquire), and a simple write (for release).

We believe that the view model can be adapted easily to these special operations. The release, being equivalent to a simple write, does not require any adaptation. The RMW operations may require the addition of a new operation to the view model. A

simple extension is to represent it by a read immediately followed by a write with some additionnal value constraints in the view.

We also believe that these new operations will not affect the definition of the access graph. Indeed a RMW will simply be represented by two edges from and to the issuing process. However, the definition of virtual cycles must be adapted to the labeling of operations and indirectly to the data itself. Indeed, in the relaxed consistency models, objects can be classified into two types: data objects and synchronous objects. Data objects are normal objects accessed through normal read and write operations. Synchronous objects are those typically used to synchronize the access to data objects. These objects are normally accessed through the special operations presented earlier, i.e., RMW for the acquire and write for the release. The classification of objects enables us to introduce three types of virtual cycles:

1. Data virtual cycle whose edges are labeled only with data objects,

2. Synchronous virtual cycles whose edges are labeled only with synchronous objects, and

3. Mixed virtual cycles whose edges are labeled with both data and synchronous objects.

Finally, the definition of data race free programs must be introduced in our approach to correctly represent relaxed consistency models. We know that an execution is data race free if any two conflicting events, op_1 and op_2, on a data object x in the execution are separated by a specific sequence of events (release-acquire) on a synchronous object y. A program is then labeled data race free if all its executions are data race free. This definition implies that there cannot be view cycles involving only data objects. Hence the only access cycles to consider for synchronization are mixed and synchronous cycles.

Bibliography

[1] S.V. Adve, A.L. Cox, S.Dwarkadas, R. Rajamony, and W. Zwaenepoel. A comparison of entry consistency and lazy release consistency implementations. In *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, pages 26–37, February 1996.

[2] S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[3] S.V. Adve and M.D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*, pages 2–14, 1990.

[4] Y. Afek, G. Brown, , and M. Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

[5] D. Agrawal, M. Choy, H.V. Leong, and A. Singh. Mixed consistency: a model for parallel programming. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing, Los Angeles*, pages 101–110, 1994.

[6] M. Ahamad, R.A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. Technical Report GIT-CC-92/34, College of Computing, Georgia Institute of Technology, 1992.

[7] M. Ahamad, J.E. Burns, P.W. Hutto, and G. Neiger. Causal memory. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG-5) (LNCS 579)*, pages 9–30. Springer-Verlag, 1991.

[8] M. Ahamad, P.W. Hutto, and R. John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDSC-11)*, pages 274–281, 1991.

[9] M. Ahamad, G. Neiger, J.E. Burns, P. Kohli, and P.W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9:37–49, 1995.

[10] M. Ahamad, M. Raynal, and G. Thia-Kime. An adaptive protocol for implementing causally consistent distributed services. In *Proceeding of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, pages 86–93, 1998.

[11] C. Amza, A. L. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. *Proceedings of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):467–475, March 1999.

[12] G. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin-Cummings, 1991.

[13] H. Attiya, S. Chaudhuri, R. Friedman, and J.L. Welch. Shared memory consistency conditions for nonsequential execution: Definitions and programming strategies. *SIAM Journal on Compputing*, 27(1):65–89, 1998.

[14] H. Attiya and J.L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.

[15] D. Bailey and et al. The nas parallel benchmarks. Technical Report RNR-94-007, RNR, March 1994.

[16] J.B. Carter. *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency.* PhD thesis, Rice University, 1993.

[17] D.R. Cheriton and W. Zwaenepoel. Distributed process groups in the v kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[18] T. Cornilleau and E. Gressier-Soudan. A combined-consistency approach : Sequential & causal-consistency. *Operating Systems Review*, 30(4):33–44, October 1996.

[19] A. de Melo. Defining uniform and hybrid memory consistency models on a unified frameworks. In *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*, January 1999.

[20] M. R. Eskicioglu, T. A. Marsland, W. Hu, and W. Shi. Evaluation of jiajia software dsm system on high performance computer architectures. In *Proc. of the 32nd Hawaii Hawaii International Conference on System Sciences (HICSS-32)*, January 1999.

[21] A. Fekete, M.F. Kaashoek, and N. Lynch. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. *Journal of the ACM*, 41(1):35–69, January 1998.

[22] S.S. Fu and N.-F.Tzeng. Aggressive release consistency for software distributed shared memory. In *Proceedings of the 17th Int'l Conf. on Distributed Computing Systems (ICDCS-17)*, pages 288–295, May 1997.

[23] V. K. Garg and M. Raynal. Normality: a consistency condition for concurrent objects. *Parallel Processing Letters*, 9(1):123–134, 1999.

[24] K. Gharachorloo, S.V. Adve, A. Gupta, J. Henessy, and M.D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15:399–407, 1992.

[25] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Henessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th annual international Symposium on Computer Architecture*, pages 15–26, 1990.

[26] P.B. Gibbons and E. Korach. The complexity of sequential consistency. In *Proceedings of the fourth IEEE symposium on Paralell and Distributed Processing*, pages 317–325, December 1992.

[27] G. Girard and H.F. Li. Evaluation of two optimized protocols for sequential consistency. In Proceedings of the 32nd Annual Hawaii International Conference on System Sciences, January 1999.

[28] M. P. Herlihy and J. M. Wing. Linearizability : A correctness condition for concurrent objects. *ACM Transactions on Computer Systems*, 12(3):463–492, July 1990.

[29] L. Higham, J. Kawash, and N. Verwaal. Defining and comparing memory consistency models. In *Proc. of the 10th Int'l Conf. on Parallel and Distributed Computing Systems (PDCS-97)*, pages 349–356, October 1997.

[30] M.D. Hill. Multiprocessors should support simple memory consistency models. *IEEE Computer Magazine*, 31(8):28–35, 1998.

[31] W. Hu. Reducing message overhead in home-based software dsms. In *Proceedings of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, pages 7–11, June 1999.

[32] W. Hu, W. Shi, and Z. Tang. Home migration in home-based software dsms. In *Proceedings of the First Workshp on Software Distributed Shared Memory (WSDSM'99), Rhodes, Greece*, pages 21–26, June 1999.

[33] P.W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDSC-10)*, pages 302–311, 1990.

[34] L. Iftode, J.P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. *Theory of Computing Systems*, 31(4):451–473, July/August 1998.

[35] A. Itzkivitz, A. Schuster, and Y. Talmor. Harnessing the power of fast, low latency, networks for software dsms. In *Proceedings of the First Workshp on Software Distributed Shared Memory (WSDSM'99), Rhodes, Greece*, pages 63–69, June 1999.

[36] R. John. Implementing and programming weakly consistent memories. Technical Report GIT-CC-95-12, Georgia Institute of Technology, March 1995.

[37] R. John and M. Ahamad. Evaluation of causal distributed shared memory for data-race-free programs. Technical Report GIT-CC-94-34, Georgia Institute of Technology, March 1994.

[38] R.E. Kessler and M. Livny. An analysis of distributed shared memory algorithms. In *Proceedings of the 9th International Conference on Distributed Computing Systems (ICDCS-9), Newport, CA*, pages 498–505, June 1989.

[39] P. Kholi, G. Neiger, and M. Ahamad. A characterization of scalable shared memories. In *Proceedings of the 22nd International Conference on Parallel Processing*, pages I332–I335, 1993.

[40] L. Lamport. Time, clocks and the ordering of events. *IEEE Communications of the ACM*, 21(7):558–565, September 1978.

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

[42] H.F. Li and G. Girard. A hierarchy of view consistencies and exact implementations. In *Proceedings of the First Workshp on Software Distributed Shared Memory (WSDSM'99), Rhodes, Greece*, pages 109–113, June 1999.

[43] T.-Y. Liang, D.-Y. Chuang, and C.-K. Shieh. Thread selection in software dsm systems. In *Proceedings of the First Workshp on Software Distributed Shared Memory (WSDSM'99), Rhodes, Greece*, pages 47–51, June 1999.

[44] R.J. Lipton and S. Sandberg. Pram: A scalable shared memory. Technical Report CS-TR-180-88, Dept. of Computer Science, Princeton University, September 1988.

[45] R. McNab and F.W. Howell. Using java for discrete event simulation. In Proc. Twelth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW), Univ. of Edinburgh, pages 219–228, 1996.

[46] M. Mizuno, M. Raynal, and J. Zhou. Sequential consistency in distributed systems. In A. Schiper K. Birman, F. Mattern, editor, *Proc. of the Int. Workshop on Theory and Practice in Distributed Systems (LNCS 938)*, pages 224–241. Springer-Verlag, July 1995.

[47] D. Mosberger. Memory consistency models. *ACM Operating System Review*, 27(1):18–26, 1993.

[48] M.C. Ng and W.F. Wong. Adaptive schemes for home-based dsm systems. In *Proceedings of the First Workshp on Software Distributed Shared Memory (WS-DSM'99), Rhodes, Greece*, pages 13–20, June 1999.

[49] M. Raynal and M. Ahamad. Exploiting write semantics in implementing partially replicated causal objects. In *Proceedings of the 6th EUROMICRO Conference on Parallel and Distributed Processing*, pages 157–163, January 1998.

[50] M. Raynal and A. Schiper. From causal consistency to sequential consistency in shared memory systems. In Thiagarajan P.S., editor, *Proc. of the 15th Conference on Foudations of Software Technology and Theorical Computer Science (LNCS 1026)*, pages 180–194. Springer-Verlag, December 1995.

[51] M. Raynal and A. Schiper. A suite of definitions for consistency criteria in distributed shared memories. *Annales des Tlcommunications*, 52(11):652–661, 1997.

[52] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. of the*

7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), pages 174–185, October 1996.

[53] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[54] A. Silbershatz and P.B. Galvin. *Operating System Concepts*. Addison-Wesley, 1997.

[55] J.P. Singh, W.-D. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. Technical Report CSL-TR-92-526, Stanford University, June 1992.

[56] V. Sricharan and R. Govindarajan. Study of cahe and tlb performance in a dvsm system. In *Proceedings of the First Workshp on Software Distributed Shared Memory (WSDSM'99), Rhodes, Greece*, pages 53–57, June 1999.

[57] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5):54–64, May 1990.

[58] J. C. Ueng, C. K. Shieh, and Q. C. Lin. Design and implementation of proteus. In *Proceedings of the First Workshp on Software Distributed Shared Memory (WSDSM'99), Rhodes, Greece*, pages 39–46, June 1999.

[59] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodical considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.